

THE FLORIDA STATE UNIVERSITY  
COLLEGE OF ARTS AND SCIENCES

**ENHANCED SECURITY FOR MOBILE AGENT SYSTEMS**

**By**

**JEFFREY T. MCDONALD**

A Dissertation submitted to the  
Department of Computer Science  
in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

Degree Awarded:  
Fall Semester, 2006

Copyright © 2006  
Jeffrey T. McDonald  
All Rights Reserved

The members of the Committee approve the Dissertation of Jeffrey T. McDonald defended on October 20, 2006.

Alec Yasinsac  
Professor Directing Dissertation

Sam Huckaba  
Outside Committee Member

Michael Murmester  
Committee Member

Lois Hawkes  
Committee Member

Robert van Engelen  
Committee Member

Approved:

David Whalley, Chair, Department of Computer Science

Joseph Travis, Dean, College of Arts and Sciences

The Office of Graduate Studies has verified and approved the above named committee members.

To my loving wife Angela and my two wonderful children, Allie and Tucker. You all are blessings from God and a huge part of my success in life. I love you.

## ACKNOWLEDGEMENTS

*"A man can receive nothing unless it has been given to him from heaven."*

*John 3:26-27*

Knowledge and scientific discovery in the human sense are only an uncovering of the principles by which the Creator fashioned our world. The greatest knowledge we can attain is found only in context to a relationship with God through His Son Jesus, the Messiah. All scientific discovery and intellectual pursuits pail in comparison to the surpassing knowledge of knowing Him as Savior, Friend, and Lord. It is therefore appropriate that I gratefully acknowledge Him first as the source of my intellect, abilities, and all that I am. I truly thank my heavenly Father for the opportunity to pursue and express all the intellectual giftings He has given to me.

I wish to thank several people who have assisted me in my academic journey and that have guided me towards completion of this particular research effort. My advisor, Dr. Alec Yasinsac, is by far the key element to my success. I wish to thank him for his mentorship, constant direction, and continued encouragement. This thesis would certainly have been impossible without his guidance. Thanks are due also to Dr. Mike Burmester and Dr. Breno de Medeiros for many insightful security-related discussions.

Thanks to Willard Thompson, Todd Andel, and Sarah Diesburg for being great officemates and providing constant encouragement. Thanks also to other members of the security research group for their attentiveness and feedback through countless briefings. I also wish to acknowledge my AFIT sponsors for their support and for providing this fellowship opportunity.

Last, and most importantly, I wish to thank my family—my wife Angela, our children Allie and Tucker, for their support during this busy time of my life. They are certainly the other essential element to my success. Thanks Dad, Chris, and John for all your love and prayers. Mom could not be here to read the final version—but her encouragement was indeed crucial to me reaching the finish line. Thanks also to my wife's family for their invaluable help to include a brief stay at the Lodge Hotel and many Southern home-cooked meals.

## TABLE OF CONTENTS

LIST OF FIGURES.....	ix
LIST OF TABLES.....	xiii
ABSTRACT.....	xiv
CHAPTER 1 INTRODUCTION .....	1
1.1    The Problem Area .....	1
1.2    Research Objectives .....	3
1.2.1    Multi-Agent Architectures for Security .....	3
1.2.2    Mobile Agent Trust Frameworks.....	4
1.2.3    Program Encryption.....	4
1.3    Conventions .....	5
1.3.1    Cryptographic Primitives and Protocols.....	5
1.3.2    Boolean Functions and Circuits.....	5
1.3.3    Turing Machines and Programs .....	6
1.4    Chapter Summary .....	6
CHAPTER 2 MOBILE AGENT SECURITY .....	7
2.1    Mobile Agent Paradigms .....	7
2.1.1    Defining Agents .....	7
2.1.2    Defining Mobility .....	8
2.1.3    Mobile Agent Interactions.....	9
2.1.4    Emerging Standards.....	15
2.1.5    Research Trends .....	16
2.2    Mobile Agent Security .....	17
2.2.1    Multi-agent Issues .....	17
2.2.2    Threats and Requirements.....	18
2.2.3    Malicious Agents .....	20
2.2.4    Malicious Hosts .....	22
2.3    Chapter Summary .....	25
CHAPTER 3 MULTI-AGENT ARCHITECTURES for SECURITY.....	26
3.1    Chapter Overview.....	26
3.2    Mobile Agent Data Integrity using Multi-agent Architecture (MADIMA).....	26
3.2.1    Requirements for Data State Protection.....	27
3.2.2    Partial Result Protection Mechanisms.....	28
3.2.3    Describing the Problem .....	29
3.2.4    Architecture Overview .....	31
3.2.5    Related Security Issues.....	34
3.2.6    Fault Tolerance Issues .....	35
3.2.7    Performance Issues.....	35
3.2.8    MADIMA Summary.....	36
3.3    Hybrid Approaches for Secure Multi-Agent Computations.....	36
3.3.1    SMC Integration with Mobile Agency.....	37
3.3.2    Invitation and Response Protocol.....	38
3.3.3    Multi-Agent Trusted Execution .....	42
3.3.4    Hybrid SMC Approach Summary .....	43

3.4	Chapter Summary .....	44
CHAPTER 4 MOBILE AGENT TRUST FRAMEWORK .....		45
4.1	Chapter Overview.....	45
4.2	Security Requirements and Mechanisms for Mobile Agents .....	46
4.3	Trust Framework .....	48
4.3.1	Defining Principals.....	48
4.3.2	Defining Trust Relationships.....	51
4.4	Trust Algorithm .....	52
4.4.1	Trust Decisions .....	52
4.4.2	Trust Acquisition .....	56
4.4.3	The Role of Trusted Hosts.....	56
4.5	Application Security Models .....	57
4.5.1	The Military Model .....	58
4.5.2	The Trade Model .....	60
4.5.3	The Neutral Services Model .....	61
4.6	Chapter Summary .....	63
CHAPTER 5 PROGRAM ENCRYPTION.....		64
5.1	Chapter Overview.....	64
5.2	Motivating the Question.....	65
5.2.1	Mobile Agents.....	66
5.2.2	Sensor Nets.....	66
5.2.3	Geo-Positional Location Information .....	67
5.2.4	Financial Transactions.....	67
5.2.5	Protecting Embedded Keys in Programs.....	67
5.2.6	Transforming Private Key Crypto-Systems into Public Key Systems.....	67
5.3	Defining Program Encryption .....	67
5.3.1	Measuring Cryptographic Security .....	68
5.3.2	Heuristic Views of Obfuscation.....	68
5.3.3	Theoretical Views of Obfuscation .....	69
5.3.4	Why We Need a Different Security Model .....	72
5.3.5	Program Understanding .....	73
5.3.6	Program Context .....	75
5.3.7	Protecting Program Intent using Program Encryption .....	75
5.4	Creating Perfect Black Box Obfuscation .....	77
5.4.1	One-Way Functions and Black Box Obfuscation.....	77
5.4.2	Implementing Perfect Black Box Obfuscation .....	79
5.5	Defining the Random Program Security Model .....	81
5.5.1	Random Data and Random Programs/Circuits .....	81
5.5.2	Random Program Oracles and White Box Obfuscation .....	84
5.5.3	Proving Random Programs Exist .....	85
5.5.4	Proving Random Circuits Exist.....	90
5.6	Creating White Box Protection Based on Randomization .....	99
5.6.1	Comparing Data and Program Encryption.....	99
5.6.2	Integrating Black and White Box Protection .....	101
5.6.3	Intent Protection with White Box Randomizing Transformations.....	102
5.6.4	Distinguishing Random Selections of $T_{DOW_A}$ from $P_R$ .....	104
5.6.5	Obfuscators that Randomly Select Circuits from $T_{DOW_A}$ .....	104
5.7	Creating Perfect White Box Protection.....	106
5.7.1	Existence of 2-TM Obfuscators for Bounded Input-Size Programs.....	107

5.7.2	Provably Secure Obfuscators for Bounded Input-Size Programs .....	108
5.7.3	Perfect Obfuscation in a Private Key Setting.....	109
5.7.4	Protecting Bounded Input-Size Programs with Easily Learnable Input .	113
5.8	Implementation Work .....	114
5.9	Chapter Summary .....	115
CHAPTER 6 CONCLUSIONS.....		117
APPENDIX A COMPREHENSIVE SURVEY OF MOBILE AGENT SECURITY .....		119
A.1	Evaluating Agent Security Mechanisms .....	119
A.2	General Host Protection .....	121
A.2.1	Sandboxing (SBFI) .....	121
A.2.2	Safe Code Interpretation .....	122
A.2.3	Code Signatures.....	123
A.2.4	State Appraisal .....	124
A.2.5	Proof Carrying Code.....	125
A.2.6	Path Histories .....	126
A.2.7	Policy Management.....	127
A.3	General Agent Protection .....	129
A.3.1	Contractual Agreements/Reputation .....	130
A.3.2	Detection Objects .....	130
A.3.3	Oblivious Hashing.....	130
A.3.4	Protective Assertions.....	131
A.3.5	Execution Tracing.....	131
A.3.6	Holographic Proofs.....	134
A.3.7	State Transition Verification.....	135
A.3.8	Reference States.....	136
A.3.9	Environmental Key Generation.....	137
A.3.10	Secure Routing.....	138
A.3.11	Multi-Hop Trust Model .....	139
A.3.12	Returning Home .....	139
A.3.13	Phoning Home .....	140
A.3.14	Trusted Nodes/Third Parties.....	140
A.3.15	Server Replication/Fault Tolerance .....	143
A.3.16	Agent Replication/Mutual Itinerary Recording .....	144
A.3.17	Route/Itinerary Protection.....	145
A.3.18	Sliding Encryption.....	149
A.3.19	Trusted/Tamper-resistant hardware .....	150
A.3.20	Function Hiding with Encrypted Functions .....	151
A.3.21	Function Hiding with Coding Theory.....	153
A.3.22	Undetachable signatures.....	154
A.3.23	Policy Management Architectures .....	156
A.4	Agent Data Protection .....	156
A.4.1	Digital Signature Protocol .....	158
A.4.2	One-Time Symmetric Keys.....	158
A.4.3	Bitmapped XOR Protection .....	159
A.4.4	Targeted State.....	160
A.4.5	Append-Only Containers .....	161
A.4.6	Multi-Hops Integrity.....	161
A.4.7	Partial Result Authentication Codes .....	162
A.4.8	Hash Chaining .....	165

A.4.9	Set Hash Codes .....	169
A.4.10	One Time Key Generation (OKGS) .....	171
A.4.11	Configurable Protection .....	172
A.4.12	Modified Set Authentication Code .....	173
A.4.13	Chained IP Protocol.....	174
A.4.14	EIGamal Encryption.....	174
A.4.15	Protocol Evaluation.....	175
A.5	Secure Multi-Party Computations.....	176
A.5.1	Evaluation Techniques and Primitives.....	176
A.5.2	Single Round Computations and Agent Integration .....	177
A.5.3	Non-Interactive SMC Approaches .....	179
A.5.4	Multi-Round SMC Approaches .....	182
A.6	Multi-Agent Architectures .....	184
A.7	Trust Infrastructures .....	186
A.7.1	Trust Management in Distributed Environments .....	186
A.7.2	Trust and Mobile Agents.....	187
APPENDIX B	TRUST MODEL ELABORATION .....	189
B.1	Trust Scenario with Mobility and Agency .....	189
B.2	Modelling Agent Applications .....	193
APPENDIX C	ENUMERATING COMBINATIONAL CIRCUITS .....	197
APPENDIX D	PROGRAM ENCRYPTION ARCHITECTURE .....	200
APPENDIX E	ISCAS CIRCUIT DEFINITIONS AND BENCH FORMAT .....	210
APPENDIX F	BOOLEAN EXPRESSION DIAGRAMS.....	213
REFERENCES.....		218
BIOGRAPHICAL SKETCH.....		234

## LIST OF FIGURES

Figure 1: Malicious Host Threat Classification .....	3
Figure 2: Blind Disruption versus Effective Tampering .....	4
Figure 3: Considerations in Agent Mobility.....	7
Figure 4: Software Agent Space .....	8
Figure 5: Mobile Agent Lifecycle .....	9
Figure 6: Host Agent Middleware.....	11
Figure 7: Agent Itinerary Description.....	11
Figure 8: FIPA Agent Communication Methods [Poslad et al. 2002] .....	13
Figure 9: Agent Interaction Model .....	14
Figure 10: Agent Kernel and Identity Definition with Security Attributes .....	14
Figure 11: FIPA Agent Management Specification .....	16
Figure 12: Taxonomy for Defining Mobile Agent Security.....	18
Figure 13: Summarizing Mobile Agent System (MAS) Attacks .....	19
Figure 14: Malicious Agent Threats .....	20
Figure 15: Architecture for Host Security .....	22
Figure 16: Malicious Host Threats .....	23
Figure 17: Itinerary Specification In Mobile Agents.....	24
Figure 18: Agent Protection Overview .....	26
Figure 19: Partial Result Data Expression .....	27
Figure 20: Stateful/Stateless Agent Interactions and Data Integrity.....	30
Figure 21: Data Integrity Attacks.....	30
Figure 22: Launching Task Agent ( $t$ ) and Single-Hop Computation Agent ( $a$ ).....	32
Figure 23: Using Replicated Computation Agents ( $a,b$ ).....	32
Figure 24: Data Collection Agents ( $a,b,c,d$ ) .....	33
Figure 25: Data Collection Modes.....	33
Figure 26: MADIMA Security Configurations .....	35
Figure 27: Secure Multi-Agent Computations .....	36
Figure 28: Agent Task Realized as Secure Multi-Party Computation .....	37
Figure 29: User Task $F$ Implemented as Secure Multi-Agent Computation.....	39
Figure 30: Initialization and Invitation Phases.....	39
Figure 31: Response and Recovery Phases.....	40
Figure 32: Invitation Agents Sending Host Requests.....	40
Figure 33: Response Agents and Execution Environments.....	41
Figure 34: Fully-Trusted Middle Man with Multi-Agent/Multi-Round SMC .....	42
Figure 35: Phase 3 with Semi-Trusted Middlemen Execution Sites .....	43
Figure 36: Phase 4 Migration to Originating Host .....	43
Figure 37: Defining the Agent .....	49
Figure 38: Defining Executing/Dispatching/Trusted Hosts.....	50
Figure 39: Three Entities Affecting Trust in Mobile Agent Environments.....	50
Figure 40: Principals and Entities Associated with an Agent .....	51
Figure 41: Simplifying Trust Assumptions in Mobile Agent Application .....	52
Figure 42: Trust Decisions for Mobile Agent Security .....	53
Figure 43: Trust/Security Decisions on Agent Dispatch.....	54
Figure 44: Trust/Security Decisions on Agent Migration .....	54
Figure 45: Trust Framework.....	55
Figure 46: Acquired Trust over Multiple Applications.....	56

Figure 47: Application Security Models.....	58
Figure 48: Military Model Initial Trust Relationships.....	59
Figure 49: Trade Model Initial Trust Relationships.....	61
Figure 50: Neutral Services Model Initial Trust Relationships.....	62
Figure 51: Results in Program Encryption .....	64
Figure 52: Application Example for Program Encryption .....	66
Figure 53: Adversarial Program Intent Detection .....	76
Figure 54: Black box Obfuscation with Recovery Model.....	80
Figure 55: Black box Obfuscated Program .....	80
Figure 56: Considering Random Data and Random Circuits/Programs .....	82
Figure 57: Random Program Selection.....	83
Figure 58: Random Program Oracle .....	84
Figure 59: TBIA Machine Depiction .....	87
Figure 60: Simple One-Bit Architecture.....	89
Figure 61: The ISCAS-85 C <sub>17</sub> Benchmark Circuit in BENCH Notation .....	92
Figure 62: BED Definition of ISCAS-85 C <sub>17</sub> .....	93
Figure 63: Examples of Circuit Signatures.....	93
Figure 64: Exponential Blowup of Functional Representation .....	96
Figure 65: Comparing Data Ciphers with Program Obfuscation / Encryption.....	100
Figure 66: Example of Data Permutation and Substitution .....	101
Figure 67: White Box Protected Programs.....	102
Figure 68: Full Intent-Protected Program P'.....	103
Figure 69: Circuit Substitution and Permutation.....	105
Figure 70: Circuit Encryption in Context to HLL Code Protection .....	106
Figure 71: Bounded-Size Input DES .....	112
Figure 72: Fully Generalized Bounded Input-Size Program Obfuscation .....	112
Figure 73: Architecture for General Program Intent Protection (P.exe→ P'.exe).....	115
Figure 74: Sandboxing.....	122
Figure 75: Safe-TcL Padded Cell Concept .....	123
Figure 76: Simple Authentication .....	123
Figure 77: Integrity and Authentication .....	124
Figure 78: State Appraisal Technique.....	125
Figure 79: PCC Framework .....	126
Figure 80: Path Histories.....	127
Figure 81: Agent Policy Management.....	128
Figure 82: Model for Oblivious Hashing.....	131
Figure 83: Protective Assertion Framework.....	132
Figure 84: White and Black Code .....	133
Figure 85: Code Fragment and Trace.....	133
Figure 86: Execution Tracing Model .....	134
Figure 87: Extended Execution Tracing.....	134
Figure 88: Holographic Proof Checking .....	135
Figure 89: Replay Attacks.....	136
Figure 90: Reference State Mechanism.....	137
Figure 91: Environmental Key using Hash.....	138
Figure 92: Routing Based on Associations .....	139
Figure 93: Multi-Hop Trust Model.....	139
Figure 94: Agent Returning Home .....	140
Figure 95: Agent Phoning Home.....	141
Figure 96: Agent Protection Using TTP .....	142
Figure 97: Trust-Level Exchanging Protocol.....	142

Figure 98: Trusted Host Configurations .....	143
Figure 99: Simple Agent Pipeline .....	144
Figure 100: Replicated Agent Pipeline .....	144
Figure 101: Cooperating Agents .....	145
Figure 102: Agents w/ Variable Itineraries .....	146
Figure 103: Anonymous Itinerary .....	147
Figure 104: Chained IP Protocol .....	148
Figure 105: Public Key Data Encryption .....	148
Figure 106: Sliding Encryption .....	149
Figure 107: Trusted Hardware – Full/Partial .....	150
Figure 108: CED and CEF .....	152
Figure 109: Achieving Non-Interactive Privacy of Computation with CEF .....	152
Figure 110: A CEF Based On Coding Theory.....	153
Figure 111: Undetachable Signature Scheme .....	154
Figure 112: RSA-Based Undetachable Signature Scheme .....	155
Figure 113: Public/Private Data .....	157
Figure 114: One-Time Protection.....	159
Figure 115: Bitmap/XOR Data Protection .....	160
Figure 116: Targeted State Protocol.....	161
Figure 117: Append-Only Container .....	162
Figure 118: Multi-Hops Protocol.....	163
Figure 119: Simple MAC-based PRAC .....	163
Figure 120: PRAC with Hash-Based MAC.....	164
Figure 121: Publicly Verifiable PRACS .....	165
Figure 122: Encapsulated Offers .....	166
Figure 123: Protocol Interaction of PVCDS.....	167
Figure 124: Forward Privacy .....	168
Figure 125: Chained MAC Interaction.....	169
Figure 126: Publicly Verifiable Signature .....	169
Figure 127: Set Hashing Data Collection.....	170
Figure 128: One Time Key Generation Scheme .....	171
Figure 129: Key Generation Module .....	172
Figure 130: E-E-D Property.....	174
Figure 131: State Update Function .....	179
Figure 132: Host Output Function .....	179
Figure 133: ACCK Protocol w/ Generic Computation Service .....	180
Figure 134: Verifiable Distributed Oblivious Transfer Protocol .....	181
Figure 135: Oblivious Threshold Decryption Protocol.....	181
Figure 136: Agent Approaches to SMC .....	182
Figure 137: Multi-Agent Secure Computation .....	183
Figure 138: Trust Decisions and Principles in a Mobile Agent Setting.....	189
Figure 139: Agent, Application Owner, and Agent Developer Trust Relation .....	189
Figure 140: Host-Agent Trust Relations.....	190
Figure 141: Host and Entity Interactions in Agent Application .....	193
Figure 142: Agent Code, Unique Agent Instance and State Set.....	194
Figure 143: Same Agent Code used in Different Agent Instance .....	194
Figure 144: Same Agent Code, Different Agent Itinerary.....	195
Figure 145: Different Code Base, Same Agent Itinerary.....	195
Figure 146: Different Codebase, Different Agent Itinerary .....	195
Figure 147: Agent That Revisits Hosts in Itinerary.....	196
Figure 148: Multiple Agents with Same Codebase, Unique Itineraries .....	196

Figure 149: Multiple Agents with Same Codebase, Common Itinerary.....	196
Figure 150: MASCOT Generalized Randomization .....	200
Figure 151: ISCAS, BENCH, BED Deployment Diagram .....	201
Figure 152: Circuit Generation Library and Replacemen.....	201
Figure 153: MASCOT-EVALUATE Component.....	202
Figure 154: MASCOT-GENINPUT Component.....	203
Figure 155: MASCOT-PAD Component .....	203
Figure 156: MASCOT-DEPAD Component .....	204
Figure 157: MASCOT-RANDCIRCUIT Component.....	204
Figure 158: MASCOT-NANDCOVERT Component.....	205
Figure 159: MASCOT-RSABIN Component .....	205
Figure 160: MASCOT-3DESBIN Component .....	206
Figure 161: MASCOT-CONCAT Component.....	206
Figure 162: MASCOT-MERGE Component.....	207
Figure 163: MASCOT-EMBEDMULTI Component .....	207
Figure 164: MASCOT-CIRC2PROG Component .....	208
Figure 165: MASCOT-CIRCUITGEN Component .....	208
Figure 166: Circuit Library Specification (Binary).....	208
Figure 167: MASCOT-CANONICAL Component.....	209
Figure 168: BENCH Circuit Format.....	210
Figure 169: BENCH Grammar in Extended BNF Notation.....	212
Figure 170: BED C Program Created from C-17 BENCH Specification .....	214
Figure 171: Example Circuit P – Gate Level and Truth Table View .....	214
Figure 172: Example Circuit P in BED Notational Views .....	215
Figure 173: Example Circuit P (Canonical SOP) in BED Notational Views .....	216
Figure 174: ISCAS C <sub>432</sub> in BED Notational Views.....	217

## LIST OF TABLES

Table 1: Distributed Computing Paradigms .....	9
Table 2: Agent Middleware Systems.....	10
Table 3: Host Security Threat/Requirements Matrix .....	21
Table 4: Agent Security Threat/ Requirements Matrix .....	23
Table 5: Agent Data Privacy .....	28
Table 6: Agent Data Integrity Properties .....	28
Table 7: Agent/Host Security Requirements w/ Abbreviations.....	46
Table 8: Agent Security Requirements and Related Mechanisms.....	47
Table 9: Principals in Mobile Agent Systems (expressed in extended BNF notation) ....	48
Table 10: Trust Relationships .....	52
Table 11: Heuristic Obfuscation Metrics .....	68
Table 12: Heuristic Obfuscation Techniques .....	69
Table 13: Examples of Commercial Obfuscators.....	70
Table 14: Legal Program Considerations.....	86
Table 15: Ten Bit Instruction Set Architecture (TBIA) .....	86
Table 16: Modified TBIA with New Instruction .....	88
Table 17: Gate Definitions Under $\Omega_2$ .....	94
Table 18: Circuit Encoding for Family $C_{nms\Omega}$ .....	94
Table 19: Binary Size Representation for Circuit Encoding .....	95
Table 20: Program Encryption Results Overview .....	116
Table 21: Security Evaluation Criteria.....	120
Table 22: Host Protection Mechanisms .....	121
Table 23: Agent Protection Mechanisms .....	129
Table 24: Data Protection Mechanisms .....	156
Table 25: Abstract Data Protection Model .....	172
Table 26: Methods of single-round secure function evaluation.....	178
Table 27: Pertinent Results for Non-Interactive Secure Multi-party Computations.....	178
Table 28: Security Requirements with Associated Detection/Prevention Mechanisms	191
Table 29: Trust Decisions in Mobile Agent Applications .....	192
Table 30: High Level View of ISCAS-85 Circuit Library .....	211

## ABSTRACT

Mobile agents are an application design scheme for distributed systems that combine mobile code principles with software agents. Mobile computing emerged over the last decade with a vision for code that changes its execution location—moving from platform to platform in a heterogeneous network carrying an embodied, updatable state. Agents are software processes that act on a user’s behalf, perform particular functions autonomously, and interact with their environment to accomplish their goals. We consider in this thesis historical mobile agent security research while also gauging current trends.

Program mobility and autonomy are ultimate distributed computing expressions—programmers can view the network as a seamless canvas for application development. Disconnected host operations give a key advantage to mobile agents; however, researchers agree that protecting a stand-alone autonomous mobile agent with software-only approaches remains difficult. In this thesis, we produce several results that enhance mobile agent security and provide generalized code protection. We propose and define several novel techniques that protect mobile agents in ubiquitous environments and that solve practical problems in the program obfuscation field. We contribute to the field in the following ways:

**Generalized Black Box Program Protection.** We provide a novel technique for hiding a candidate program’s input/output relationships by using a data encryption padding technique. This method provides general program/circuit protection and relies on the semantic security strength found in common data encryption ciphers. Analyzing the black box relations for such protected programs cannot reproduce the original program’s input/output mapping.

**Generalized White Box Program Protection.** We semantically protect the white-box source code/gate structure information for a relevant program class defined by bounded input size. By using simple Boolean canonical circuit forms, we create an obfuscation technique that effectively hides all information regarding the source code or circuit gate structure.

**Embedded-Key Program Protection.** Leveraging our white-box results, we demonstrate how to embed an encryption key in programs that have small input size with measurable security. This technique gives foundations for solving the classic computer security problem regarding how transform any private-key cryptosystem into a public-key cryptosystem.

**Analyzing Mobile Code Protection Schemes for Code Privacy.** The Virtual Black Box (VBB) has been a theoretical foundation for understanding obfuscation strength for some time. We consider programmatic intent protection for mobile agents and pose a new model for obfuscated code security based on random programs.

**Tamperproofing Mobile Code.** We lay foundations for a new code protection methodology for mobile agents based on techniques used in the data encryption field. Specifically, we employ circuit encryption techniques that use combined sub-circuit permutation and substitution. As a result, we appeal to indistinguishability notions for circuits drawn uniformly from large sets and establish properties for obfuscators that provide intent protection based on randomization.

**Trust Framework for Mobile Agents.** Security tends to be Boolean and rigid in its application. Mobile agents in unknown and ubiquitous environments need a flexible security model that accounts for the unique challenges they face. We develop a novel framework to capture principles and trust relationships specific to the mobile agent paradigm. Our framework fills in the shortfall gap in current trust frameworks that attempt to deal with agents and mobility.

**Application Security Models.** Initial trust levels between mobile agent principals depend on the application security model. Application designers can provide initial trust conditions to characterize the mobile execution environment; we seed a mobile interaction trust database with these conditions. We define three different mobile agent settings that exhibit common security characteristics: the military model, the neutral services model, and the trade model. We apply these models in context to our trust framework and show their relevance in designing secure mobile agent applications.

**Multiple-Agent Protection Based on Secure Mobile Agent Computations.** Multiple agents provide greater capability for security in mobile contexts. We develop multiple agent architecture for mobility utilizing hybrid secure multi-party computation models, trusted high-speed threshold servers, and multiple agents.

**Multiple-Agent Scheme to Provide Data Encapsulation Protection.** We develop a novel approach to deal with colluding malicious hosts in context to data state integrity attacks. Our architecture utilizes three cooperating agent classes that prevent partial results disclosure by their interaction and by using public data bins.

**Comprehensive Mobile Agent Security History.** We provide a comprehensive mobile agent security history. We create and employ taxonomy for describing and understanding all security aspects that relate to mobile agents: mobility, threats, requirements, mechanisms, verification, evaluation metrics, and mechanisms.

# CHAPTER 1

## INTRODUCTION

Software agents are both a design paradigm and implementation level construct for designing distributed systems [1, 2]. Defined in artificial intelligence (AI) research, agents are software components that perform autonomous actions in order to accomplish predefined goals [3]. In the AI-based view, agents are software components that act on a user's behalf by carrying knowledge, reasoning over beliefs, representing user intentions, and communicating via some standard mechanism with other agents [4]. Agents are autonomous, goal-driven, adaptive, proactive, mobile, and social based upon the rules and actions provided by the designer [5].

*Mobile* agents [6] integrate software agents and the distributed programming paradigm known as *mobile code* [7, 8, 9]. Mobile programs that are *autonomous* and *reactive* to environmental changes (referred to henceforth as *mobile agents*) have found usefulness in domains such as information retrieval [10], e-commerce [11], network management [12, 13], digital image processing [14], tele-care assistance [15], grid computing [16], and peer-to-peer networking [17]. Real-world commercial applications based on mobile agents are not realizable until agent frameworks adequately address security concerns—no matter how useful or beneficial the mobile agent paradigm may be. Our research contributes stepping-stones to a secure mobile agent paradigm.

We begin by showing how to protect mobile agent data integrity when malicious hosts collude, provide architecture that guarantees host data privacy and execution integrity, and show how to reason about security choices when agents interact with unknown parties. We organize this thesis to reflect the corresponding research agenda. In Chapter 3 we present methods that positively counter integrity and execution integrity attacks by using multiple agent coalitions. In Chapter 4, we present a novel framework for exercising mobile trust management decisions.

Our major accomplishment addresses how to protect mobile agent code privacy and execution integrity in remote environments. In other words, we reduce a malicious party's actions from intentioned, smart code alterations to blind disruption. In Chapter 5, we give novel approaches to solving this historically tough problem, in the face of several theoretic impossibility results for obfuscation and software tamperproofing. We show first how to protect the black box properties of a general program with provable security and with reasonable efficiency; we define a new model by which to judge software obfuscation strength—according to known cryptographic security properties. We demonstrate in this thesis a methodology for producing randomized, executably encrypted circuits with provable white box security properties that are not subject to the traditional impossibility results. We design a methodology to provide perfect semantic white box program protection—with provable security properties—for a relevant class of programs. Finally, our approach gives one of the first known solutions for how to protect an embedded cryptographic key securely within a program and the first known public key encryption system that uses only symmetric key cryptographic computations.

### 1.1 The Problem Area

Current pioneers describe future generation computing with phrases such as “the network is the computer<sup>1</sup>” (network-aware programming) and terms such as “ubiquitous computing<sup>2</sup>” (the one-person-to-many-computer relationship common around the world today). In this brave new computing world, we must protect privacy and execution integrity for code located outside developer control or outside its native executing environment. Mobile computing emerged in the last decade and envisioned programs with an embodied, updatable state that move from platform to platform in a heterogeneous network environment [18].

---

<sup>1</sup> Trademark of Sun Microsystems, Inc

<sup>2</sup> Alan Key, Apple Computing (see <http://www.ubiq.com/hypertext/weiser/UbiHome.html>)

Migrating or “itinerant” agents act on a user’s behalf and perform a particular task autonomously. When they finish processing, agents return home or take further initiative once accomplishing their user-directed goal. Some researchers have tried to analyze why mobile agents have not achieved a widespread use outside academic circles [19] to this point. A consistent fear that agent systems cannot guarantee availability, integrity, and scalability while keeping the overhead manageable [20] tops the list for their adoption failure.

In order to be successful, practical mobile agent implementations must match system functionality with available security defenses and manage those protection aspects that are still unattainable. Managing agent security mechanisms requires an infrastructure to support mobility beyond those required for typical distributed systems—causing cumbersome implementation headaches. In some cases, agent applications will require a new security perspective based on non-Boolean trust. Deploying future autonomous/distributed applications running in possibly hostile computing environments will demand that security problems are addressed.

We gain benefit by defining and solving security problems in the mobile agent realm. Particularly, if mobile agent security issues can be solved, other security problems associated with distributed computing paradigms pale in comparison. In addition, agent security requirements readily overlap with needs in other traditional research areas such as software tamperproofing, virus protection, security integration with software engineering, policy enforcement, piracy prevention, digital rights management, and secure remote computing. Such ancillary result areas provide great impetus for our primary focus on furthering mobile agent security.

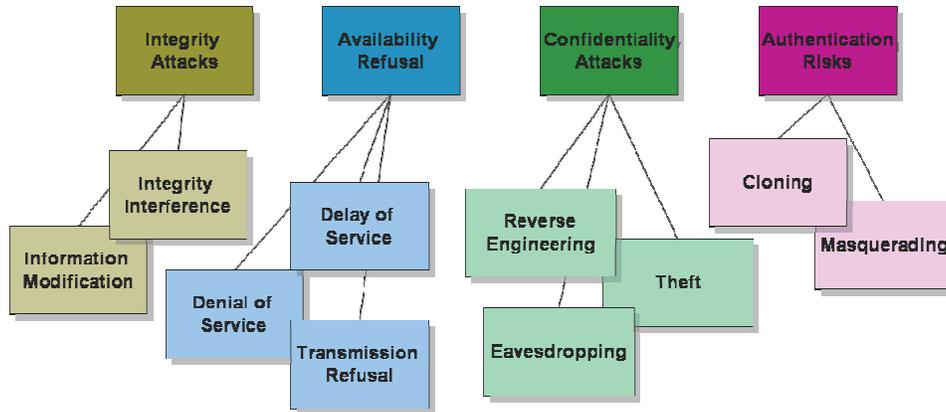
Agent security requirements are normally distilled into two categories [21, 22]: host protection and agent protection. We define our solution space starting with the hardest problem: protecting an agent from a malicious host. As the agent migrates, an intermediate hosts can alter its current state (and therefore its functionality) in unintended and malicious ways [23, 24]. Explicitly stated, how can agent integrity, privacy, availability, and authentication be protected when a remote host has full access to agent code and state being executed? Our results address this target problem.

Regarding agent protection, Bierman and Cloete [21] summarize four malicious host attack categories, illustrated in Figure 1: *integrity attacks*, *availability refusal*, *confidentiality attacks*, and *authentication risks*. Integrity and confidentiality alterations reveal and exploit the private information contained in the code and dynamic agent state. Together with authentication risks, these attacks represent attempts by a malicious party to gain unfair advantage without explicitly refusing agent execution. Hosts that perform strict service denial can starve the agent for resources, provide the agent wrong information, or destroy the agent without execution or migration. These host types represent the worst-case mobile agent risk.

Farmer and Guttman in [23] believe the questions regarding whether an interpreter will run an agent correctly or whether a server will run an agent to completion are impossible to answer. Even though other impossibility claims in [23] have been challenged, such as whether agent code and data can be kept private [25, 26, 27] and whether an agent can carry a key [28, 29, 30], we do not argue whether or not certain server actions can be prevented. Agents that execute on interpreters located at a remote host execute under the remote host’s power and control—not the originator’s control.

Single mobile agents are in many cases hard to protect against all possible malicious threats. In some cases, using multiple agents supports accountability or secure delegation without fixed hardware use. In other cases, introducing multiple agent classes enhances trusted hardware use. We investigate as a secondary goal possibilities for multi-agent architectures that increase mobile agent security and allow greater security requirements coverage.

Assuming that mobile agents and their intended execution environments are in different security domains or administrative control realms, no mechanisms exist to prevent absolute service denial attacks such as resource starvation or to guarantee honest host input. No current protection methods can reliably *prevent* strict denial of service in the mobile agent paradigm unless tamperproof hardware or secure co-processors [31, 32] completely control the remote execution environment. Even with tamperproof hardware, malicious parties can attack the remote host’s physical environment or indirectly influence an agent’s execution [22].



**Figure 1: Malicious Host Threat Classification**

We can *detect* service denial by employing certain security mechanisms such as trusted verification servers [33] and timed execution limits [34]. We refer to this alteration form alteration as **blind disruption** because, *at most*, an adversarial host can only circumvent correct program execution blindly. On the other hand, adversaries that use **effective tampering** execute remote agents with the intent to observe or alter the normal code execution in order to gain some benefit (integrity, confidentiality, and authentication attacks). Such threats include agent itinerary alteration, code replay attacks, changing execution pathways, and proprietary algorithm discovery. Figure 2 illustrates the distinction among tampering attacks.

Considering non-Byzantine faults that do not terminate a program, users prefer that programs terminate rather than continuing execution with erroneous or possibly corrupted results. Non-malicious terminating faults are at least detectable—even though we may not discern the failure cause or its remedy easily. For Byzantine program errors in mobile agents, we can detect strict service denial easier than *partial* denials where adversaries effectively alter mobile code for their own malicious intent without detection. We desire to prevent effective tampering as a research goal, not only for mobile agents but also for software in general. As a result, we present in this thesis results that preserve code privacy and execution integrity against attacks by malicious parties.

## 1.2 Research Objectives

Based on the malicious threat environment facing mobile applications, we pose and answer in this thesis three questions related to code security and agent protection:

*How can we enhance security with multiple agents?* (Chapter 3)

*How can we integrate trust into mobile agent security?* (Chapter 4)

*How can we tamperproof mobile agents and protect software in general?* (Chapter 5)

We provide relevant answers to these questions in both incremental results and significant contributions. We frame each research area according to these questions and lay out results from our investigation in the following manner.

### 1.2.1 Multi-Agent Architectures for Security

In order for mobile agents to have widespread acceptance, mobile applications must adequately address user-specific security concerns. Increased security requirements limit supportable system mobility, although distributed trust offers greater protection hope against malicious activity. Stand-alone mobile agents may require similar help in order to enforce security requirements. Our thesis results give methods to strengthen security in mobile agent paradigms by using multiple agent interactions. We pose and evaluate architectures that accomplish specific security requirements for mobile agents. Chapter 3 presents our research results for this objective.

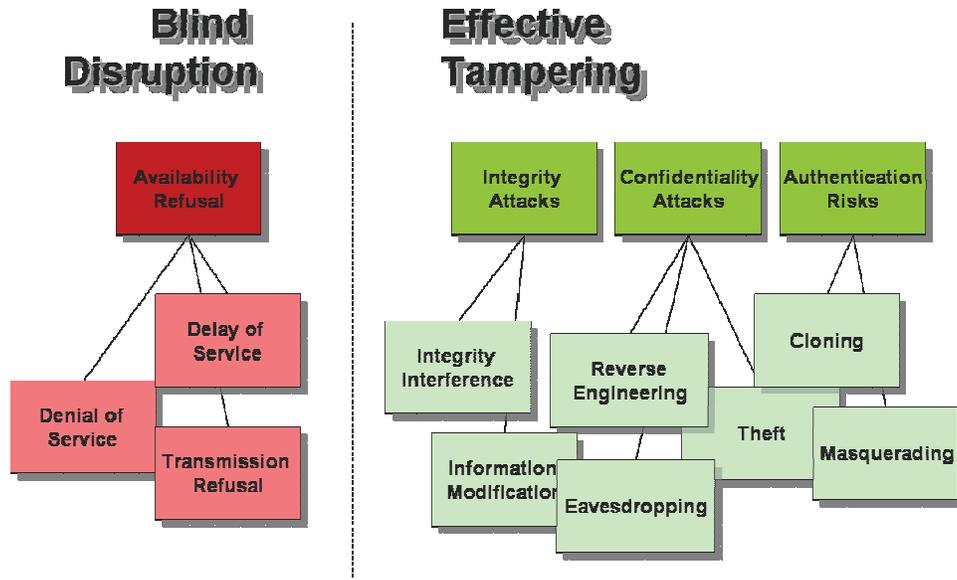


Figure 2: Blind Disruption versus Effective Tampering

### 1.2.2 Mobile Agent Trust Frameworks

Traditionally, mobile agent security has focused on protection mechanisms that keep malicious parties from altering the agent and on protection mechanisms that keep malicious agents from harming other parties. Researchers have done little to bridge the gap between requirements, trust expression, and protection mechanisms at an application-centric level. When dealing with application development, trust properties clearly define security requirements. Trust formulation has been given considerable thought both in distributed networking applications [35, 36, 37, 38] and mobile agents [39, 40, 41, 42, 43]. Mobility as an application feature complicates security because a host receiving a mobile agent to execute must make distributed trust decisions with little or no prior knowledge. Likewise, agents acting on a user's behalf must evaluate trust relationships with hosts in possibly unknown environments. Applications based upon mobile agents must blend user security requirements with environmental trust expectations where agents execute.

Typically, execution platforms perform software authentication and integrity checking to manage trust in a networked environment. In a mobile computing environment, both remote hosts and mobile code may act maliciously. We develop a trust model for mobile agents with novel features: linking application security requirements with mechanisms based on trust, reasoning about trust properties for generic security mechanisms, and application models for initial trust among principals in a mobile agent setting. Chapter 4 details our research results for this objective.

### 1.2.3 Program Encryption

Providing protection against effective tampering attacks against mobile agents remains an open problem in computer science. Researchers continue to seek ways to *prevent* certain tampering attacks realizing that they can only *detect* strict service denials (blind disruption) a posteriori. Finding ways to reduce effective tampering to blind disruption remains an active interest area. In this thesis, we develop effective means for both black box and white box protection that guard the agent's programmatic intent. These techniques are fully general for programs with small input size, but do not apply to all program classes. We develop also a theoretical foundation for understanding code protection based on program recognition and random programs [44]. We present our research results for this objective in Chapter 5.

## 1.3 Conventions

We briefly describe the notational conventions used throughout this thesis.

### 1.3.1 Cryptographic Primitives and Protocols

Cryptographic techniques provide secure functionality for enforcing various requirements such as confidentiality, integrity, and authentication.

**Symmetric key encryption** employs a secret key to encrypt a message into ciphertext and the same key to decrypt the ciphertext into the original message. For our purposes,  $E(K,M)$  and  $E_K(M)$  indicates a symmetric encryption algorithm  $E$  which encrypts data string  $M \in \{0,1\}^*$  using the secret key  $K$  to produce ciphertext  $C \in \{0,1\}^*$ :  $C = E(K,M)$ . Two notable symmetric encryption algorithms include Data Encryption Standard (DES) and Advanced Encryption Standard (AES).

For clarification, we describe decryption under a symmetric key scheme by  $D(K,C)$  and  $D_K(C)$  indicates a symmetric encryption algorithm  $D$  which decrypts data string  $C \in \{0,1\}^*$  using the secret key  $K$  to reproduce the original plaintext  $M \in \{0,1\}^*$ :  $M = D(K,C)$ . Under symmetric key cryptography, the message sender and receiver must agree on the secret key beforehand.

**Asymmetric key encryption** involves the using two different keys: one public ( $K$ ) and one private ( $K^{-1}$ ). In order for Alice to send Bob a message, Alice encrypts her message  $M$  with Bob's public key ( $K_B$ ). On receipt, Bob uses his private key ( $K_B^{-1}$ ) to decrypt the message. Signatures are an authentication technique associated with asymmetric encryption where a message originator encrypts a message with their private key ( $K^{-1}$ ). The other principal, upon receiving the message, can verify the sender's identity using the sender's public key ( $K$ ). In order for Alice to verify Bob as the sender of a message  $M$ , Bob signs (encrypts) his message  $M$  with his private key ( $K_B^{-1}$ ). On receipt, Alice uses Bob's public key ( $K_B$ ) to verify the message.

**Symmetric Block Ciphers.** A block cipher is a function  $E: \{0,1\}^k \times \{0,1\}^m \rightarrow \{0,1\}^m$  that takes a  $k$ -bit key and an  $m$ -bit (block length) plaintext input and returns an  $m$ -bit ciphertext string. The inverse function  $D: \{0,1\}^k \times \{0,1\}^m \rightarrow \{0,1\}^m$  takes the same  $k$ -bit key and an  $m$ -bit ciphertext string and returns an original  $m$ -bit plaintext string. We let  $E_K(M)$  denote the encryption of block message  $M \in \{0,1\}^m$  with a specific key  $K \in \{0,1\}^k$  and let  $D_K(C)$  denote the decryption (inverse encryption) of block message  $C \in \{0,1\}^m$  with the same key  $K \in \{0,1\}^k$ . We assume that any block cipher  $E$  of interest to us is a strongly pseudorandom function that is a permutation on  $\{0,1\}^m$ , as defined for example by Goldreich in his textbook [188].

**Message Protocols.** When a principal  $X$  sends message  $m_i$  to principal  $Y$ , we indicate this by  $X \xrightarrow{m_i} Y : F_1, F_2, F_3, \dots, F_n$ , where the message contents are the fields  $F_1, F_2, F_3, \dots, F_n$ .

### 1.3.2 Boolean Functions and Circuits

A Boolean function (also known as a gate) is a map  $f: \{0,1\}^n \rightarrow \{0,1\}$ . For  $n = 2$ ,  $f$  is a 2-input Boolean functions. A basis  $\Omega$  is a set of Boolean functions. We define a circuit over a basis as a directed acyclic graph (DAG) having either *nodes* corresponding to functions in  $\Omega$  being termed *gates* or having *nodes* with in-degree 0 being termed *inputs*. We distinguish one (or more) nodes as *outputs*. We compute the value at a node by applying the corresponding function to the values of the preceding nodes. We define the circuit *size* as the number of *gates*. We define the circuit *depth* as the length of the longest directed path from an input to an output. We say the basis  $\Omega$  is *complete* if and only if all  $f$  are computable by a circuit over  $\Omega$ . We define the *size* of the basis  $\Omega$  as the number of functions composing it and represent it using  $|\Omega|$ . We define  $B_n$  as the set of all Boolean functions with  $n$  inputs.

**Combinational Circuit:** We refer to standard Boolean circuits over  $\Omega = \{\text{AND,OR,NOT}\}$  and let  $C$  be a circuit with  $n$  inputs and  $m$  outputs. For  $x \in \{0,1\}^n$ ,  $C(x)$  denotes the result of applying  $C$  on input  $x$  and specify that  $C$  computes function  $f: \{0,1\}^n \rightarrow \{0,1\}^m$ . A combinational circuit (block, component) consists of logic gates that process  $n$  input signals  $x_{n-1}, \dots, x_0$  into  $m$  output signals  $y_{m-1}, \dots, y_0$  using a function  $y = f(x)$ , in such a way that output signals depend only on the current input signals.

**Truth Tables.** Assuming  $P: \{0,1\}^n \rightarrow \{0,1\}^m$ ,  $T(P)$  indicates the  $m \cdot 2^n$  size matrix of input/output pairs that represent the truth table (logical) relationship of  $P: \forall x, [x,y] = [x, P(x)]$ .

**Canonical Forms.** We represent the canonical form of a Boolean function  $f$  with  $n$  inputs as a sum of its products (minterms). Each product ( $\wedge$ ) has  $n$  terms and the summation has at most  $2^n$   $\vee$ -terms. We give the upper-bound for the canonical form of any Boolean formula as  $O(n2^n)$  gates.

$$f(x_1, x_2, \dots, x_n) = \bigvee_{f(a_1, a_2, \dots, a_n)=1} (x_1^{a_1} \wedge x_2^{a_2} \dots \wedge x_n^{a_n})$$

### 1.3.3 Turing Machines and Programs

Unless otherwise noted, Turing machines and circuits are identified by their normal descriptive representations as strings in  $\{0,1\}^*$ . TM stands for a Turing machine while PPT stands for a probabilistic polynomial-time Turing machine. Given algorithm  $ADV$ , algorithm  $M$ , and input string  $x$ , the notation  $ADV^M(x)$  indicates the output of algorithm  $ADV$  when executed on input  $x$  using oracle (black box) access to  $M$ . The black box oracle  $M$  can be a circuit or TM. When  $M$  is a circuit, algorithm  $ADV$  receives query responses  $M(x)$  from oracle  $M$  on input  $x$ ; when  $M$  is a TM, algorithm  $ADV$  can perform black box queries of the form  $(x, t)$  and receive  $M(x)$  if  $M$  halts within  $t$  steps on  $x$  or receive the halt symbol ( $\perp$ ) otherwise.

When algorithm  $ADV$  is a PPT,  $ADV(x; r)$  indicates the output of  $ADV$  when executed on input  $x$  using random tape  $r$ .  $ADV(x)$  is the distribution induced by choosing  $r$  uniformly from the distribution and executing  $ADV(x; r)$ . For a distribution  $D$ , we indicate with the notation  $x \xleftarrow{R} D$  a random variable  $x$  distributed according to  $D$ . For a set  $S$ , we indicate with the notation  $x \xleftarrow{R} S$  a random variable distributed uniformly over the all the elements of the set  $S$ . A function  $\alpha: \mathbb{N} \approx \mathbb{R}^+$  is negligible if, for any positive polynomial  $p$ , there exists  $N \in \mathbb{N}$  such that  $\alpha(n) < p(n)^{-1}$ , for any  $n > N$ .

We let  $P | E$  refer to the concatenation of program  $P$  with the program  $E$  such that  $(P|E)(x) = E(P(x))$ , for all  $x$ . Given a program  $P: \{0,1\}^* \rightarrow \{0,1\}^*$  and  $\forall x, y = P(x)$ , we let  $|x^P|$  represent the input size of  $P$  in bits and let  $|y^P|$  represent the output size of  $P$  in bits. Let  $P$  be defined as function  $P: \{0,1\}^n \rightarrow \{0,1\}^{|y^P|}$  and  $E: \{0,1\}^{|x^E|} \rightarrow \{0,1\}^m$ . We define the concatenation of  $P$  with  $E$  as  $P | E: \{0,1\}^n \rightarrow \{0,1\}^m$ .

## 1.4 Chapter Summary

We introduce in this chapter the field of mobile agent security and software protection. We give a review of several incremental results corresponding to our research and highlight the more significant results concerning program intent protection. We outline the remainder of the thesis as follows. Chapter 2 and Appendix A present a comprehensive literature review on mobile agent security, program protection, and trust frameworks that are applicable to our research. We present issues and results associated with our research objectives individually. Chapter 3 describes the security utility and design benefits for using multi-agent architectures and provides results in developing such architectures. Chapter 4 introduces a novel framework for integrating trust into mobile agent security decisions. Chapter 5 presents our research results for developing mobile code and software protection schemes that enhance code privacy and execution integrity. Chapter 6 concludes with a summary and discussion.

## CHAPTER 2

### MOBILE AGENT SECURITY

Our three-pronged approach to strengthening mobile agent security involves diverse computer science disciplines. In this section, we provide a literature review for appropriate areas related to our research and results. Appendix A provides a more comprehensive analysis for the interested reader. In Section 2.1, we first define mobile agents and their characteristics as a distributed computing paradigm. In Section 2.2, we define the requirements and threats associated with securing mobile agent systems.

#### 2.1 Mobile Agent Paradigms

Two worlds merge in the mobile agent paradigm: software agents and distributed computing. These worldviews have very different research goals, associated standards, and underlying assumptions. Mobile agent *frameworks* are the meeting point between theory and practice for mobile agents—providing a means for agent construction, migration, and execution in real-world applications. Figure 3 depicts this relationship and points us to considerations for mobile agent security.

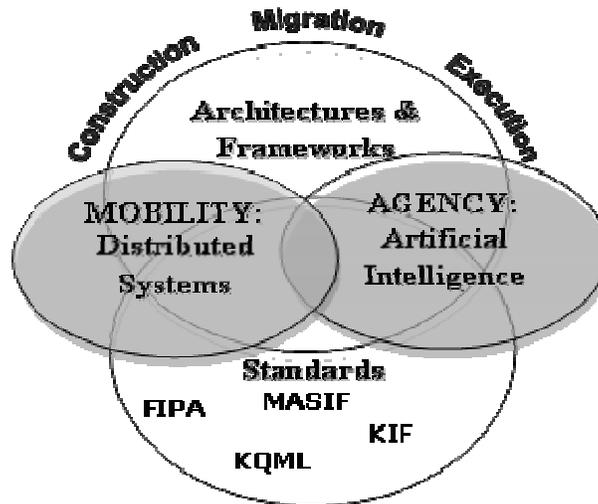


Figure 3: Considerations in Agent Mobility

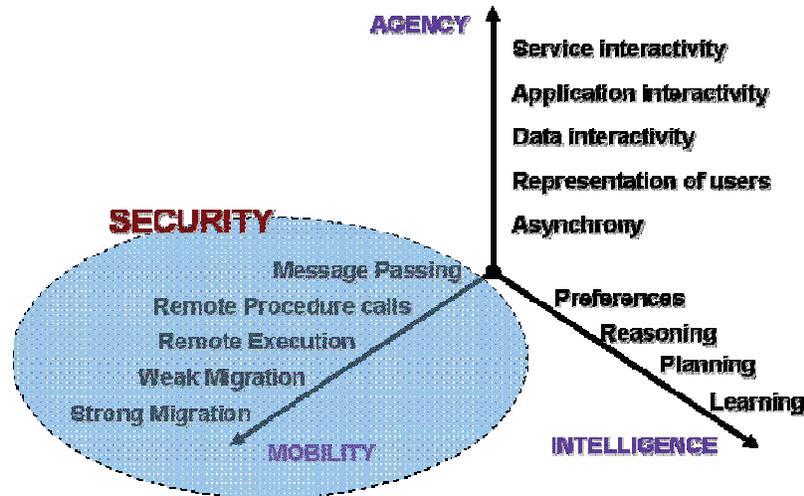
##### 2.1.1 Defining Agents

In applied artificial intelligence (AI), agents perform autonomous actions in order to meet user-preferred goals [3]. We describe an agent as automated software that assists a user and acts on their behalf. Agents perform tasks by carrying knowledge, reasoning over beliefs, representing user intentions, and communicating via some standard mechanism with other agents [4]. We describe agent behavior as autonomous, goal-driven, adaptive, proactive, mobile, and social based upon predefined rules and actions provided by their designer [5].

Researches like Odell [45] describe agents not in *human* terms but as active objects with private execution threads. Agent actions in the object-centric view arise from thread interactions, conditional statements, method invocations, object interactions, exception handling, and serializable persistence [46]. Agents, under any definition, are software processes that execute within some environment. Like any other software component, they communicate with both users and other processes via predefined protocols such as message passing.

Single agents (like those that help a user to better customize repetitive tasks) do not require collaboration with other agents. In multi-agent systems, we place agents into classes based on

labor divisions representing their functionality. Single or multiple agent instances from different agent classes work together and are deployed for some common system goal: networking, interface assistance, filtering, information fusion, brokering, transaction processing, monitoring, decision making, knowledge management, and many others [47]. Agent *infrastructures* or *middleware* provide ability for agents to interact, communication via agent communication language (ACL) protocols, and specify standard ontology in a *framework* [48].



**Figure 4: Software Agent Space**

Several researchers have made strides in defining intelligent agents and developing agent-oriented software engineering techniques; such results are established by Rao/Georgeff [49], Decker *et al.* [50], Bradshaw [2], Object Management Group [51], and Tomic/Agha [52]. Researchers consider contributions by Wooldridge and Jennings [3, 5, 53, 54] to be seminal. Classic multi-agent scenarios involve cooperating processes with incomplete or specialized capabilities working in a decentralized manner, usually with distributed information across asynchronous computations [55].

Multi-agent systems embody social, goal directed behavior and establish message-passing protocols used by agents that define a system. Gilbert *et al.* [56] describe autonomy and intelligence as orthogonal design spaces in considering software agents. Mobility, which is the ability to migrate to another location and perform a task, remains an *additional*, but non-essential agent feature. Figure 4, derived from Rothermel and Schwehm [57], depicts our interest in taxonomical definition from among these three spaces as the security aspects for agents that exhibit mobility.

### 2.1.2 Defining Mobility

In the distributed computing realm, mobile or itinerant agents are a natural extension to remote code execution [18]. Mobility removes the requirement that a process must remain confined to the host where it began execution. Mobile agents as a *distributed computing* paradigm manifest in four different expressions, illustrated in Table 1 [57]. Rothermel and Schwehm define each paradigm based on the program location (the know-how), the resources used by a program, and the execution environment (the processor) that a program runs on [7].

In a mobile agent scheme, the program moves to its resource location and executes on the local environment using the remote processor to update its internal state. Reduction in network loads and supporting asynchronous and disconnected operations are two benefits, among others, seen by researchers such as Lange and Oshima [58] and Kotz and Gray [59]. Researchers such as Chess *et al.* [18,60], Carzaniga *et al.* [9], Ghezzi and Vigna [7], Fugetta *et al.* [8], Riordan and Schneier [61], Bradshaw *et al.* [1], and Milojevic *et al.* [62] establish foundational premises for combining *code mobility* with *agency*.

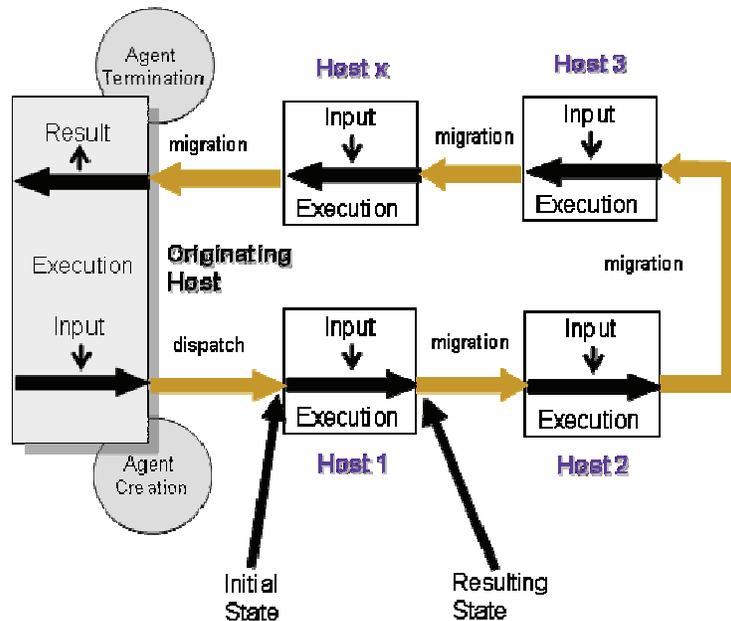
**Table 1: Distributed Computing Paradigms**

Computing Paradigm	Mobility Mechanism
Client-server Message passing	Transporting data
Remote Evaluation Code on Demand	Transporting code + data
Mobile Objects Weak Migration	Migrating code + data
Mobile Agent Strong Migration	Migrating code + data + state

**2.1.3 Mobile Agent Interactions**

A static program (code), a dynamically changing state (data), and a program thread (execution state) together compose a mobile agent. Agent construction, migration, and execution do not occur in a vacuum: applications must define security requirements for such interactions as well. Researchers use different models to represent agent and host interactions, with no agreed upon standard as to which representation generically captures a mobile agent system.

Figure 5 illustrates the mobile agent lifecycle used by Hohl [63] and shows an agent from creation to termination. An originator creates an agent with an initial state and dispatches it to the first host. Each subsequent host takes the previous agent state as a starting point and provides appropriate host input, updating the dynamic agent state appropriately. Input encompasses all data provided to the agent while on the remote host: communications from other agents on the same or different hosts, communications with the originating host, results from system calls, and results from service invocations. An agent ends execution on a particular host when it completes local processing and requests migration. The agent migrates back to the originating host and provides the task result to the application owner.



**Figure 5: Mobile Agent Lifecycle**

**Host Environments.** The agent platform or (remote) host must provide to agents an execution environment that we term *middleware*. The host operating system controls this environment and the middleware provides all necessary services to an agent. Middleware may provide primitive operations to agent programmers via services and other facilities that adhere to

a predefined standard. The agent middleware ensures correct itinerant or static code execution, provides runtime service access, and places constraints on native resources such as CPU, memory, local disk, or bandwidth [22].

Several commercial and academic middleware systems exist and we provide Table 2 for reference. Bellavista *et al.* [64], Schoeman and Cloete [65], Altmann *et al.* [66], Wong *et al.* [67], Fuggetta *et al.* [68], and Rothermal *et al.* [69] provide descriptive analysis for mobile agent architectures. Fritz Hohl's mobile agent list<sup>3</sup> provides current commercial and research based mobile agent middleware while Thorn [70] presents an early literature review on mobile code languages and platforms.

**Table 2: Agent Middleware Systems**

System	Languages	Developer
ADK	JAVA	Tryllian, Netherlands
Aglets	JAVA	IBM, Japan
Ajanta	JAVA	Univ. of Minnesota, USA
Ara	C/C++, TCL, JAVA	Univ. of Kaiserslautern, Germany
Concordia	JAVA	Mitsubishi, USA
CyberAgents	JAVA	FTP Software Inc.
D'Agents	AgentTCL	Dartmouth, USA
FIPA-OS	JAVA	Nortel, USA
ffMAIN	TCL, Perl, JAVA	Univ. of Frankfurt, Germany
JACK	JAVA	Agent Oriented Software Group, USA
JADE	JAVA	Telecom Italia Group, Italy
JATlite	JAVA	Stanford, USA
KAFKA	JAVA	Fujitsu, Japan
Knowbots	Python	CNRI, USA
MOA	JAVA	Open Group, UK
Mole	JAVA	Univ. of Stuttgart, Germany
MonJa	JAVA	Mitsubishi, Japan
NOMADS	Aroma JVM	Inst. for Human/Machine Cogn., USA
OAA	C, Java, VB	SRI International, USA
Plangent	JAVA	Toshiba, Japan
TACOMA	TCL, C, Python, Perl, Scheme	Cornell, USA / Tromso, Norway
sEmao	JAVA	Fraunhofer-Institut GD, Germany
SOMA	JAVA	Univ. of Bologna, Italy
Voyager	JAVA	Objectspace, USA
ZEUS	JAVA	BT Labs, UK

Figure 6 depicts the agent middleware providing facilities to receive a marshaled agent, instantiate an execution environment for the agent based on its current state, and allow the agent to run until its next migration. Agent middleware captures agent state and marshals it via a departure point defined as either a raw socket or a dedicated communication channel. Some mobile code systems force the programmer to transition state from one platform to another manually, though strongly mobile architectures do this implicitly for each migration. Middleware offers native services for visiting agents (service points in Figure 6) or allows direct access to underlying operating system resources based on a predefined security policy.

An agent interacts with a host environment in three ways that cause security concern. First, mobile agents *move* and change their execution location. Researchers distinguish the agent movement expression from distributed systems specification; agent security seeks to protect the agent *itinerary* specifically. Second, mobile agents may need to *talk* with other agents or with the originating host using some predefined protocol, whether agent specific or not. Agent middleware must protect communication availability, integrity, and secrecy. Third, agents interact with a host server in meaningful ways that change their state. The code, state, and thread interactions between agent and host are described in different ways and examples can be found in Vitek and Castigna [71], Serugendo *et al.* [72], Fugetta *et al.* [68], Hohl [63], Borselius and Mitchell [73], and Yee [74].

<sup>3</sup> Available <http://draco.cis.uoguelph.ca/link.html>, October 2005

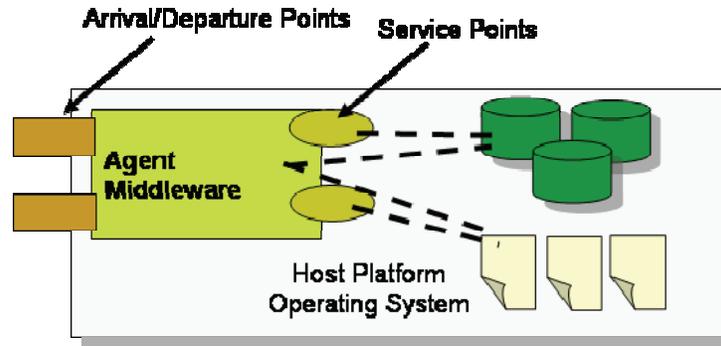


Figure 6: Host Agent Middleware

**Agent Mobility.** Mobile agents visit one or more hosts in a heterogeneous network and may or may not return to their originator. We term a migration from server to server a *hop*: multi-hop agents visiting more than one server and single-hop or one-hop agents visiting only one server. One-hop agents are similar to Java applets that download into browsers; Ghezzi / Vigna [7] and Rothermel / Schwehm [57] classify applets as *weak* mobility forms. Weak mobility also includes one-hop agents who transfer results back to the originating host via message passing. Ordille [75] refers to an agent that visits a host and migrates immediately back to its originator as a two-hop boomerang.

We refer to an agent's visited host set as an *itinerary* and depict them in Figure 7 as the migratory transitions between host platforms. We view route information as specialized agent data *state*, with some static *code* part dedicated toward updating and using it for migration, or as a special agent addendum used by the underlying middleware. The agent owner predetermines hosts in a *fixed itinerary* while an agent dynamically determines hosts as it migrates in *free-roaming itinerary*. In the latter case, an agent decides the next hop in the route either with help from the host environment or on its own using built-in communication mechanisms.

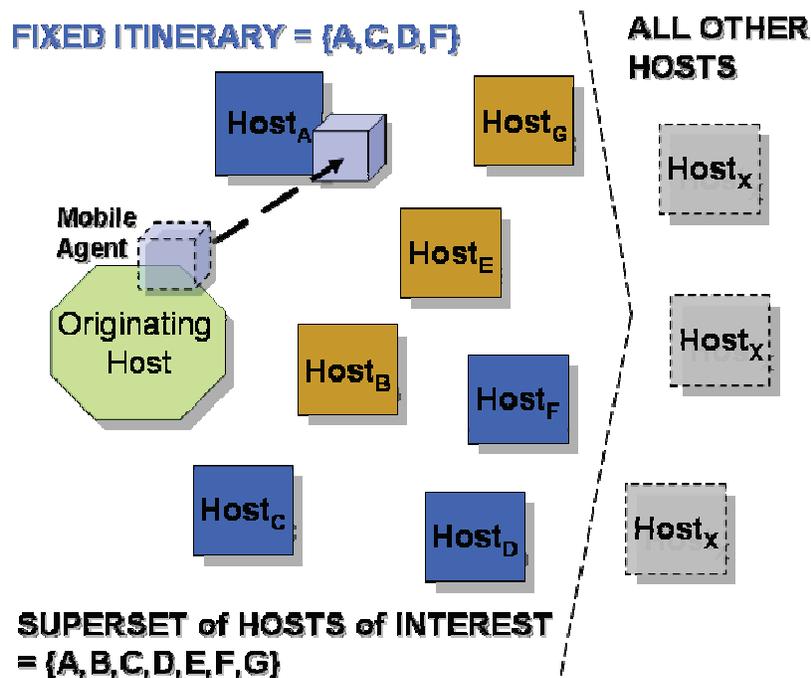


Figure 7: Agent Itinerary Description

An application owner may provide the agent a possible host superset to visit while also allowing the agent freedom to determine dynamically a subset to visit. Researchers like Ordille [75], Wilhelm and Staamann [76], Borrell *et al.* [77], Chen and Chang [78], and Knoll *et al.* [79] focus on describing and protecting agent itineraries. There are several possibilities for an agent's itinerary, described notionally from the agent set and host set seen in Figure 7. An agent with only the first hop specified represents the most difficult security requirement for integrity—described as the set  $\{A\}$  based on Figure 7. For an autonomous, free-roaming scenario, the agent adds new possibly unknown hosts to its migration list every time it migrates. A fixed unordered superset can also be known a priori as a bound on the agent's travel, represented by the unordered set  $\{A,B,C,D,E,F,G\}$  in Figure 7. This itinerary configuration allows dynamic free-roaming traversal through a host subset. Another mixed itinerary mode allows the originating host to embed an initial static itinerary within the agent while still allowing dynamic additions to the list as the agent migrates. The originating host can also specify the exact host subset an agent can visit and in what order—creating the most restrictive configuration. Figure 7 depicts this specification type as the ordered set  $\{A, C, D, F\}$ .

Some *security* mechanisms require a fixed itinerary and a known host superset, while others are more flexible and support free-roaming traversals. Researchers such as Aridore and Lange [80] and Tahara *et al.* [81] pose methods for using meta-level agent design patterns for specifying agent itinerary during software development phases; agent frameworks vary themselves in how they handle agent itinerary. Satoh [82] defines formal agent itinerary specifications for free-roaming scenario support.

Several formal models derived from distributed systems research are used to reason logically about mobility. We term these formal mobility expressions and their communicative relationships as process calculi or process algebras. Such calculi provide a strict notational expression for locations, resources, threads, networks, authorization, and programmatic execution used in describing mobility. Milner and his colleagues [83,84] created the  $\pi$ -calculus to model independent parallel processes that perform message-passing handshakes on specified channels.  $\pi$ -calculus has become a baseline algebra from which many variations have descended and other calculi are compared against. Serugendo *et al.* [85] compare the various formalisms used to describe mobile agents.

Polyadic  $\pi$ -calculus refers to architecture that models messages between multi-object processes and researchers typically extend the calculus for specific purposes. Extensions to  $\pi$ -calculus include support for asynchronous operations [86], support for mobility with distributed(D)- $\pi$  [87,88], modeling communication between processes with nomadic- $\pi$  calculus [89], and incorporating security primitives like cryptographic operations with Spi calculus [90]. Other algebras that define mobility (most deriving from  $\pi$ ) include the Join calculus [91], the mobile Ambient calculus [92] which describes cooperating mobile agent processes, the UNITY [93] and Mobile UNITY calculi [94, 95] which address specific mobile devices and disconnected wireless operations, and the Seal calculus [96] which models secure transactions over distributed networks like the Internet.

Formally expressing security and “good/bad” relationships are possible in algebraic models. The crypto-loc calculus [97], the SLam calculus [98], and the Spi calculus [90], for example, support using cryptographic or security primitives in process interactions. These expressive frameworks extend work in other modal logics and assign distributed processes security and authentication properties.

**Agent Communication.** Agents not only have the ability to change execution location but also have the ability to interact with other agents. Middleware may or may not provide specific facilities to support communication possibilities. Standards for multi-agent systems categorize agent communications as either intra-platform or inter-platform [99]. Rothermal and Schwem [57] subdivide them into four categories: agent-to-service interaction, mobile agent-to-mobile agent interaction, agent group communication [100], and user-to-agent interaction. All four classes carry with them security requirements to include privacy, integrity, and availability.

Poslad *et al.* [101] review the agent communication methods found in the Foundation for Intelligent Physical Agent (FIPA) Message Transport Specification [99]. Figure 8 details the

different ways an agent can communicate with another agent according to the specification. The first method involves agents sending messages to a local agent communication channel (ACC) via some proprietary interface. The service then sends the message to the remote ACC using a message transfer protocol. In the second method, an agent interfaces directly with the ACC on a remote host where another agent resides. Finally, an agent can send a message directly to another agent using a direct non-FIPA communication method.

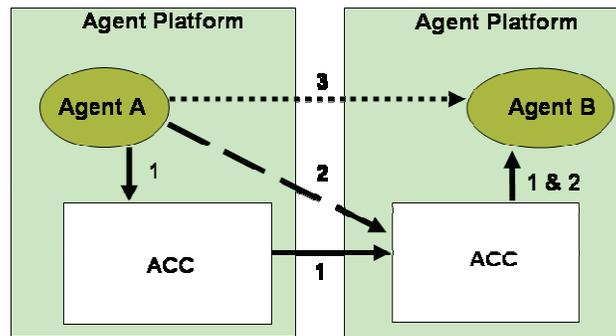


Figure 8: FIPA Agent Communication Methods [Poslad et al. 2002]

Chess *et al.* [18] define agent communication languages such as Knowledge Interchange Format (KIF), Knowledge Query Manipulation Language (KQML), and Ontolingua<sup>4</sup> for specifying agent-to-server and agent-to-agent communications. Thirunavukkarasu *et al.* [102] address basic security protection such as integrity, privacy, and authentication in KQML. Their protocol allows agents that have some or no cryptographic primitive support to negotiate security using performatives. Further, the protocol supports privacy, authentication, and non-repudiation, but does not address message replay.

Other research efforts for securing agent communications deal with adapting security into newer standards. Tan *et al.* [103] focus on FIPA inter-platform communication by creating security specifications for S/MIME content-type messages. Their architecture includes signed data, enveloped data, clear-signed data, and signed-enveloped data as content types for KQML messaging, all accomplished by manipulating message data itself. Mobile agent communications are slightly different, because the host and agent communicate something more than just *data*. Labrou *et al.* [104] summarize issues with integrating agent communication languages in mobile settings.

**Agent Resource Access.** Agents may simply borrow the remote host's processor; however, mobile architectures normally assume agents migrate to remote hosts to update their state by meaningful host interactions. Mobile agents accomplish work that usually requires an input from a particular host, whether the price for a particular good in an e-commerce setting or an information retrieval result. A mobile agent queries the host for this input, performs processing on the data, and embodies the result in its dynamic state. Security mechanisms ensure that an application owner or host can distinguish meaningful changes from malicious changes.

To reason about security requirements for host resource access and its effect on an agent (namely state transitions), researchers describe the agent-to-host interaction in different ways. Several authors [74, 105] describe the agent computation as a combined function pair: the state update function and the host output function. Yee [74] depicts this as a query sequence  $q_i$  issued by an agent against a host resource set  $R_i$  on host  $S_i$ . Figure 9 depicts this interaction and identifies  $y$  in relation to the dynamic agent state,  $R$  in relation to host services,  $q$  in relation to the agent's static code, and  $f()$  as an intermediate function that marks agent state transitions. Yee expresses the agent static code ( $q_i$ ) as query exchanges and assumes it does not change during server execution on the agent's behalf.

<sup>4</sup> Ontolingua Homepages, Available: <http://www.ksl.stanford.edu/software/ontolingua>

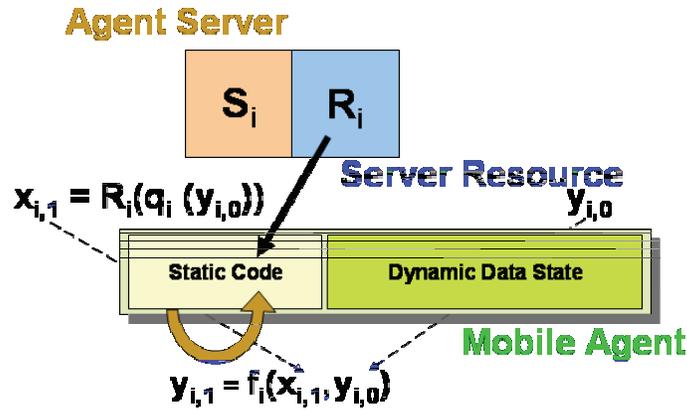


Figure 9: Agent Interaction Model

As Figure 9 illustrates, the query result ( $x_{i,1}$ ) depends on the query itself ( $q_i$ ) which incorporates the initial agent state on agent arrival ( $y_{i,0}$ ). This query represents the server executing the agent's static code, using the current agent state as a basis for interaction with the code, and then creating a new state version due to the interaction. After the host executes the first agent code segment, the host computes a new state ( $y_{i,1}$ ) and produces a new state used for the next agent query result ( $x_{i,2} = R_i(q_i(y_{i,1}))$ ). Each time the agent issues a query (executes another code segment), it obtains a new result  $x_{i,j}$  and creates a new state ( $y_{i,j}$ ). A given host can have several query exchanges with an agent in this model, representing the agent's full static code execution and the final agent state just prior to its next migration.

As another example, Hohl [106] uses the RASP abstract state machine model to define possible attacks on the mobile agent. Hohl cites deficiencies in other models (Turing machines, RAMS, and stack machines) that do not allow manipulating the state transition function—an essential facet in modeling mobile agent interactions. The RASP model allows agent code representation and state information manipulation accordingly. Other computational models for mobility express security properties broadly including data encapsulation, execution integrity, or execution privacy (see for example [25, 34, 107, 108, 109, 110, 111]).

**Agent Identity and Naming.** Specifying the agent identity has many important security ramifications. Leaving an agent unidentified makes the agent vulnerable to many possible attacks that include interleaving attacks from modified or mutated agents. Roth [108, 112] devises a simple identification scheme based on the asymmetric key signature function and a secure one-way hash function to help counter cut-and-paste attacks in various protocols (depicted here in Figure 10).

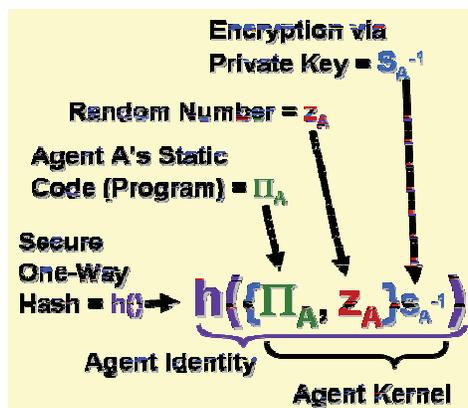


Figure 10: Agent Kernel and Identity Definition with Security Attributes

Roth defines the agent kernel and illustrates one method to identify an agent using security primitives. Agent migration represents a protocol exchange similar to those used in network protocol descriptions. Figure 10 depicts the agent kernel (the agent's static code or program embodiment) as a signed random number copy combined with the static code. The hashed signature provides a unique identity for a particular agent instance, assuming the application owner uses a sufficiently large random number not vulnerable to reuse.

To summarize, we define agents according to resource access, communication, mobility, and identity. Applications levy security requirements that relate directly to the mobile agent definition. Research communities express each interaction class differently. Agent middleware implementers, likewise, realize agent interactions differently using agent programming languages, communication languages, host services, and middleware architecture. The community requires a standardized definition for services and architecture to help simplify security specification—and we discuss possibility for such standards next.

#### 2.1.4 Emerging Standards

Multi-agent and mobile agent systems use different architectures and implementation schemes. They lack interoperability and security integration with each other and this situation stems largely in part from no clear emergent standards. Rothermal *et al.* [69] point out that the early attraction to mobile agents in both academia and industry resulted in non-standard mobile agent system implementations. Though standards for describing mobile agent interactions are few, even worse there are no agreed upon standards for describing mobile agent security [113].

Two standards have emerged for multi-agent technology: the Object Management Group's (OMG) Mobile Agent Systems Interoperability Facility (MASIF) and the specifications promulgated by FIPA, mentioned previously. The CORBA security model for OMG essentially absorbs the MASIF standards for security, even though CORBA has no strict mobility perspective. Poslad *et al.* [114, 115, 116, 117] present many security considerations for FIPA specification and they indicate FIPA appears to be the stronger candidate for adoption.

In 1998, FIPA published a preliminary specification for security [118]. The architectural specification for FIPA [119] came short for proposing security services but did provide for identification, access permissions, content integrity, and content privacy. Additionally, FIPA standards also address message transport protection, agent management protection, and security support protection. In March 2005, the IEEE Computer Society<sup>5</sup> became the umbrella organization for FIPA and formed a standards committee to support it.

Figure 11 illustrates that, the FIPA specification subdivides an agent platform into distinct services and operations [120]. The proposed FIPA agent management model organizes multi-agent systems, including those with mobility, by defining agents as autonomous processes that communicate via agent communication languages. Agents can look up other agents via a standard Directory Facilitator that serves as “middleman broker”. The Agent Management System controls agent operations on a particular host as a supervisor while the Message Transport Service [99] provides both intra- and inter-platform communication with other agents.

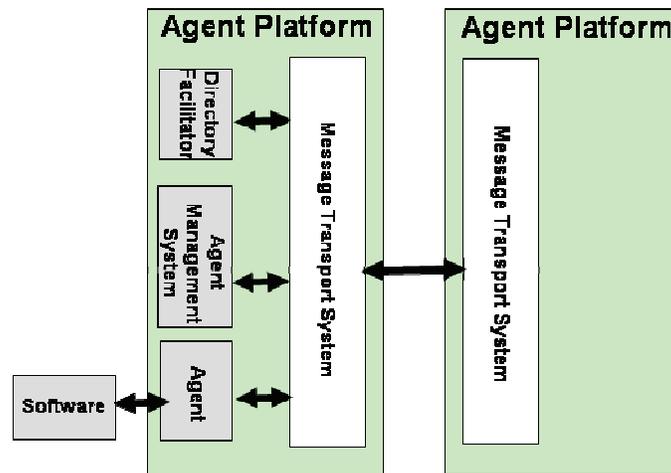
Roth *et al.* [121] posed a definition for mobile agent system interoperability which applies not only to underlying architectural assumptions but security protocols as well. Specifically, systems are interoperable if a mobile agent in one framework/system can migrate to a second, heterogeneously different, framework/system and communicate seamlessly with native agents. Their approach, instead pushing FIPA or MASIF standards downward, comprises a bottom up push using voluntary, practical interoperability features. Whether upward or downward in its implementation approach, standardization efforts in the future must include mobile agent security as an interoperability facet.

Pogg *et al.* [122] and Zhang *et al.* [123] address how to add support to FIPA for agent security. Even though FIPA envisions primarily *static* agents in multi-agent contexts that might be *mobile*, the specifications are still applicable in many ways to mobile agents and security requirements. As Poslad *et al.* [115] further note, no single or de facto standard for *mobile* agent security has

---

<sup>5</sup> IEEE Standards, Available: <http://www.computer.org/standards>

emerged despite feverish research efforts over the years. Mobile agents are more interesting from the security perspective than static multi-agent interactions: more opportunities for malicious activity springs from the way migrating agents interact with executing hosts. If researchers solve the security issues for mobile agents, they likewise solve most multi-agent issues as well.



**Figure 11: FIPA Agent Management Specification**

Richards [113] summarizes relationships between MASIF, OMG, FIPA, and other related standards by noting that realized security will vary greatly based on agent system differences and their specific implementations (even if standards are followed). Agent systems are not necessarily compatible with all the possible security mechanisms posed in the literature. Solutions are normally *middleware* specific and applying all defensive mechanisms to one system remains an impractical task. Researchers and implementers face a common dilemma: how do security mechanisms relate to requirements and standards, especially when applications drive specific security requirements?

### 2.1.5 Research Trends

Chess *et al.* [60] questioned claims touting mobile agent benefits, finding some claims still unproven at the time. In the same spirit, Rothermal *et al.* [69] identified at least three key elements in mobile agent environments early on that were missing: 1) security standards, 2) control structures, and 3) transactional support. Not surprisingly, Schoder and Eymann [124] noted some time after that the four top mobile agent technical challenges were security related: 1) a need for highly secure agent execution environments; 2) performance and functional limitations resulting from security; 3) virus scanning and epidemic control mechanisms; and 4) transmission efficiency, for example, a courier agent in contrast to a simple SMTP mail object. Initial mobile agent research seemed to solve many simple problems, while leaving many harder security related issues unanswered.

Milojicic's [62] interview with several researchers demonstrates a parochial mobility position in the agent community: some view mobility as a non-required but possibly beneficial agent system feature and others view mobility as a foundationally different paradigm to build applications around. Despite differences in the community over benefit, many researchers echoed Kotz and Gray's sentiment [59] that mobile agents were inevitable and "coming soon". Tschudin [125] termed mobility the "Zeitgeist" at the turn of the century. Designers envisioned more useful application features to end users in environments where limited bandwidth, disconnected operations, and mobile devices are prevalent.

Researchers have analyzed why mobile agents did not achieve widespread use outside academia despite efforts from the previous decade. Vigna [126] suggests ten reasons why agents have failed and his analysis shows nearly half are security related: 1) agents can be brainwashed; 2) they cannot keep secrets well; 3) they are difficult to authenticate and control; and 4) they have similar characteristics to worms. Samaras [19], in examining why the industry

has not embraced the technology as most in the research community expected, states that security problems remain the culprit. Johansen [127] rightly points out that mobile paradigm opponents focus on inherent architecture problems, namely the technical details that guarantee host and agent integrity. Finally, Roth [20] details obstacles to mobile agent adoption and cites the security (or lack thereof) as a common basis for fear. He observes that few sure mechanisms guarantee availability, integrity, and scalability in agent systems while keeping the overhead manageable.

Like Kotz and Gray [59], we expect that agents are indeed “coming” in the future and that practical implementations for mobile agent systems are incumbent upon pairing system features with available security defenses—while managing those protection aspects that are not attainable. The next section reviews security requirements and threats in the mobile agent environment and presents a framework for understanding both malicious host and malicious agent defense strategies.

## 2.2 Mobile Agent Security

As Matthew Henry<sup>6</sup> reminds us: “*corruptio optimi est pessima*—that which was originally the best becomes when corrupted the worst.” Mobile agents have potential for elegance and flexibility in distributed systems design; the risks posed by malicious mobile code or hosts currently overshadow any perceived benefit. As a result, a possibly good design abstraction such as mobile agents becomes the worst security nightmare. Both agent servers (referred to as *hosts*) and mobile agents can be maliciously altered in ways that go beyond normal software, network, and systems operations [24]. The literature contains many candidate solutions that mitigate possible attacks from malicious hosts, malicious agents, network adversaries, and underlying host platform compromise. Trying to grasp relevant research results and choose candidate security solutions for implementation in real-world applications remains a difficult task.

We assert that security engineering is vital to successful future mobile agent development efforts. Security must be incorporated from the ground up into any mobile agent system. Bellavista *et al.* [64] echo this sentiment:

*“The ultimate challenge is ... unifying security with system engineering... just as [mobile agent] system engineers analyze and select system features to answer functional requirements, security engineers must develop applicable security models and deploy those security measures that can make available different qualities of security service depending on specific security requirements of applications and on the most suitable trade-off between security and efficiency.”*

In order to address security requirements properly, we need to link mobile agent security mechanisms to the threats present in the mobile environment. Classifying various security mechanisms and their relative effectiveness for achieving security goals are closely associated. We discuss next the security requirements for multi-agent and mobile agents systems and analyze implementation specific security properties. As Figure 12 highlights, by nature, agent mobility creates a unique threat environment that includes possibly untrusted agents executing on possibly untrusted hosts. Multi-agent security and mobile agent security share similar issues and we discuss these next.

### 2.2.1 Multi-agent Issues

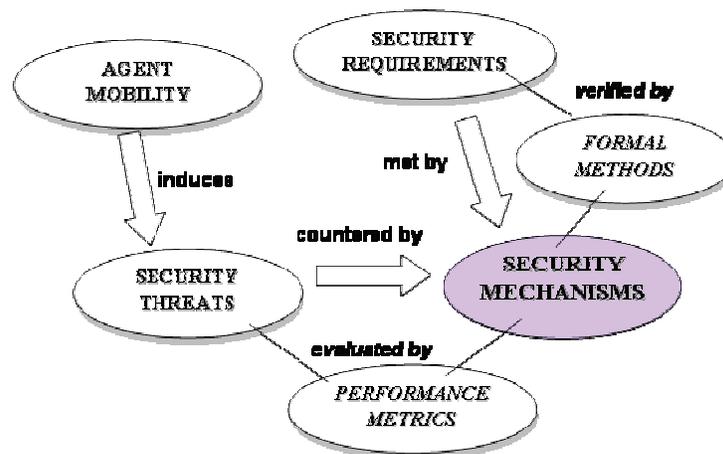
Researchers have treated security in *multi-agent systems* essentially as an afterthought since the field’s inception (unlike the *mobile agent* field where researchers placed precedence on security from the beginning). Wong and Sycara [128] consider malicious activity in multi-class systems and identify the need for several features: uniquely identifiable agents, agent key certification and revocation, agent services integrity, and secure communication channels.

---

<sup>6</sup> Matthew Henry’s Commentary: New Modern Edition, Electronic Database. Hendrickson Publishers, Inc., 1991.

Borselius [129,130] notes that agent security issues for communication are equivalent to normal requirements for confidentiality, integrity, authentication, availability, and non-repudiation in typical software applications.

Multi-agent security deals primarily with the protecting ACL messages passed between static agents deployed around the network and the security properties associated with host execution environments. Agents in multi-agent systems sense the environment and decide whether to raise the overall application security level. Security reduces to whether applications allow unencrypted transactions. Wells *et al.* [131] define such a security approach as an adaptive defense coordination architecture.



**Figure 12: Taxonomy for Defining Mobile Agent Security**

Bresciani *et al.* [132] develop multi-agent descriptions for complexity and security associations between actors and provide an analysis framework for whether critical security measures are met by the system design. Multi-agent systems often introduce security features via different agent classes that provide specific functionality (encryption, integrity checking, status checking, and so forth). The Tropos environment [133] analyzes whether certain agent classes are too taxed with security duties and assesses the consequences for their failure. Braynov and Jadliwala [134] propose a formal analysis technique that uses coordination graphs to detect malicious agent confederacies. This model assumes that cooperating malicious agents must cooperate to achieve their goals; coordination graphs reveal when malicious agents work together. The algorithm defines links, relationships, and cooperation between agent group members in order to establish (malicious) task correlation. The graphs help root out insiders by highlighting actions that an agent cannot perform alone given current resources. On the more practical side, Parks *et al.* [135] give initial results from a red-team approach that launches practical attacks against existing multi-agent architectures. Their attack categories consider the agent middleware and host operating system itself in addition to vulnerabilities at the communication level.

Researchers in multi-agent systems are beginning to address security and introduce countermeasures to threats in the software analysis and design phase. Multi-agent systems focus on vulnerabilities related to static messaging protocols; however, the priorities, threats, and requirements in the mobile environment demand greater attention and we discuss these next.

### 2.2.2 Threats and Requirements

We consider that developing good requirements for mobile agent security and matching those with existing security mechanisms will increase long-term mobile architecture success. As researchers like Rothermel *et al.* [69] state:

*“The vision of mobile agents as the key technology for future electronic commerce applications can only become reality if all security issues are well understood and the corresponding mechanisms are in place.”*

We agree that any future vision for using mobile agents must precisely define and address security issues—including current technology limitations, the problems that can be solvable, the problems that are impossible to solve, and the problems that remain left for research. Orso *et al.* [136] pose non-mobile specific security solutions that automatically analyze security requirements to determine the correct countermeasures set for an application. Requirements for agent security should be expressible in clear and traceable relationships to security mechanisms. We believe the catch-22 in mobile agent systems design is that “killer” mobile agent applications do not exist [127,137]. With so few real world mobile agent applications in existence, researchers encounter greater problems in devising candidate security solutions. The largely different variables and configuration possibilities that affect mobile agent security make solution / mechanism implementation even more difficult.

Security issues have been at the research forefront since mobile code emerged as a design paradigm for distributed systems. Chess [24] and Farmer *et al.* [23] describe several reasons why the *mobile agent* paradigm violates long held assumptions about the computational environment. In particular, we can normally attribute program actions to a person and believe the person intended the program’s actions by its execution—an assumption not true when programs migrate. Mobile agents are similar to malicious viruses because they can migrate from host to host without an ability to discern their intent before malicious actions have corrupted a system. Agent middleware has full and complete control over the mobile code content—exposing programs to unusual vulnerability.

Four threat categorizations in the mobile environment include: 1) attacks by malicious agents against hosts; 2) attacks by malicious hosts against agents; 3) attacks by agents against other agents; and 4) attacks by other entities against the host platform. Figure 13 depicts these various interactions using arrows and letters:

- Figure 13-(A), malicious agents can attack host platforms.
- Figure 13-(B), malicious hosts can attack agents.
- Figure 13-(C), malicious agents can attack other agents on their current platform.
- Figure 13-(D), adversaries can attack the underlying network transport mechanism.
- Figure 13-(E), agents can attack agents on other platforms.
- Figure 13-(F), agent platforms can attack other platforms.
- Figure 13-(G), intruders can launch assaults on the underlying host operating system.

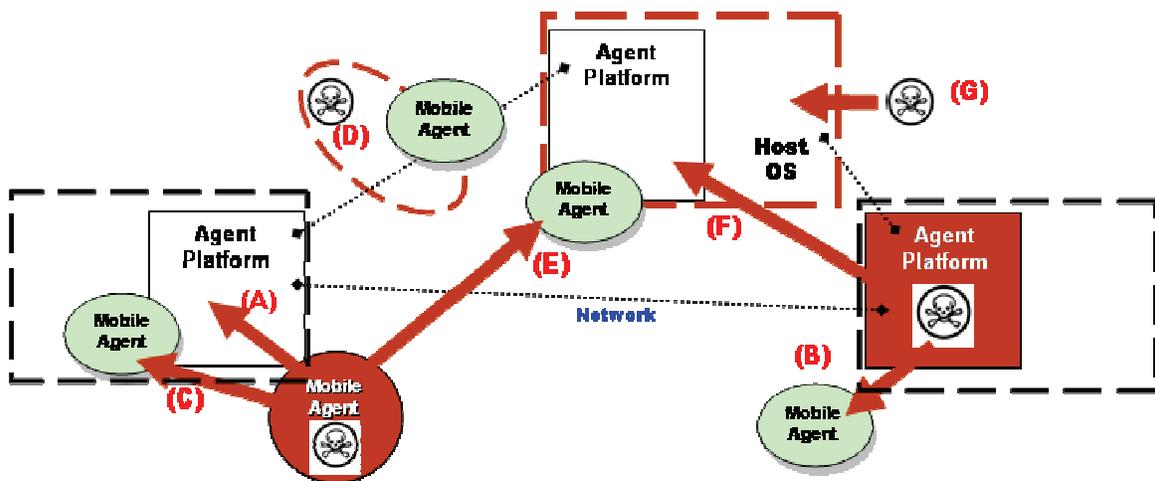


Figure 13: Summarizing Mobile Agent System (MAS) Attacks

### 2.2.3 Malicious Agents

An agent with hostile intent acting upon a server can exercise capabilities similar to a worm or virus. A remote host grants an agent trust in order to execute and use resources. Once given access, the agent executes like a normal process with rights to some or all host resources (CPU, disk, memory, bus, ALU, network channel, public host service, etc.). The agent can attempt to either gain unauthorized access to host resources or wrongly use the authorizations granted by the host [48]. Figure 14 summarizes the threats posed by hijacked agents.

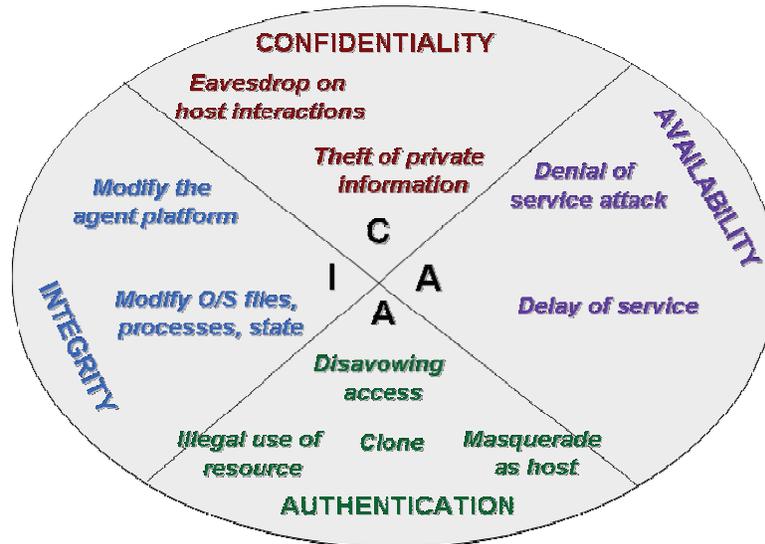


Figure 14: Malicious Agent Threats

Malicious agents can execute service denial attacks by unrestricted resource consumption on the host machine. They can also work to disrupt host operations and other agents by unreasonable requests or blocking certain services. Service denial threats also involve agents that issue worthless resource requests. Agents can also eavesdrop and monitor remote host resources, such as the communications channel or host ports, and try to gain unauthorized access to private host information. The host state, memory, and operating system resources may be all or partially available to the agent, depending on the middleware environment. While agents need freedom to communicate with other agents, migrate, and execute their programs, this freedom also exposes the underlying resources to risk.

An executing host demands agent accountability, especially when an agent maliciously commits or subverts transactions and denies involvement subsequently. Changes to the agent state or code can cause an agent to become malicious in nature itself. Agents may also be programmed to act politely up to a point—and then may abuse the privileges they are given. It remains a difficult task for the host to examine code intent, whether mobile or not, and to evaluate whether the agent possesses a legal execution state. As such, a receiving host needs a mechanism to monitor integrity changes to the agent state and code. Table 3 summarizes the threat and requirements correlation matrix for host security.

Tschudin [125] recognizes three essential needs concerning the mobile agent host: 1) *authenticating* the mobile agent; 2) *authorizing* the agent to use host resources; and 3) *allocating* resources to the agent for execution. These categories make useful analytic tools to formulate host requirements for possibly malicious agent interaction in future applications. When we consider authentication, the central question becomes whether we can verifiably identify a principal. The answer remains critical in mobile agent environments—normal trust relationships may be broken when an agent migrates along a multi-hop itinerary. Wilhelm *et al.* [138] express agent trust with four different levels, ranging from blind trust, trust based on a good reputation,

trust based on control and punishment, and trust based on policy enforcement. Policy enforcement represents the only strong trust establishment mechanism that relies on technology.

**Table 3: Host Security Threat/Requirements Matrix**

<b>Threat (Mobile agents can...)</b>	<b>Requirement (Hosts should...)</b>
<b>consume host resources unfairly (CPU, disk, memory, DOS)</b>	- monitor agent resource consumption - prevent illegal agent cloning
<b>delay responses to host to cause delays of service to other agents</b>	- monitor agent resource consumption - implement fine grained resource control based on policy
<b>masquerade as another user or agent</b>	- authenticate static agent code - establish agent identity or owner identity
<b>perform illegal operations on other processes, agents, or files</b>	- provide fault tolerant environments separate from normal processes - implement fine grained resource control based on policy
<b>have state corrupted due to traversal through the network</b>	- appraise dynamic agent data state integrity - implement trust-based agent authorization policies
<b>carry program code designed or altered for malicious intent</b>	- authenticate static agent code - appraise static agent code safety properties
<b>work together with other agents in joint colluding attacks</b>	- participate in inter-host data sharing
<b>deny execution results or activity</b>	- trace agent activity in non-repudiatable ways
<b>eavesdrop on agent or host communications (ports, channels, etc.)</b>	- implement fine grained resource control based on policy - secure intra-host agent communication channels
<b>steal information illegally from host or other agents</b>	- authorize agent to read or write only certain data - provide encryption services for visiting agents and local host resources
<b>be intercepted in route to a receiving host</b>	- secure inter-host communication

Similarly, Swarup [139] describes three trust appraisal levels required for incoming agents: *authentication, code appraisal, and state appraisal*. Based on these requirements, hosts should provide a safe execution environment (for agents) that limits access to resources and provides authentication and appraisal mechanisms for arriving agent code and state. Static program checkers and cryptographic primitives that support authentication and integrity provide methods for code appraisal. Hosts require a verification mechanism to perform state appraisal; they must discern runtime program safety and may examine trace logs for such purposes. Hosts decide the resource allocation level to grant an agent via code and state appraisal characterization and assign an appropriate authorization level. Reiser and Vogt [140] propose a conceptual architecture for host security that provides several hosts services and layered security services. Figure 15 illustrates the security features pipeline embodied in their approach.

We describe existing mechanisms for host security in Section A.2. Hosts determine agent integrity and assess safety by code or state appraisal mechanisms. Hosts allocate resources by combining access control and resource constraints based on the agent's authorization level. Ideally, an agent should not be able to bypass the execution environment. Jansen and Karygiannis [22] describe a reference monitor as a tamperproof service that mediates underlying resources. Among various reference monitor properties, several apply to host security for mobile agents. Hosts require some mechanism to isolate agents from operating system processes and from other agent processes (the sandbox seen in Figure 15).

Access control mechanisms guard computational resources and middleware providers must provide them to support mobile agent interactions. Reference monitors support agent-to-host or agent-to-agent information exchange as a basic service and may require cryptographic primitive services from the host if the agent does not provide native encryption. Hosts typically establish the agent identity, application owner identity, or itinerary host identity using cryptographic means. Finally, hosts need auditing capabilities for security-related environmental aspects—resource usage, file or process access, communication channel usage, and host operating system health. Malicious hosts can corrupt agents and use them against friendly hosts—therefore many agent

protection mechanisms (Section A.3) apply equally to host protection (Section A.2). We consider threats for malicious hosts and requirements for agent security next.

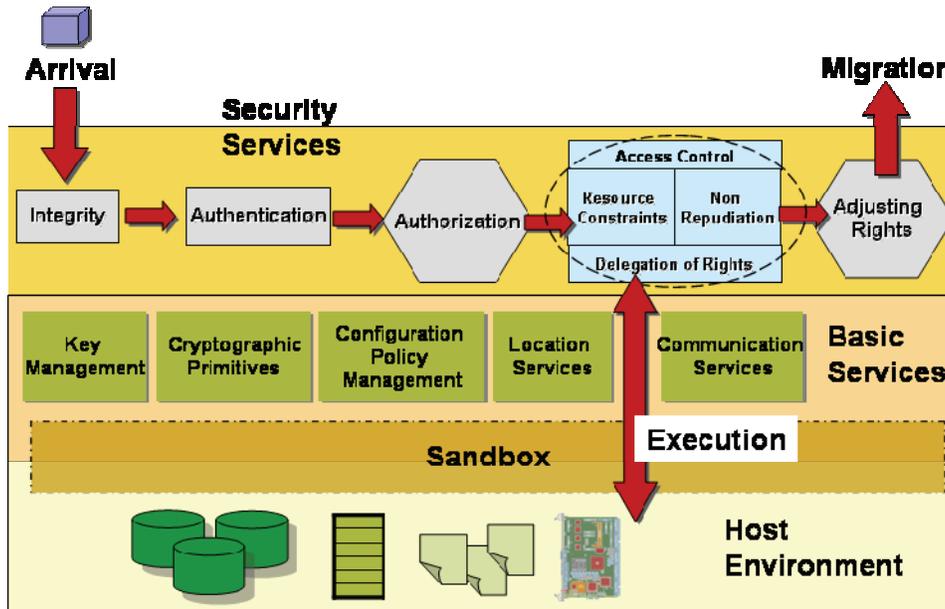


Figure 15: Architecture for Host Security

#### 2.2.4 Malicious Hosts

Host attacks are similar to agent attacks in many ways, but their problems remain the hardest to solve (host *platforms* could be mobile themselves but their resources are considered static for the agent's local environment). As background, we discuss Hohl's model [106] describing possible attacks in the malicious host environment. Bierman and Cloete [21] also classify host threats and detail appropriate agent protection mechanisms against the malicious host by security category. Five major concerns describe malicious host capabilities. Figure 16 organizes these based on categorization by Cubillos and Guidi-Polanco [48] and Table 4 summarizes the host attack threats as expressed by Hohl's model [106] with appropriate requirements for security implementation.

(1) **Inspection:** The host agent platform has the ability to inspect or observe an agent's static and dynamic part. Agents may also communicate with other parties during their traversal—thus requiring a means to establish secure channels without malicious observation. In some mobile agent contexts, the ability to see other host computational results can give a host an unfair advantage—as in the case e-commerce bidding applications. The host platform can also see every instruction executed by the agent. If the owner desires to hide a particular algorithm, a malicious host may reverse engineer the agent code if the code does not securely hide the computation [32]. In sum, the remote host has complete control over agent execution lacking other protection means [34]. We desire that hosts have the ability to execute code on a user's behalf without gaining any knowledge regarding what that code accomplishes. This ability remains a formidable challenge for malicious host protection and we provide solutions to counter such attacks in this thesis.

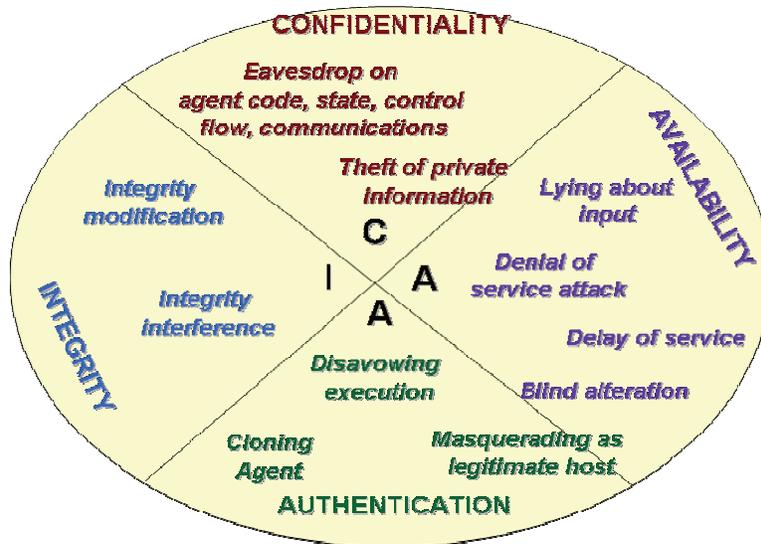
(2) **Modification:** Host platforms can modify the static code (possibly introducing a malicious agent) or modify previous host data results. Malicious parties can change code control flow to subvert or change the computational result [34]. Malicious attacks can include reading and writing data elements, program lines, state values, memory contents, and language expressions. An adversary may also have the ability to override the agent code interpretive

environment and alter intended execution results. The remote host may also change communications to influence an agent unfairly.

**Table 4: Agent Security Threat/ Requirements Matrix**

Threat (Hosts can ...)	Requirement (Agents should have...)
<b>read and modify the data state information</b>	- tracing mechanisms to audit host execution steps - checking mechanisms for legal/correct execution state - evaluation mechanisms to verify state/code consistency - mechanisms to copy state for later verification
<b>read and modify static code</b>	- masking services to hide algorithm functionality - detection mechanism to determine code modification
<b>read and modify the agent state by manipulating host</b>	- tracing mechanisms to audit host execution steps - auditing of host user inputs with non repudiation
<b>modify runtime environment</b>	- verification services to guarantee interpreter integrity
<b>control results of system calls</b>	- deadlock and livelock detection mechanisms
<b>read and modify agent communication channels</b>	- encryption capabilities for private communications

(3) **Denial of Service:** The host environment places the agent at its mercy: it can simply remove the agent from its planned migration and create a virtual “black hole” (see [141]). The host can also append any arbitrary computational result to an agent’s state, ignoring the original agent’s mission. Likewise, since data services represent a key part in the agent’s execution life cycle, the server can deny mobile agents access to data sources or lie about their input [137]. Few *detection* mechanisms exist to address denial of service (DoS), and even fewer *prevent* DoS in the host environment. We consider most security mechanisms successful if they reduce adversary attacks to blind disruption (DoS).



**Figure 16: Malicious Host Threats**

(4) **Replay:** A malicious host can perform black box manipulation by providing an agent with arbitrary data to observe its outputs and possibly discern its intentions [34]. An adversary can execute an agent repeatedly using different inputs each time by replaying code. Wilhem *et al.* [32] note this experimentation type represents an indirect attack all agents are subject too, whether or not the application owner enforces privacy via cryptographic operations.

(5) **Masquerading:** When masquerading, adversaries steal an agent's identity and launch subsequent impersonation attacks. Because the host can also fool an agent with wrong system call results, the host can trick the agent into believing it has arrived home so that the agent executes code that reveals confidential data.

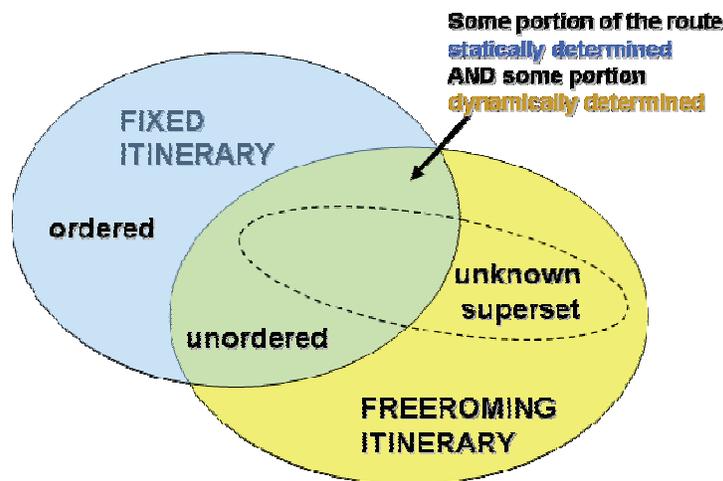
Middleware providers must provide protection for agent code, itinerary, and data state from these various threats. We consider the itinerary as a data state component, but, for security purposes, a middleware environment must protect the unique aspects of an itinerary that differ from the general agent data state. We address requirements for agent data protection further in Chapter 3 and provide a comprehensive review of existing data protection techniques in Appendix A.4.

**Protecting Agent Code.** The agent possesses static code (unless a mobile agent uses self-modifying code) and each code piece has an associated authentication and integrity property (signatures). Appraisal mechanisms attempt to prevent maliciously altered code execution on the remote host based on static agent code evaluation. This evaluation may provably determine an agent's safety level and might involve satisfaction by an agent executor that an agent does not violate certain policies. Agent and host protection are mutually dependent: malicious hosts can alter an agent and the next host in the multi-hop agent path must be able to discern such changes. The same mechanisms that authenticate a mobile agent to a remote host are normally the same mechanisms that guarantee agent code integrity during its lifecycle.

We express agent code trustworthiness as three requirements:

- (1) *Authenticating the code's owner/developer and the code's identity*
- (2) *Integrity verification that code received matches the code transmitted by the owner*
- (3) *Probabilistic proofs that code meets some predefined security policy*

**Protecting Agent Itinerary.** Agents can travel either on a free-roaming or fixed itinerary. Some mobile code systems only require single-hop, weakly mobile programs. When the agent has a static itinerary, malicious activity includes forcing an agent to skip certain host platforms or redirecting the agent to unspecified hosts [76]. In the dynamic setting, an agent can obtain new hosts to visit while it migrates, thus exposing the itinerary to random alterations and deletions without an ability to know alteration has occurred. Figure 17 shows the design space for the agent, including the case where some agent itinerary portion remains fixed and some remains dynamically determined.



**Figure 17: Itinerary Specification In Mobile Agents**

Agents lose trust with each new migration in a multi-hop setting without security mechanisms in place to protect the itinerary. Multiple colluding hosts can share itinerary information and therefore complicate protection. Itinerary protection involves using outside parties if necessary to

ensure a malicious host does not alter entries, delete entries, or add entries to their benefit. When agents must provide their own protection, they can use honest parties or the dispatching host to detect prior modifications.

## **2.3 Chapter Summary**

Mobile agents as a research field have an extensive history that crosses several related disciplines. We divide our quest to strengthen mobile agent security into three major result areas: reducing effective tampering to blind disruption via program encryption, integrating trust into the security decision process for mobile agents, and finding practical multiple agents applications to enhance security.

To introduce mobile agent security in this chapter, Section 2.1 briefly reviews code mobility paradigms and Section 2.2 introduces requirements related to mobile agent security. We refer the reader to a more comprehensive mobile agent security review in Appendix A including descriptions. We frame our results against multiple different host/agent protection mechanisms (Appendix A.2 and Appendix A.3). For our Chapter 3 results, Appendix A.4 details the literature related to data encapsulation techniques, Appendix A.5 reviews literature for secure multi-party computation, and Appendix A.6 provides background on multi-agent security. We give further background material in Appendix A.7 for our Chapter 4 results relating to trust frameworks. Chapter 3, 4, and 5 present results related to the objectives we pose in Section 1.2: how can we enhance security with multiple agents, how can we integrate trust, and how can tamperproof mobile agents. We turn our attention first to how multiple agents enhance security in mobile contexts.

## CHAPTER 3

### MULTI-AGENT ARCHITECTURES FOR SECURITY

This chapter contains material from two published works—one describing a novel technique for partial result protection based on cooperating multiple agents [142] and the other appearing in *Lecture Notes in Computer Science* which describes a hybrid approach for integrating secure multi-party computation with multiple mobile agents [143].

#### 3.1 Chapter Overview

As Figure 18 expresses, two aspects compose agent protection: protecting the agent's static executable code from disclosure or alteration and protecting the agent's dynamic state as it incrementally changes during execution. "Strength in numbers" can produce positive results for security to deter malicious parties from altering the agent's data result. We present two architectures in this chapter based on multiple agent interactions, which enforce specific requirements: enforcing strong data integrity a posteriori (Section 3.2) and guaranteeing host data privacy/state integrity (Section 3.3). We refer the reader to a more comprehensive background review on intermediate data state protection (discussed in Appendix A.4), group cryptographic techniques (discussed in A.5), and multi-agent paradigms (discussed in Appendix A.6).

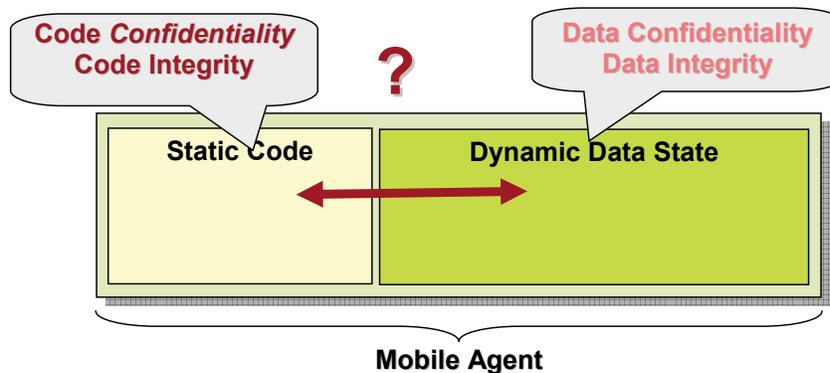


Figure 18: Agent Protection Overview

#### 3.2 Mobile Agent Data Integrity using Multi-agent Architecture (MADIMA)

We focus first on protecting the intermediate data results an agent gathers as it migrates through a network and introduce mobile agent data integrity using multi-agent architecture (MADIMA). Agent data protection keeps the agent data state safe from observation (*confidentiality*) or keeps it safe from alteration (*integrity*) by malicious hosts. *Integrity* violations are typically only *detectable* after the agent returns to its origination, when it reaches an honest host in the itinerary, or when it stores partial results with a trusted third party. MADIMA *prevents* integrity violations (without data aggregation) and *detects* violations (with data aggregation) by using multiple cooperating agents to accomplish user tasks.

We liken agent state integrity to transmitting a computation results collection back to an originating host without deletion, truncation, or alteration of individual results. Current solutions for integrity attacks detect malicious activity a posteriori. Such attacks require re-executing the agent and assume the application owner can successfully discover modifications even when multiple malicious hosts are present. Existing mechanisms cannot detect certain attacks involving colluding malicious hosts. We devise a solution to this problem by transferring partial results via cooperating mobile agents and thus prevent alteration completely, even when cooperating malicious hosts are present.

### 3.2.1 Requirements for Data State Protection

Roth [109] summarizes the requirements for agent data state protection. First, agents may have information that needs to remain private until the agent migrates to a trusted host. Second, agents carry partial result computations that require protection. The agent owner should be able to attribute a given partial result to the host that created it. Both Yee [31] and Karjoth *et al.* [144] give solutions for protecting free-roaming agents and pose schemes for expressing agent data protection mechanisms. Maggi and Sisto [107] formalize data privacy characteristics and attributes in their work. An agent's execution can be described by the set of hosts  $I, \{i_1, i_2, \dots, i_k\}$ , and the associated set of data states  $D, \{d_1, d_2, \dots, d_k\}$ , that represents the incremental change in agent state as it visits each host and performs its task. Given an originating host  $i_0$ , we describe the agent's path as the ordered set  $\{i_0, i_1, i_2, \dots, i_k, i_0\}$ . Figure 19 illustrates this representational scheme.

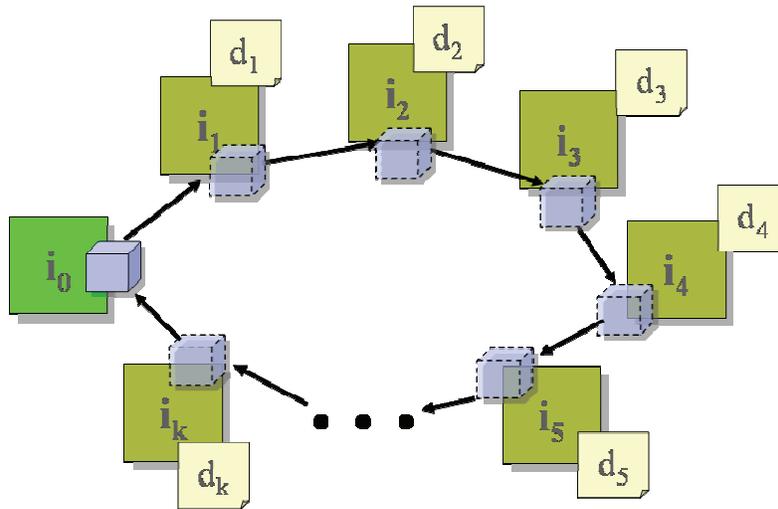


Figure 19: Partial Result Data Expression

In certain application settings, an agent may visit the same host more than once before returning to the originator, and so the set  $D$  could represent an ordered or unordered data results set. When an agent arrives back at its originating host, with task accomplished, the data results set  $D', \{d'_1, d'_2, \dots, d'_k\}$ , represents the incremental changes in agent state as it migrates around the network. The sets  $D$  and  $D'$  should be equivalent if no malicious hosts were present. Using this descriptive method, we define four attacks against the collected agent data integrity:

- (1) **Truncation:** An adversary initializes an agent's state back to a state from some previous host visit, essentially erasing intermediate results between two or more colluding malicious hosts. We state this as an attacker deleting all offers after the offer of host  $j$  and refer to this as "truncation at  $j$ ".
- (2) **Cancellation:** An adversary deletes a data item from the set  $D$ .
- (3) **Insertion:** An adversary inserts a data item into the set  $D$ .
- (4) **Substitution:** An adversary cancels a data item in  $D$  then immediately inserts another in its place, essentially replacing another host's data result. We term a series of phony offers at some host  $j$  as "growing a fake stem at  $j$ ".

Maggi and Sisto [107] and Karjoth *et al.* [144] provide several attributes that describe agent data privacy, summarized in Table 5, and desired data integrity properties, summarized in Table 6. As we discuss our approach to multi-agent data integrity, we reference these properties in relation to candidate protection mechanisms. Hosts detect data integrity attacks (data insertion or results deletion) *after* an agent has visited a malicious host and an application owner must rely on

appropriate detection mechanisms to be in place. Data confidentiality, however, must be proactive in the sense that it prevents revealing sensitive information to a host by using cryptographic primitives. Ideally, a mobile agent security scheme should provide data and origin confidentiality, data non-reputability, and strong data integrity.

**Table 5: Agent Data Privacy**

Term	Definition
<b>Data Confidentiality</b>	Any data element, $d'$ , should only be readable by the originating host $i_0$ .
<b>Origin Confidentiality</b>	(Forward Privacy) The identity of the host $i'$ that contributed data result $d'$ should only be determined by the originating host $i_0$ .
<b>Data Authenticity</b>	The originating host $i_0$ can determine the identity of the host $i'$ that added the data element $d'$ .
<b>Data Non-Repudiability</b>	(Stronger form of Data Authenticity) The originating host $i_0$ can prove the identity of the host $i'$ that added the data element $d'$ .

**Table 6: Agent Data Integrity Properties**

Property	Definition
<b>Strong Data Integrity</b>	After receiving the agent back, the originating host $i_0$ can detect any insertion or any cancellation: $D \neq D'$
<b>Weak Data Forward Integrity</b>	After receiving the agent back, the originating host $i_0$ can detect any cancellation: $D \not\subseteq D'$
<b>Trusted Data Integrity</b>	After receiving the agent back, the originating host $i_0$ can detect any cancellation from a set trusted hosts, $I_t: \{d_i \mid i_i \in I_t\} \not\subseteq D'$
<b>Strong Data Forward Integrity</b>	After receiving the agent back, the originating host $i_0$ can detect any substitution: $d_i' \neq d_i$
<b>Strong Data Truncation Resilience</b>	After receiving the agent back, the originating host $i_0$ can detect any truncation
<b>Data Truncation Resilience</b>	After receiving the agent back, the originating host $i_0$ can detect some truncations
<b>Insertion Resilience</b>	After receiving the agent back, the originating host $i_0$ can detect any insertion
<b>Publicly Verifiable Forward Integrity</b>	Any intermediate server, $i'$ , can verify the computation result of the computation state

### 3.2.2 Partial Result Protection Mechanisms

Historically, several protection mechanisms use multiple agents to transfer partial results for safekeeping during agent migration. Roth [280] proposes that an agent transfers commitments to another cooperating agent that verifies and stores the information gathered (see Appendix A.3.16). Dispatching hosts send agents to disjoint executing host sets and in turn send each other commitments via a host-provided secure communications channel. Roth's approach provides non-repudiation and requires a malicious host to corrupt other hosts that are on the co-operating agent's future itinerary.

Chained encapsulated results, partial result authentication codes, per-server digital signatures, append-only containers, and sliding encryption provide various intermediate result protection levels (see Appendix A.4). These mechanisms use digital signatures, encryption, and hash functions in different chained relationships to provide detection and verification services. With these techniques, the originating host or an honest host in the agent path can identify when

previous servers have inserted, truncated, or changed information from previous intermediate results carried by the agent. Loureiro *et al.* propose a subsequent protocol in [311] that allows a host to update its previous offer or bid. Roth proves that several protocols suffer from replay and oracle attacks because they do not dynamically bind the agent data state to its static code [108].

When malicious hosts collude, several protocols remain weak in detecting cooperating hosts that share secrets or send information to change intermediate host results. Specifically, we define truncations as integrity attacks where a malicious host resets the agent data state to a previous state (computed at a previous host). Using dynamically determined itineraries, existing mechanisms cannot detect truncation attacks.

Vijil and Iyer in [283] augment the append-only container with a means to detect mutual collusion and actually identify which hosts performed the tampering. Protocols may not be able to detect truncations at all. Maggi and Sisto in [107] provide a formal definition to describe protocol interactions in several different data protection mechanisms. In particular, they observe that protocols need to implement truncation resilience. Our multi-agent architecture separates data computation and data collection into different agent classes and services—providing a viable means to protect against truncation attacks.

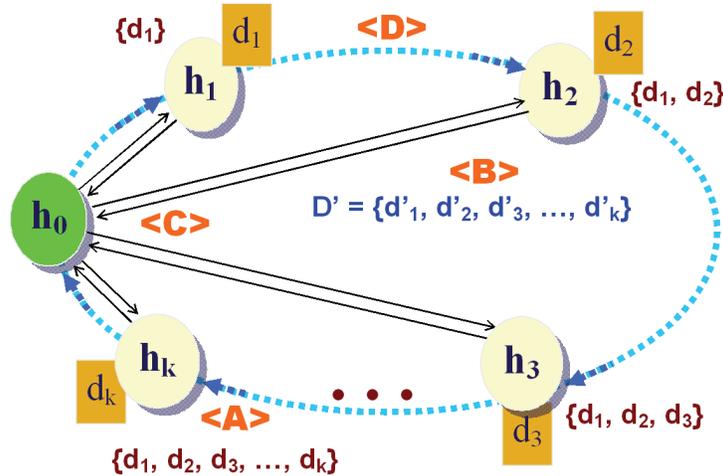
### 3.2.3 Describing the Problem

Initial work in mobile agents such as [18] identified two information-gathering modes: stateless and stateful. In a stateless approach, agents intermittently send information acquired back home to the originator or migrate home after each hop. In a stateful mode, the agent embodies in its data state results from prior host execution and carries with it a growing information collection to each subsequent host in the itinerary. *Independent data* comprises the offers, bids, or results that an agent uses to make decisions. Single-hop agents acting in a remote code execution paradigm [267] migrate to a remote host, operate on independent data, and then send results back to the home platform or migrate back to the home platform carrying the result. In other words, the agent “result” is independent from the “result” on any other host where the host carries out the same computation.

To illustrate independent data, an agent that carries out a “sum” operation can do so by collecting inputs from a host and storing each value in a data collection. When the agent returns home, the values stored in the collection are added together to complete the operation. The multi-hop agent data state in this example depends on previous agent executions only in the sense that the collected data item set must be carried forward faithfully from the previous host. In this case, malicious hosts carry out truncations, insertions, and deletions by modifying the “values” carried by the agent.

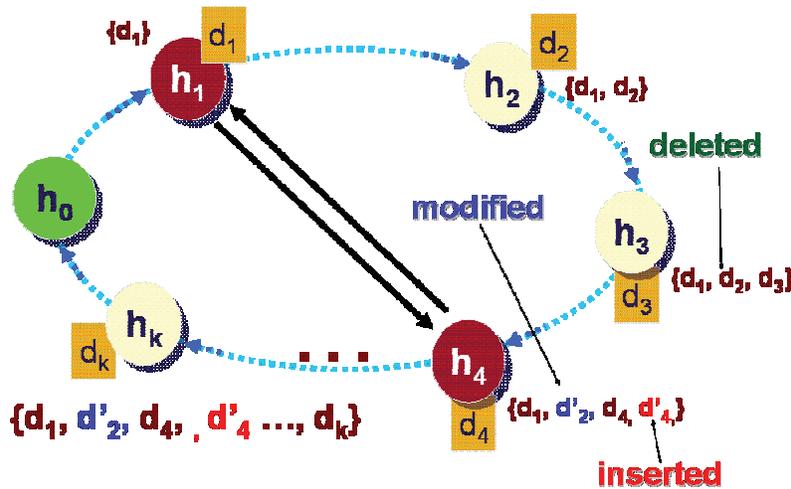
When an agent migrates from host to host performing such a query or computation in a multi-hop mode, the agent appends the current host results to previous results embodied in the agent. Figure 20, letter <C>, illustrates the agent migration paths for single-hop logic, where agents return to the originating host after each execution. One single agent performs this computation type by making  $k$  roundtrip migrations in a single-hop manner while  $k$  agents can migrate to each host independently and perform the same computation, where  $k$  represents how many executing hosts the agent visits. The application owner performs data fusion or sorting after the agent collects all host results.

In some agent applications, the agent computational result at state  $d_x$  depends on the computation results from previous agent states  $\{d_1, \dots, d_{x-1}\}$ . Figure 20, letter <D>, indicates the multi-hop agent path as the agent traverses a network, migrating from host to host carrying out computations. We represent in this application setting a competitive, electronic transaction where agents collect bids or offers in various contexts, such as airline reservation [18, 31, 144, 360]. A multi-hop bidding agent can be designed to embody all bids for each visited hosts in its data state and apply logic to determine the winner once all possible hosts are visited (thus utilizing *independent data*).



**Figure 20: Stateful/Stateless Agent Interactions and Data Integrity**

In the multi-hop approach, the data set grows linearly as each host executes the agent code, adding a data state to the migrating agent's protected area (we borrow the term "protected area" from standard literature on data protection [144] to describe agent data state encapsulations guarded by cryptographic techniques). On migration from  $h_1$  to  $h_2$ , for example, the data set grows from  $\{d_1\}$  to  $\{d_1, d_2\}$  after execution by  $h_2$ . The application owner can design the bidding agent to carry the lowest bid amount and the winner's identity in its state and to allow updates based on each host input (illustrating *dependent* data). Considering the "sum" example, an agent with dependent data carries a sum variable that each host in the itinerary updates by executing the agent code using their local input. The agent returns home with the sum calculated from the last host that it visited. We refer to independent data also as *data aggregation* because a correlation exists between the agent's previous and current execution state. For the multi-hop agent, Figure 20- $\langle B \rangle$  illustrates the set  $D'$  the agent *does* have on arrival back at the originating host and set  $D$ , Figure 20- $\langle A \rangle$ , indicates the data results it *should* have. We define *strong data integrity* along with other researchers [107, 144] as the ability to detect whether  $D, \{d_1, d_2, \dots, d_k\} \neq \text{set } D', \{d'_1, d'_2, \dots, d'_k\}$  on the agent's return to  $h_0$ .



**Figure 21: Data Integrity Attacks**

Figure 21 illustrates two colluding malicious hosts ( $h_1$  and  $h_4$ ) and three different integrity attacks. Strong data integrity mechanisms detect all truncation attacks even when colluding hosts are involved. On receiving the set  $D$ , the malicious host  $h_4$  may choose to do the following:

- (1) delete a data state ( $d_3$  as seen in the Figure 21);
- (2) insert a new fictitious state ( $d'_4$  as seen in figure);
- (3) modify an earlier state ( $d'_2$  as seen in the Figure 21); or
- (4) completely erase all previous states by using the data set  $\{d_1\}$  received from its malicious partner  $h_1$ .

### 3.2.4 Architecture Overview

We assert that the simplest method to prevent data integrity attacks comes from avoiding contact with potential malicious hosts. In MADIMA, we attempt to reduce or eliminate exposing partial data state results—preventing malicious attacks versus detecting them a posteriori. MADIMA also leverages both stateless (single-hop) and stateful (multi-hop) agents by using three different agent class interactions: *task agents*, *data computation agents*, and *data collection agents*.

A distinction exists between using the *same* agent logic replicated multiple times [277, 288] and using *different* agents to accomplish a single purpose-driven task [18], which our scheme utilizes. Kotzanikolaou *et al.* in [258] present architecture where a master agent and multiple slave agents conduct electronic transactions cooperatively. Slave agents are mobile and travel to only one particular host to negotiate, but cannot complete a transaction without returning to the master agent. Our approach resembles master/slave relationships in the sense that we use a master task agent that spawns and directs information gathering from multiple computation and collection agents, and then carries out any transaction logic separately.

Similar to the master/slave relationship in [258], the *task agent* in MADIMA serves to coordinate task efforts using the other two classes. *Data computation agents* perform a wide function range, but are dispatched either single-hop or multi-hop in different configurations depending on the requirements for security or reliability. Computation agents leave data results in publicly accessible data services known as *data-bins* rather than carrying results in their mutable state. *Data collection agents* visit hosts independently or the remote host generates them in response to computation agents visits. In either case, they carry results back to the application owner for fusion by the task agent. This approach solves the truncation attack problem from colluding malicious hosts by eliminating partial results exposure to malicious parties.

When we implement data aggregation in this manner, we have more freedom to use multi-hop agent logic. This architectural variation resembles execution tracing proposed by Vigna in [267], but we use communications with the host in our scheme to verify *integrity* of data (versus code execution) and we also *automate* collection activities (versus leaving them ad-hoc). The underlying data collection architecture supports other security measures that require log archival such as execution tracing and data encapsulation [31, 144, 286].

**Task Agents.** The *task agent* embodies an application owner's task desire, such as purchasing an airline ticket with fixed criteria set. This single agent directs the overall job. The task agent resides on the originating host or a trusted third party host that remains online (a buying service host for example). *Task agents* spawn either a single multi-hop agent or multiple single-hop agents to perform information gathering or bid requests. Spawned *computation agents* have fixed or free-roaming itineraries that visit host servers in a specific subject domain (such as airline reservation systems) and perform queries based on user criteria. The task agent waits until a minimum number of data results (specified by the user) are gathered or until a specified time elapses, at which point the task agent notifies the application owner the job failed. Upon receiving query results gathered by *data collection agents*, the task agent fuses data results. Transaction logic may allow the task agent to complete a financial commitment based on the query responses from the computation/collection agents via a single-hop computation agent.

**Computation Agents.** *Computation agents* traverse an ordinary mobile agent route and need to be uniquely identifiable to prevent the replay attacks expounded by Roth in [108]. As Figure 22 illustrates, a task agent can remain at the originating host or the application owner can transfer the agent to a trusted third party so that the originating host may go offline. The task agent contains the agent code for the various computations it requires and the itinerary. For

greater fault tolerance, the application owner can replicate computation agents as described in [277]. In either configuration, the data bin links each computational data result to the agent's identity, the originating host's identity, and a unique transaction identification known only to the application owner to support later pickup.

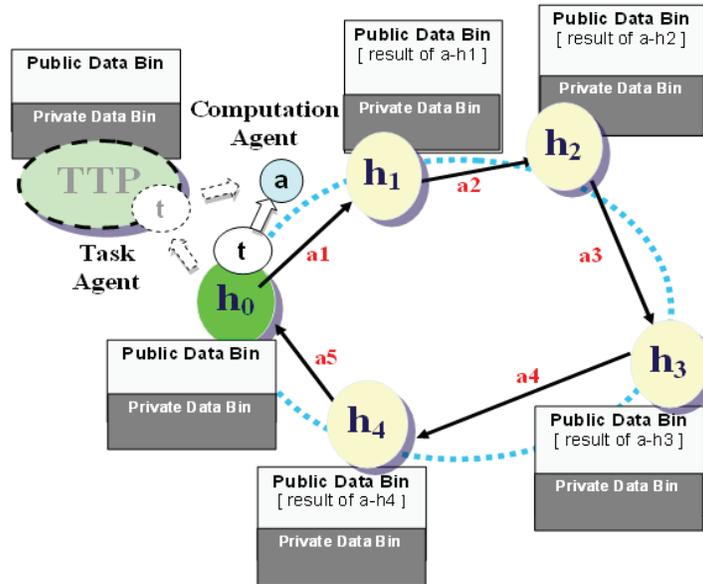


Figure 22: Launching Task Agent ( $t$ ) and Single-Hop Computation Agent ( $a$ )

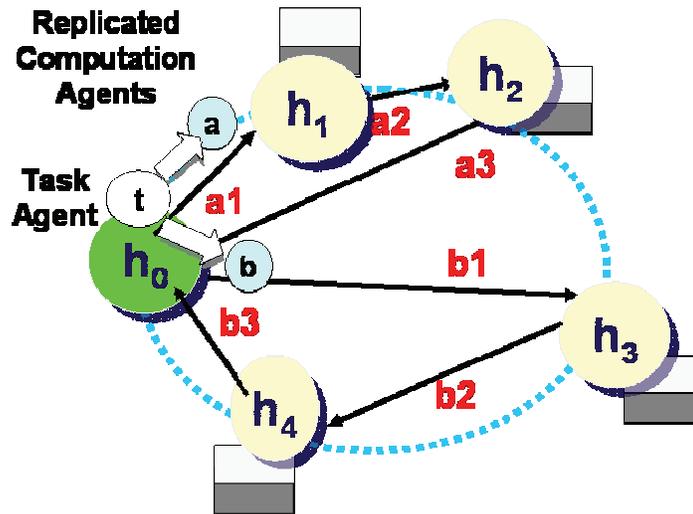


Figure 23: Using Replicated Computation Agents ( $a, b$ )

Figure 22 depicts an application owner spawning the computation agent ( $a$ ) that visits  $h_1, h_2, h_3,$  and  $h_4$  with migrations  $a_1, a_2, a_3, a_4,$  and  $a_5$ . Figure 23 depicts an application owner that launches two replicated multi-hop computation agents: agent  $a$  visits  $h_1$  and  $h_2$  with migrations  $a_1, a_2,$  and  $a_3$  while agent  $b$  visits  $h_3$  and  $h_4$  with migrations  $b_1, b_2,$  and  $b_3$ . At worst, a malicious host may only denial or delay service to the computation agent or keep back its own independent data result from the collection agent. We achieve authenticity and non-repudiability in MADIMA by binding the originating host's identification and the unique agent identifier with the agent data state. If multiple computation agents are launched single-hop, the computation agent leaves no data state simply returns to the originating host carrying the data result, as in remote evaluation operations [267].

**Data Collection Agents.** Data collection agents are responsible for the single-hop mission to carry back encapsulated data states or query results to the originating host. Figure 24 illustrates the data collection agent's activity. The task agent ( $t$ ) spawns the data collection agents after previously dispatching computation agent ( $a$ ) seen in Figure 22. Each data collection agent ( $a, b, c, d$  in Figure 24) stores its payload in the originating host's private data bin and notifies the task agent on arrival. In this aspect, MADIMA *data bins* provide private holding areas for data results to support task agent data fusion on the originating host and public holding areas where visiting computation agents may store results and collection agents retrieve them. Depending on whether agent developer uses independent or dependent data modes, the application owner can encapsulate data results in the computation agent using standard data integrity approaches such as [31, 144, 284, 286].

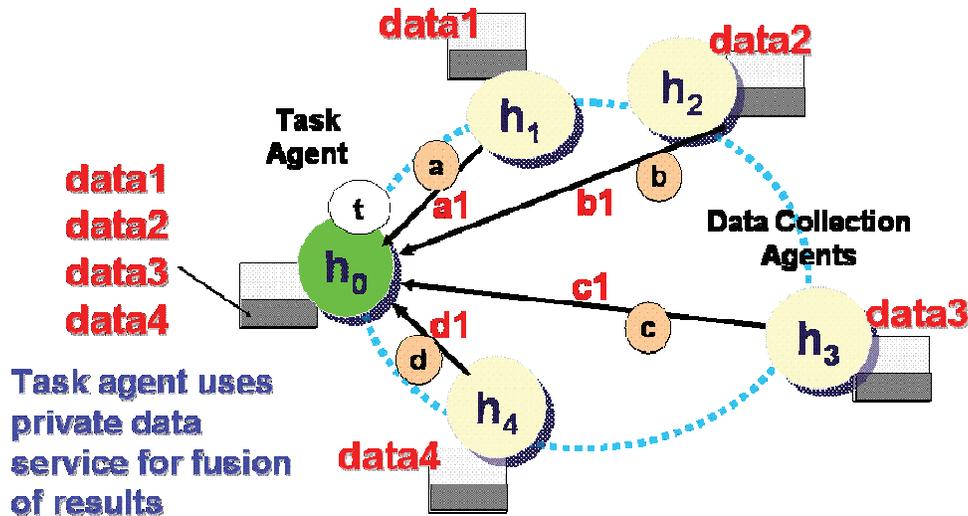


Figure 24: Data Collection Agents (a,b,c,d)

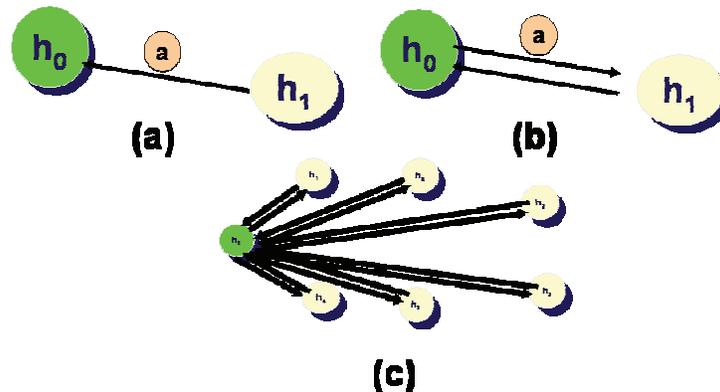


Figure 25: Data Collection Modes

Data collection agents are single-hop agents that have the highest security possible when they act in one-to-one relationship with executing hosts. As Figure 24 illustrates, data collection agents are executed in three possible configurations: Figure 24-(a), server-based response mode; Figure 24-(b), host-based request mode; and Figure 24-(c), autonomous data collection mode. In *server-based response mode*, each server visited by a computation agent spawns a data collection agent that performs an authenticated and encrypted single-hop result transfer. In the *host-based request mode*, the originating host sends data collection agents to each host in the computation agent's itinerary. The task agent responds to computation agent completion and

sends a collection agent in this mode. The *autonomous data collection mode* begins with a host that has just launched task agents or by agent servers who send results to trusted third party collection points on a recurring time interval; in either case, hosts spawn single-hop data collection agents that return a data result directly to the task agent/originating host. This approach resembles the “garbage collection” service that runs in background within the JAVA interpreter. With this method, we build data collection as a routine service that interfaces data bins with executing hosts and dispatching hosts.

In MADIMA, data collection protocols ensure that only the originating host / task agent can retrieve its own data results and that a task agent can request previous data results for fusion only at locations where a trusted computing environment exist. After all data collection activities have been accomplished and the task agent collates and filters results, it can spawn further computation agents that perform single-hop transactions or additional data gathering. In such cases where single-hop agents accomplish transactions like credit card billing, the architecture does not require data collection. To summarize, MADIMA utilizes three agent classes that use various interaction combinations to accomplish a user task.

**Data Collection Services (Bins).** MADIMA agent middleware uses a data service, referred to as a *data bin*, to store encapsulated (cryptographically protected) agent data states. *Data lockers* in [145] are described as a service provided for mobile users that keeps their data in secure and safe locations attached to fixed networks. We construct our data bin similarly: we use a data service to store intermediate agent computation results securely during an agent’s transit through a network. We incorporate data bins for *security* purposes versus the *convenience* normally associated with data lockers.

Data bins have a public locker, where computation and collection agents can store and retrieve results, and a private locker, where host-originating task agents can store partial results for later fusion. In independent data operations, computation agents do not arrive back at the originating hosts with a state payload containing a protected result set. Instead, the computation agent leaves the execution result (embodied in the mutable state or as a query result) on each host via the public data bin service, protected with an agreed upon encryption scheme. The data collection protocol ensures via authentication and non-repudiation that only the originating host can retrieve its own data results from a data bin.

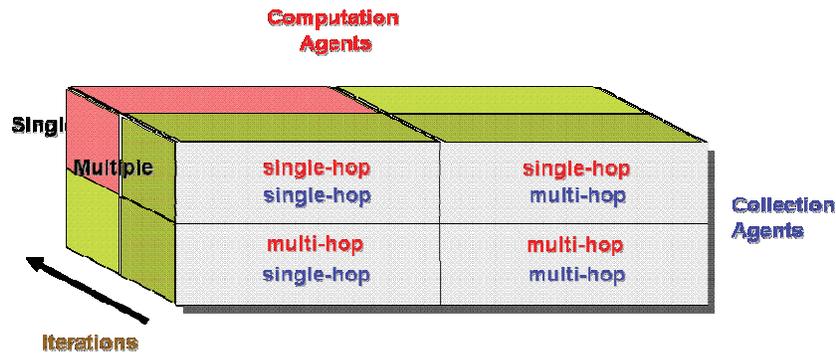
### 3.2.5 Related Security Issues

MADIMA relies on the general premise that hosts should perform agent *computations* separate from data state *collection* when using multi-hop agents. The computation agent’s static code must *interact* with partial results from previous computations in order to produce a new data state result. To perform a multi-hop task that relies on *dependent* data, a host modifies a data computation agent so that it carries only the *most recent* state as its payload, while copying and leaving a secure encrypted state version behind at each host server. We use data collection agents in this configuration as a verification authority because the application owner must compare the final agent data state against the incremental data states gathered by collection activities. MADIMA supports detecting truncation violations when computation agents use dependent data mode, but cannot *prevent* truncation attacks by multiple colluding hosts when computation agents use dependent data.

Figure 26 illustrates the design space for the three MADIMA agent classes. The agent class number (iterations) and types (single-hop/multi-hop) define the design space for a MADIMA application. Single-hop computation agents used with single-hop collections agents provide the strongest security associations possible, because interactions are always one-to-one with the application owner and remote host. *Typical* mobile agent scenarios for MADIMA envision a *single* multi-hop computation agent and *multiple* single-hop collection agents.

Because intermediate results in a typical MADIMA operating mode are subject only to single host malicious activity, we prevent manipulation, extraction, and truncation attacks on information accumulated in a multi-hop free-roaming scenario. We do not address whether a server has provided false information to the agent. We assume that alterations to the static agent code are detectable by honest hosts when we employ measures such as code signatures [31, 288] or

execution tracing [267]. We also assume that a public key infrastructure exists or that the ability to distribute shared secrets among participants exists.



**Figure 26: MADIMA Security Configurations**

MADIMA does not address service denial or random alterations to the code. When multi-hop agents with dependent (aggregated) data are used, computation agents still need the ability to mask or guard the function against smart code alterations. We do not address the ability to keep keys used by both the computation and collection agent private in this architectural description, though Chapter 5 provides positive techniques for doing so and other related work for white-box key protection are described in [146, 147].

Concerning multiple agents, Tate and Xu utilize multiple parallel agents that employ threshold cryptography to eliminate the need for a trusted third party in [288]. Tate and Xu observe their work was first to consider multi-agent settings solely for their security benefit. Endsuleit follows suit with several multi-agent architectures for security as well [344, 353]. In MADIMA, we show continued benefit for using multiple agents in mobile contexts to exploit security advantages.

### 3.2.6 Fault Tolerance Issues

The MADIMA architecture uses multiple agent *classes* given *information computation* or *data gathering* duties. The fault tolerance domain finds benefit for using multiple agents to provide guarantees on agent migration and task completion. Researchers historically focus on integrating fault tolerance to increase mobile agent system reliability (see Appendix A.3.15). Minsky *et al.* [277] propose that replicated agents and voting can decide if malicious hosts have altered agent execution. Yee proposes a mechanism to detect replay attacks in [74] while Tan and Moreau extend an execution-tracing framework in [33] to *prevent* service denial attacks.

Several fault tolerance issues arise in the MADIMA approach, just as in other schemes. For example, when a data bin service exceeds storage space allocated by the host, data bins implement a queue process (much like routers discard packets under certain load conditions). We use one or more trusted third parties for data collection activities or task agent hosting to support disconnected host operations. We mitigate task agent time-outs while waiting for computation and data collection results by providing time-based services that indicate when (computation/collection) agents are unreasonably detained or diverted.

Like other multi-agent or mobile agent systems, we do not address error recovery procedures when messages are undelivered or when migration is blocked. If data bin services fail, we envision that secondary storage services in the network are present if the originating host or buying service TTP becomes unavailable. We mitigate the original task agent's failure, failure of one or more computation agents, and failure of data collection agents by considering such approaches as the shadow model of [279]. Other work on fault-tolerance such as [148, 149, 277] provide approaches to mitigate host failures caused by malicious activity.

### 3.2.7 Performance Issues

In most cases, security comes at a cost. Multiple interacting agents bring more complexity and performance overhead to a system and we consider added security benefits against increased communications costs. We express performance issues as the difference between a

normal multi-hop agent that carries results with it and returns back to an originating host versus a static task agent that spawns one or more computation agents and receives responses from one or more collection agents.

A traditional migrating mobile agent visits  $k$  servers and performs  $k+1$  migrations (see Figure 20). The agent size grows linearly according to the added data state based on the query result size. When we use dependent data in the agent logic, the agent data state may not grow appreciably at all. For MADIMA, overhead increases by using a single static task agent (present on the originating host or a trusted third party) and by using computation agents with  $k+1$  migrations (assuming a multi-hop traversal). At least  $k$  additional data collection agents communicate with data bin services and transport results back to the host. In sum, MADIMA doubles the network transmission by  $(2k + 1)$  and increases consumption resources due to interactions from three agent classes.

### 3.2.8 MADIMA Summary

Various agent security schemes enforce various data-integrity security levels. MADIMA prevents data integrity attacks against mobile agents, especially truncations when multiple colluding hosts are present, by separating basic agent duties (computation from collection). Though other approaches communicate agent state forms to other agents or the originating host, MADIMA implements this approach in three cooperating multiple agent classes and introduces a data bin service to facilitate data computation and collection activities. We believe this approach demonstrates several security benefits:

- 1) We limit the impact any one malicious host can have on another host.
- 2) No malicious host can influence previous computations during computation agent execution.
- 3) Adversaries that wish to impact future computations in a multi-hop computation must perform smart code alteration, which we address further in Chapter 5.
- 4) We reduce integrity attacks on data state to denial of service.

We leverage the fact that an agent *must* carry computation results back at some point to the originating host; at a minimum, the agent must act upon inputs and interact with the originating host based on the result. We separate *data computation* activities from *data collection* activities to eliminate incremental result exposure to possibly malicious hosts. Whether the agent stores data results as a single modified agent state or as results embedded in different agent state values, the agent either carries data state with it or (under MADIMA) leaves the state at the host for delivery by more secure means. The multi-agent approach allows us to develop applications in a conceptual manner by leveraging agency while preventing attacks on data integrity. We now discuss our second multi-agent approach based upon secure multi-party computation.

## 3.3 Hybrid Approaches for Secure Multi-Agent Computations

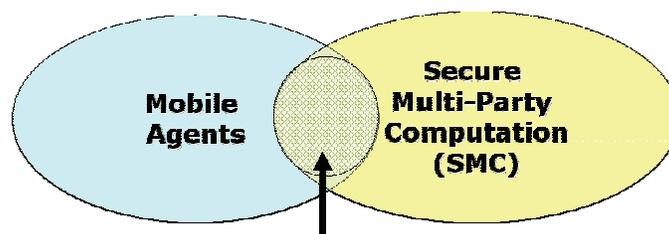


Figure 27: Secure Multi-Agent Computations

In this section, we deal specifically with how a host can keep its data input private while guaranteeing agent task execution integrity. In [143], we review and analyze methods proposed for securing agent operations when passive and active adversaries are present by using secure multi-party computations (SMC). As Figure 27 depicts, we explore specifically architectures that

support secure multi-agent computations. For greater context concerning hybrid SMC architectures, we refer the unfamiliar reader to Appendix A.5 where we discuss in detail SMC strengths and weaknesses and review research associated with mobile agent integration. We begin with a brief review on SMC integration issues with agents and then present two hybrid schemes that reduce communication overhead and maintain flexibility when applying particular protocols.

### 3.3.1 SMC Integration with Mobile Agency

A secure multi-party computation has  $n$  players,  $(P_1, P_2, \dots, P_n)$ , who wish to evaluate a function,  $y=F(x_1, x_2, \dots, x_n)$ , where  $x_i$  represents a secret value provided by  $P_i$  and  $y$  represents the (public) output. The protocol goal is to preserve player input *privacy* and guarantee computation *correctness*. SMC protocols offer several advantages for securely accomplishing a group transaction and have been a major thrust for possibly achieving code privacy in mobile contexts. Figure 28 depicts a mobile agent transaction as an idealized SMC protocol. We achieve idealized perfect security in SMC when all parties (hosts) securely provide their inputs to a trusted third party (TTP). The TTP executes function  $F$  on all inputs and we can hold the result private for only one party ( $P_0$ ) or give the result to all parties involved. Figure 28 depicts private host inputs  $(x_1, x_2, x_3, \dots, x_n)$  and a public function output  $y = F(x_1, x_2, x_3, \dots, x_n)$ . Functions such as mean, max, min, set intersection, and median are common SMC protocol questions.

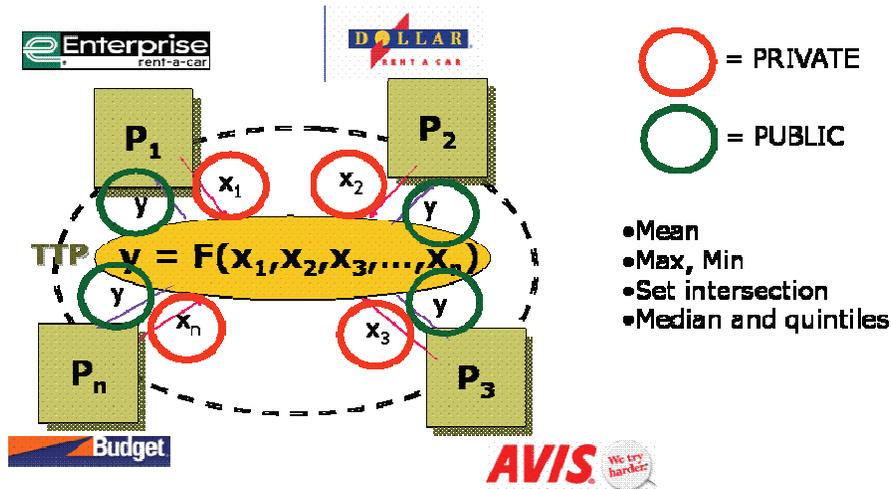


Figure 28: Agent Task Realized as Secure Multi-Party Computation

SMC protocols solve problems where inputs have the same length and where we compute functionality in time polynomial on the input length. We measure security based on the input length (using inputs  $1^n$ ) and we cannot attain greater privacy beyond the idealized TTP. When parties are motivated to submit their true inputs and can tolerate function result disclosure, we can securely implement the protocol without a TTP. Several approaches exist that define agents implementing garbled circuits in multi-party computations and that use oblivious transfer to evaluate the circuit. The application owner can send a single agent with a cascading circuit whose last migration signals the last circuit computation. Alternatively, the owner can send multiple agents with the same circuit that executes protocols in stepwise multi-round fashion. We can use a single trusted execution site or multiple TTPs connected via high-speed communication links to evaluate the SMC protocol. By combining these SMC protocols, multiple agents, and semi-trusted hosts, we achieve several security specific goals for mobile agents.

The security/threat models for SMC traditionally protect against passive adversaries that steal private inputs or protect against active adversaries that corrupt the function output. We assume TTPs do not collude and assume individual parties involved in the transaction do not collude. Any given SMC protocol specifies a maximum tolerable limit for active and passive malicious parties. The overhead for multi-round protocols comes from large numbers of small message exchanges and for single round protocols comes from transferring one large message in non-

interactive mode. Multi-round interactive protocols typically assume a perfect network/broadcast channel.

Non-interactive approaches are limited to a few protocols that derive from [25] or [105]. Single-round approaches do not require trusted third parties but come with large message sizes and their own limitations that include reliance on a trusted *entity* similar to a PKI. Tate / Xu [288] and Zhong / Yang [299] extend traditional garbled circuit non-interactive approaches with multiple agents and both architectures require knowing the visited host set before execution. Researchers continue to improve SMC protocol efficiency and we develop our agent architecture in a manner to integrate them seamlessly.

Architectures that implement SMC in mobile agent systems seek to reduce message size, number of broadcast channels required, and circuit size. To accommodate agent goals such as disconnected operations, the originator typically remains offline during the protocol evaluation. In order to support agent autonomy, we require that the agent can decide where and when to migrate. The requirement for full autonomy in the agent path and itinerary lends itself best to SMC protocols that balance trust with efficiency. While we desire to eliminate the need or requirement for any trusted third party or trusted computation service (like PKI), some application environments for SMC tolerate such assistance with no problem.

Malkhi *et al.* [335] note that SMC protocols find greater efficiency when implemented for specific tasks and this motivates researchers to focus on protocols that work in specific application contexts (like secure voting). We find this true for mobile agents as well and seek to represent agent specific tasks like auctions, trading, or secure voting with greater efficiency. Fiegenbaum *et al.* [150] implement a secure computation mechanism utilizing SMC named FAIRPLAY for collecting survey results with sensitive information. Their scheme uses data-splitting techniques and traditional Boolean circuit evaluation Yao-style [315]. Notably, FAIRPLAY uses a secure computation server, which acts as a trusted entity within the system, and initiates the 2-party function evaluation. Applications like FAIRPLAY illustrate a practical SMC implementation where the application achieves data privacy and function integrity, but the environment supports using a trusted server. Agent applications executed “in-house” benefit directly from trusted (or partially trusted) entity status.

We now introduce several hybrid approaches to SMC integration with mobile agents that can *accommodate free-roaming itineraries* as well as *reduce overall communication cost*. We deem the architectures hybrid because they account for both the strengths and weakness found in traditional multi-round SMC protocols. Communication costs remain high for multi-round protocols; we mitigate this by using trusted or semi-trusted execution sites connected via high-speed network connections. Flexibility for SMC is limited in mobile agent environments to fixed itineraries; we mitigate this by agent classes that support free-roaming itineraries while supporting traditional SMC. These approaches leverage multiparty protocol security properties while providing flexibility to integrate higher efficiency protocols in the future.

### 3.3.2 Invitation and Response Protocol

We define the Invitation and Response Protocol as a multi-agent architecture that uses semi-trusted execution sites. We define two agent *classes*: the *invitation* agent and the *response* agent, illustrated in Figure 29. We define a user task  $F$ , executing hosts  $H_1, H_2, \dots, H_n$ , with private inputs  $(x_1, x_2, \dots, x_n)$  to  $F$ , and one or more fully-trusted or semi-trusted execution servers  $ES_1, \dots, ES_z$ . We design our protocol to use *any* SMC protocol that is provably secure against passive and active adversaries; we stipulate minimum protocol security that meets Canetti’s composable security properties [352]. We delineate four phases in the protocol and elaborate them in Figure 30 and Figure 31: *Initialization, Invitation, Response, and Recovery*. We refer to the task owner  $F$  as the originator  $O$  and textually describe the protocol next.

After initialization, the originator  $O$  begins the task by sending an invitation agent that has an initial host set or that at least knows the first host. *Invitation agents* are free roaming and can make changes in their itinerary based on environmental conditions or information obtained from hosts or other information services. To guard the invitation agent against data integrity or service denial attacks, two different schemes are possible. First, a single multi-hop agent (depicted in Figure 32-(a)) can use data encapsulation techniques (see Appendix A.4) to protect its itinerary

and perform a free-roaming traversal. We assume the application owner binds the agent code to each agent's dynamic state instance. A second approach (Figure 32-(b)) is to use multiple invitation agents with possibly overlapping and redundant itineraries to reduce blind malicious intervention (service denial).

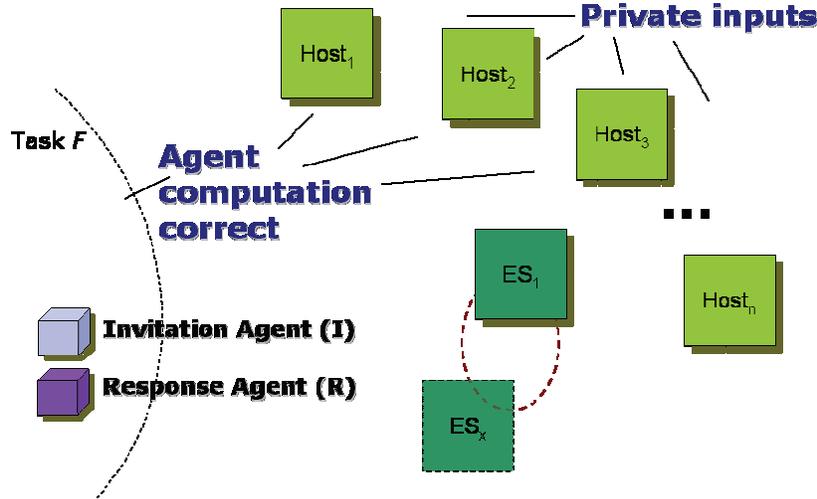


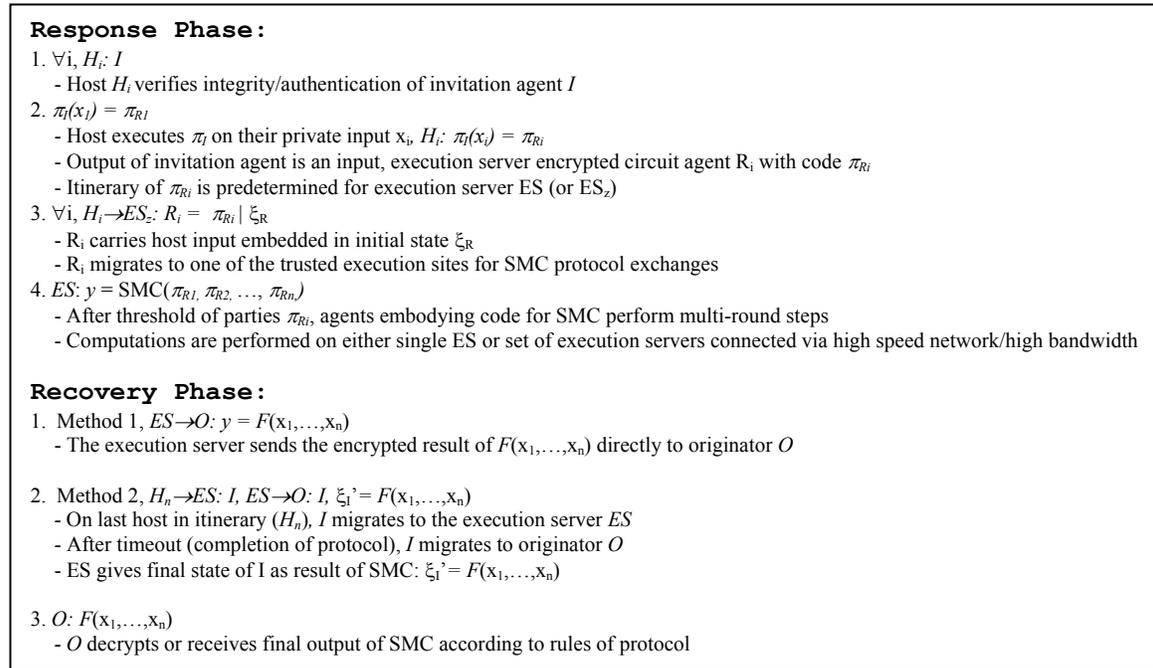
Figure 29: User Task  $F$  Implemented as Secure Multi-Agent Computation

<p><b>Initialization Phase:</b></p> <ol style="list-style-type: none"> <li>Originator <math>O</math> creates Response Agent <math>R</math> with code <math>\pi_R</math>  <math>\pi_R</math>: code of <math>R</math> <ul style="list-style-type: none"> <li>Implements any multi-round SMC protocol realizing task function <math>F</math></li> <li>Migration from host <math>H_i</math> with local input <math>x_i</math> commits host to the input</li> </ul> <math>\xi_R</math>: initial data state of <math>R</math> <ul style="list-style-type: none"> <li>Initialized at destination host <math>H_i</math> w/ private input <math>x_i</math></li> </ul> <math>\xi_R'</math>: final data state of <math>R</math> <ul style="list-style-type: none"> <li>Evaluated through SMC exchanges on <math>ES_z</math></li> </ul> <math>I_R</math>: itinerary of <math>R</math> Itinerary (single-hop) <ul style="list-style-type: none"> <li><math>\{ES_z, O\}</math></li> <li>Protected by data encapsulation technique</li> </ul> </li> <li>Originator <math>O</math> creates Invitation agent <math>I</math> using <math>\pi_I</math>  <math>\pi_I</math>: code of <math>I</math> <ul style="list-style-type: none"> <li>Embedded with <math>\pi_R</math>, returned by host output function</li> </ul> <math>\xi_I</math>: initial data state of <math>I</math> <ul style="list-style-type: none"> <li>Uniquely identifiable</li> <li>Based on nonce, cryptographic hash of agent ID</li> </ul> <math>\xi_I'</math>: final data state of <math>I</math> <ul style="list-style-type: none"> <li>Received back by originator <math>O</math></li> </ul> <math>I_I</math>: itinerary of <math>I</math> (multi-hop) <ul style="list-style-type: none"> <li><math>\{O, H_1, H_2, \dots, H_n, O\}</math> fixed</li> <li><math>\{O, H_i\}</math> free-roaming</li> <li>Protected by authentication, integrity mechanism</li> </ul> </li> </ol>	<p><b>Invitation Phase:</b></p> <ol style="list-style-type: none"> <li><math>O \rightarrow H_i: I</math> <ul style="list-style-type: none"> <li>Originator <math>O</math> dispatches <math>I</math></li> <li>Single, dynamic, multi-hop</li> <li>Migrates to one execution site</li> <li>Contact time for protocol execution required</li> <li>Code/itinerary integrity assumed</li> </ul> </li> <li><math>H_i: \pi_I(x_i) = \pi_{Ri}</math> <ul style="list-style-type: none"> <li>Invitation agent <math>\pi_I</math> accepts input and dynamically generates circuit agent <math>\pi_R</math> based on host input</li> <li>Host input is encrypted protected with public or threshold key of ES: <math>\{x_i\}_{K-ES}</math></li> </ul> </li> <li><math>H_i \rightarrow H_{i+1}: I</math>  <math>H_{i+1}: \pi_I(x_{i+1}) = \pi_{Ri+1}</math> <ul style="list-style-type: none"> <li>Each host <math>H_{i+1}</math> evaluates input <math>x_{i+1}</math> on agent and receives response agent <math>\pi_{Ri+1}</math></li> <li>Each host must choose to execute response agent (agent itinerary is pre-established) by sending to execution site</li> </ul> </li> <li><math>H_n \rightarrow O: I</math> <ul style="list-style-type: none"> <li>Migration back to owner of invitation agent(s)</li> <li>Originator <math>O</math> verifies integrity of <math>\xi_I'</math></li> <li>Timeouts are based on both return of invitation agent and response of SMC protocol evaluation on ES</li> </ul> </li> </ol>
--	--

Figure 30: Initialization and Invitation Phases

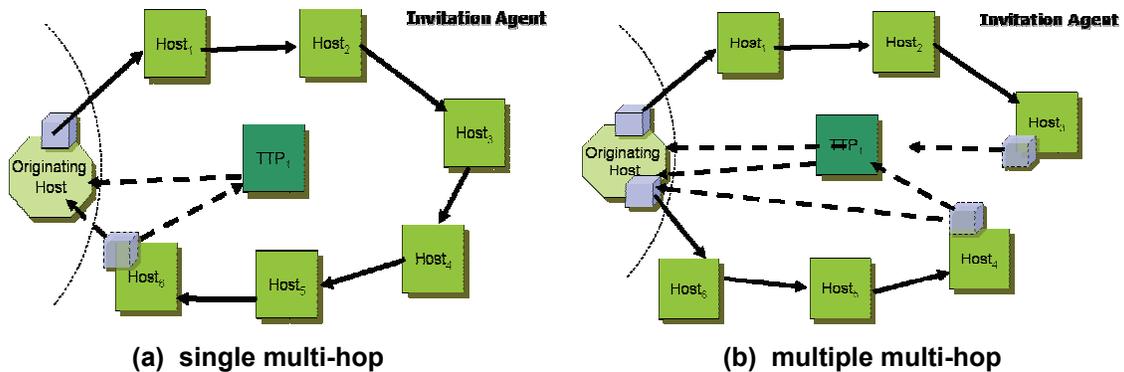
Each invitation agent has a uniquely identifiable code/state (to avoid replay attacks), but the agent collection represents a single uniquely identifiable task (such as a specific auction or airline ticket purchase). If an executing host receives an agent requesting participation in the same unique event (bid, auction, etc.), it ignores subsequent requests much like network devices that only forward packets once. Invitation agents carry with them the specifications for *input*

corresponding to the originator's task. The specification represents the normal host query in a multiparty computation. Hosts respond to the invitation by dispatching the response agent obtained by executing the invitation agent on the host with their private input.



**Figure 31: Response and Recovery Phases**

We base the response agent's code on the underlying secure multi-party computation protocol (based traditionally on garbled circuits) and we spawn them within our protocol using three different methods. First, the invitation agent can carry the code for the response agent that each host uses for response. The host will execute the response agent first on its local input and then send the response agent to a semi-trusted execution location to actually evaluate the circuit (the protocol runs described in Figure 30 and Figure 31 assume this approach). The second approach involves the invitation agent *dynamically* generating the code and circuit when a host responds positively with their input. A third method would involve each host responding to the invitation by sending its input encrypted directly to the semi-trusted execution site. This method implements the ideal SMC environment where parties send their input to a TTP for protocol execution.



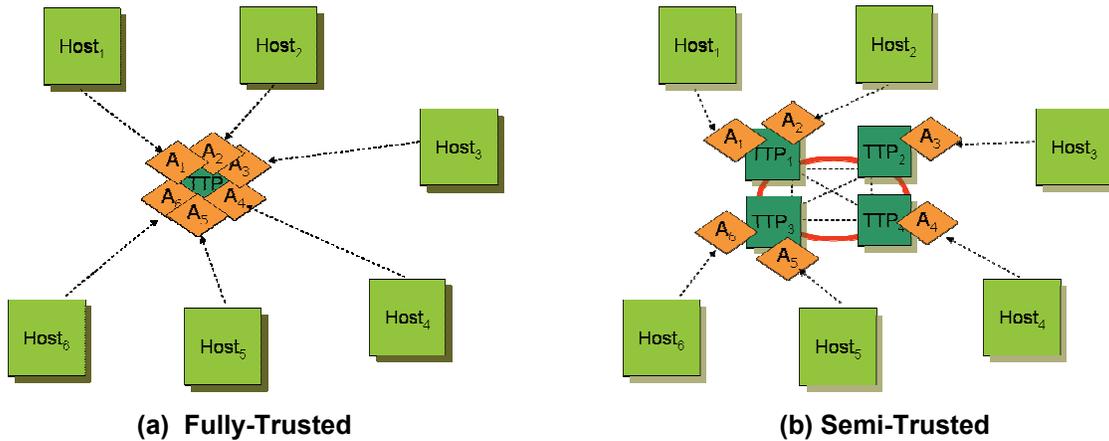
**Figure 32: Invitation Agents Sending Host Requests**

No matter which method we use, response agents in the protocol migrate to semi-trusted host environments in order to evaluate the protocol (depicted in Figure 33). The semi-trusted hosts are specifically designed to serve multi-party computations (predefined based on an underlying protocol) or provide basic agent execution environments with communication facilities. An adequate high bandwidth network to keep communication costs negligible must connect these hosts. Neven *et al.* [355] suggest using agents in such manner and they bring agents closer together by using high-speed communication links among servers in their architecture (see Figure 136 / Appendix A.4 for a more detailed explanation). Environments are semi-trusted because group and threshold operations eliminate the full trust in any one server. We describe one server in presenting example protocol runs below.

In security terms, “Invitation and Response” demonstrates the following properties. Hosts can only send one agent to the computation response—removing the possibility that a host evaluates a task circuit on multiple host inputs to game the task outcome. As long as we detect multiple host submissions (and therefore cheating), we preserve the originator’s privacy. We keep the local host input private under three scenarios:

- 1) When the execution sites are fully trusted, as depicted in Figure 33-a, we require no extra security and expect the single execution site to maintain host input privacy.
- 2) When execution sites are semi-trusted, as depicted in Figure 33-b, we use multiple trusted sites so that no one TTP receives all host inputs.
- 3) When execution sites are semi-trusted, as depicted in Figure 33-b, we may also use threshold mechanisms and data shares to distribute trust among all execution sites.

The hybrid approach advantage includes the ability to accommodate true free-roaming agent scenarios and to use any secure multi-party protocol secure function evaluation. We therefore favor protocols with high communication and low computational complexity because we send agents to a semi-trusted environment that has an assumed high-speed link among execution sites. For fully trusted execution environments, we depend on execution servers to follow all the rules and to prevent malicious tampering in order to achieve privacy and integrity. Semi-trusted environments (which map more accurately to real world scenarios) only operate on data shares where the SMC protocol uses threshold secret sharing schemes.



**Figure 33: Response Agents and Execution Environments**

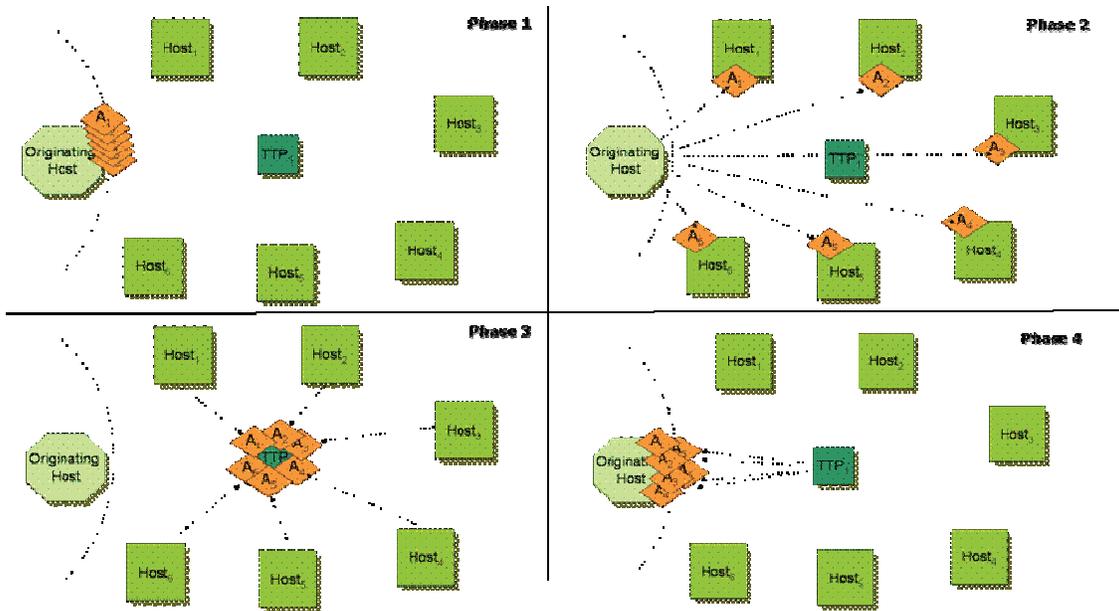
Selecting execution environments become an issue with the invitation and response protocol. The two primary requirements include high-speed communications link between all servers and a common trust level among all protocol parties and the trusted servers. Response agents only make two subsequent migrations: to the trusted server environment and then back to the originator, who can decrypt the final agent state and obtain the result.

Algesheimer *et al.* in [27] discuss a common SMC/agent integration issue concerning how the host gets its *local* output as the mobile agent migrates. In invitation and response, we handle local host output in two different ways. First, since we do not need to keep the function output private for the involved protocol hosts, we let the originator *O* provide the output to each host after the execution servers complete the secure function evaluation and response agents migrate back to the originator. Second, execution sites may send the host output share or host function output back to each host through message passing or a second agent.

### 3.3.3 Multi-Agent Trusted Execution

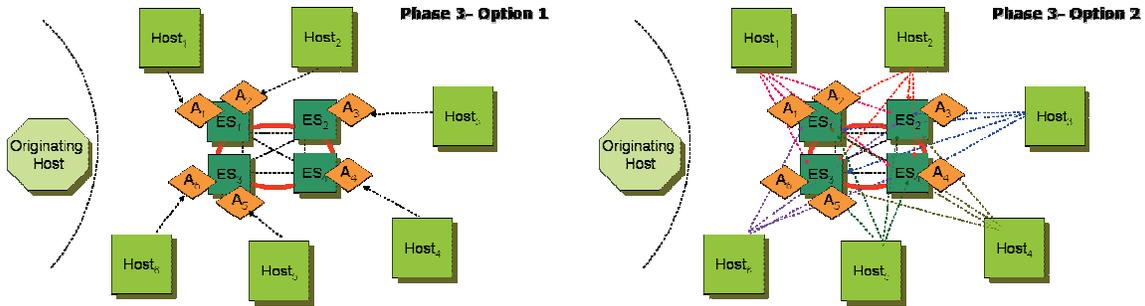
When we know the agent itinerary *beforehand*, we can use simpler agent architecture to facilitate trusted execution. Several configurations are possible for host environments that evaluate a secure computation. First, we can allow the host to act as computation environment for a cascaded circuit that requires only one execution round. Next, we can allow the host to communicate with a semi-trusted party to evaluate an encrypted circuit or to communicate with semi-trusted parties that provide threshold signal decryption services in an oblivious manner. We allow the host to be the computation environment for a multi-round circuit that receives visits from by more than one agent.

Figure 34 illustrates the four phases used in simple multi-agent trusted execution with a fully trusted intermediary execution site. Multiple agents are used to initiate a multi-party protocol among predefined hosts (beginning with Figure 34-Phase 1). Similar to the approaches used by Endsuleit *et al.* [344, 353], multi-agent trusted execution begins when agents migrate to prospective hosts, gather input (Figure 34-Phase 2), and then evaluate the protocol. When all parties fully trust *one* trusted execution environment, agents can then migrate there to accomplish a multi-round protocol, as suggested by Neven *et al.* [355] and as depicted in Figure 34-Phase 3. Upon completing the multi-round protocol, each computation agent  $A_i$  migrates back to the originating host (Figure 34-Phase 4), which can obtain the task outcome  $y = F(x_1, x_2, \dots, x_n)$ .



**Figure 34: Fully-Trusted Middle Man with Multi-Agent/Multi-Round SMC**

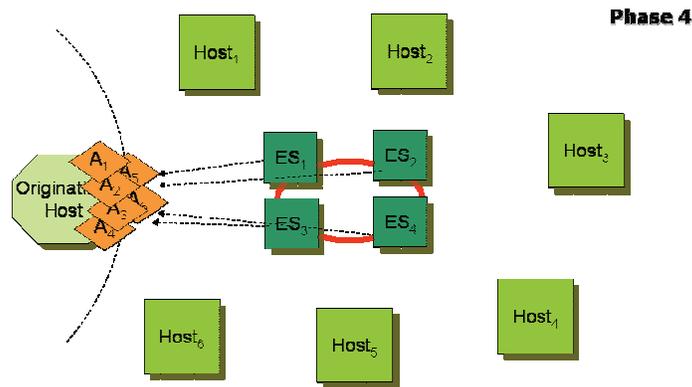
In a less trusted environment (where a server might be corrupted or stolen), we require multiple trusted execution sites linked by a high-speed and high-bandwidth communications network. Agents migrate to each host and migrate to the trusted execution site as before (Phase 1, Phase 2). Once agents obtain host input they migrate to a centralized trusted execution site and evaluate a multi-round SMC protocol. Two possibilities exist for Phase 3, as illustrated by Figure 35, Option 1 and Option 2.



**Figure 35: Phase 3 with Semi-Trusted Middlemen Execution Sites**

Figure 35-Option 1 depicts an SMC protocol where the agent carries its (unified) input and begins protocol execution after receiving a minimum number of participants. We assign agents to execution sites such that no one execute site receives all host inputs/agents. Figure 35-Option 2 depicts a secret-sharing scheme where execution sites have only a share of each host's data input. In this option, the SMC protocol chosen must protect input values using threshold encryption and decryption. This option supports shared secret schemes such as Zhong and Yang's protocol [299] for code that requires integrity and confidentiality when TTPs may collude (or face physical threats and corruption).

The trusted execution sites under Option 2 use cryptographic primitives such as verifiable distributed oblivious transfer (VDOT) perform operations based on Shamir's secret sharing scheme [278, 299]. Verifiable secret sharing schemes allows operations on data shares distributed among different parties. By using sharing techniques, parties give inputs shares so that honest protocol parties can detect any attempt to alter a commitment. Security primitives such as VDOT appeal to the security of Yao's secure circuit evaluation, the security of the 1-out-of-2 oblivious transfer, and the strength of threshold cryptography. Appendix A.5 provides a more in-depth discussion concerning these topics. After completing the protocol execution, all agents migrate to the originating host where share reconstruction takes place and the task owner receives the function output  $y = F(\dots)$  from the shares.



**Figure 36: Phase 4 Migration to Originating Host**

These hybrid architectures use underlying SMC protocol strength, mitigate performance overhead via a fully-trusted or semi-trusted execution network, and use agents that have a three-step itinerary among known possible input hosts. The agent code in this scheme uses a three step process involving migration/host data input gathering, migration/SMC protocol evaluation, and migration/data recovery with the originator.

### 3.3.4 Hybrid SMC Approach Summary

We illustrate with our hybrid architectures the distinct trade-off with integrating secure multiparty computations and mobile agent applications. We overcome the computation and communication barriers and use generic SMC protocols in a more practical manner. We have

defined two hybrid approach variations that utilize fully-trusted or semi-trusted execution environments for secure multi-agent computations. These schemes offer an alternative to other architectures suggested to date; these protocols combine advantages related to non-interactive approaches and multi-round SMC approaches. Invitation and response supports free-roaming agent scenarios where we do not know all hosts beforehand while multi-agent trusted execution enforces integrity and privacy despite colluding intermediate servers by using semi-trusted threshold data sharing. Our future work in this area involves analyzing overhead that comes from specific SMC protocol implementations.

### **3.4 Chapter Summary**

Multi-agent architectures yield benefit in solving specific mobile agent security headaches. Though they cannot address all mobile agent security requirements, we demonstrate in this chapter how specific multi-agent architectures enforce particular security requirements: partial result state protection despite multiple colluding hosts, guaranteeing host data input privacy, and supporting code execution integrity. We focus now on the role trust plays in mobile agent security decisions and specifically show how trust-related security decisions are useful in ubiquitous environments for mobile agents.

## CHAPTER 4

### MOBILE AGENT TRUST FRAMEWORK

This chapter contains material from our published work on application security models appearing in *Electronic Notes on Theoretical Computer Science* [151]. A separate technical report [152] provides further background material related to our results in this area as well. We provide in Appendix A.7 background literature relating to trust and give particular attention to how it applies to mobile agent contexts. Appendix B provides illustrative demonstration for various trust model properties that we define in this chapter.

#### 4.1 Chapter Overview

Traditionally, mobile agent security focuses on two protection issues: keeping malicious parties from altering the agent and keeping malicious agents from harming other parties including potential hosts. Several surveys [21, 22, 222] categorize and describe attacks against agent systems along with mechanisms for their defense. Researchers have given trust formulation considerable thought in both distributed networking applications [35, 36, 37, 38] and mobile agents [39, 40, 41, 42, 152]. Mobility as an application feature complicates trust because the receiving execution host must make distributed trust decisions with little or no prior knowledge. Likewise, user agents must evaluate trust with hosts in different security contexts. To date, other trust models for mobile agents have not addressed how to link requirements with appropriate agent protection mechanisms. Other trust models lack ability to integrate generic security mechanisms or reasoning about initial trust relationships. We bridge this gap by developing a trust-based security framework for mobile agents with three novel features:

- *Ability to link application security requirements with mechanisms based on trust*
- *Reasoning about trust properties for generic security mechanisms*
- *Application models for initial trust among principals in a mobile agent setting*

Our trust framework addresses several shortcomings in current models: handling generic trusted servers, describing generic security mechanisms, incorporating distributed trust paradigms, incorporating non-Boolean trust levels, relating applications to a security model with initial trust, modeling agent replay and cut/paste attacks, dealing with multiple agent interactions, and describing static interactions. We present results in this chapter in the following manner: first, we define security requirements as they relate to mobile agents and review how mechanisms relate to requirements (Section 4.2); we then define a trust framework that precisely defines all parties in the mobile agent application (Section 4.3). The trust framework delineates principals, trust relationships, trust decisions, and the role played by trusted hosts; it further provides a mechanism to express trust algorithms for mobile agent applications (Section 4.4). Based on this framework, we introduce *application security models* as a novel concept and show their relevance to mobile agent application design (Section 4.5).

In the security sense, models are useful for several things such as: helping test a particular security policy for completeness or consistency, helping document security policies, helping conceptualize or design an implementation, or verifying an implementation satisfies a requirements set. Our model for trust expression in mobile agent environments incorporates three separate notions: security requirements, security mechanisms, and trust in a mobile application setting. We examine requirements and mechanisms first.

## 4.2 Security Requirements and Mechanisms for Mobile Agents

We use trust to describe relationships among parties that have some behavioral expectation with each other. Models define participants in a system and the rules participants follow in interaction. In order to accommodate future mobile applications, we need a new model for describing mobile agent interactions based on trust *and* security. Agent systems and mobile applications need to balance security requirements with available security mechanisms in order to meet application-level security goals. Linking trust with security requirements, linking participants with trust levels, and creating models for expressing mobile agent interactions are key steps toward ubiquitous computing goals involving agents.

Practitioners routinely define *security requirements* as the desire to guarantee one or more specific properties: privacy, integrity, availability, authentication, and non-repudiation. Table 7 summarizes agent/host security requirements derived from the traditional CIA model and provides an abbreviation code for reference. We target *security mechanisms* at enforcing one or more of these requirements. We design our framework with the ability to express *security requirements* and link those requirements with *security mechanisms*—using trust relationships as a basis for evaluating their required use. Security requirements dictate both what are necessary for agent task accomplishment and the trust expectation that hosts have when interacting with an agent. To achieve these requirements, either a principal must highly trust another principal in the system in regards to a given security requirement or else a *security mechanism* must be in place to enforce that particular security requirement.

**Table 7: Agent/Host Security Requirements w/ Abbreviations**

CP	agent code privacy	hiding the algorithm or <i>function</i> of the agent code
CI	agent code integrity	ensuring agent's code remains unaltered
CF	agent code safety	ensuring agent's code is not malicious
CA	agent code authenticity	ensuring identity of agent developer is correct
IP	agent itinerary privacy	ensuring agent itinerary is secret (other than previous or next host ID)
II	agent itinerary integrity	ensuring agent itinerary is unchanged
SI	agent state integrity	ensuring execution and resulting state of the agent is unaltered
SP	agent state privacy	keeping parts of the agent data state safe from observation
AA	agent authenticity	ensuring correct identity of agent, dispatching host, application owner
AZ	agent authorization	ensuring agent is authorized access to host resources
AN	agent non-repudiation	ensuring agent commitments are kept
AV	agent availability	ensuring host does not deny agent service or proper transmission
AY	agent anonymity	keeping the identity of the agent anonymous
HA	host authenticity	ensuring correct identity of the (dispatching, executing, or trusted) host
HN	host non-repudiation	ensuring host commitments are kept
HP	host data privacy	ensuring host input data is private
HY	host anonymity	keeping the identity of the host anonymous
HV	host availability	ensuring agent does not deny the host service
HI	host integrity	keeping server data free from unauthorized agent observation

As most current literature bears out in Chapter 2 and Appendix A.3, meeting security requirements for *mobile agents* is not as simple as just applying cryptographic primitives or introducing mechanisms. Agents complicate and extend normal security requirements for a typical network-based distributed application. Requirements for security in mobile agent applications (reviewed in Section 2.2.2) derive from the unique interactions in a mobile environment. Specifically, agents are programs with three elements: static program code, a dynamic data state, and a current execution thread. Agents execute only in context to a migration path (itinerary) among a set of hosts (servers).

We construct mobile agent applications using an underlying architecture or middleware that integrates agent mobility. Since real-world practical applications drive security requirements, we assert that not all mobile agent applications require the same security level. In many cases, a

given application’s security depends on its expected operating environment—including environments filled with adversarial relations, compromising insiders, or friendly alliances with common goals.

We provide a literature review for the proposed mobile agent security mechanisms mentioned here in Appendix A, and list several for context. Host based mechanisms protect a host from malicious agents and include sandboxing, safe interpreters, code signatures, state appraisal, proof carrying code, path histories, and policy management. Agent-based mechanisms protect the agent from outside malicious activity and several commonly referenced mechanisms include encrypted functions, detection objects, replication with voting, reference states, time-limited execution, digital signatures, phoning home, anonymous routing, trusted third parties (TTP), secure multi-party computation (SMC), multi-agent systems (MAS), intermediate data result protection, undetachable signatures, environmental key generation, execution tracing, and tamperproof hardware (TPH).

Protection mechanisms can allow agent transactions despite unknown or poor trust environments. Wilhelm *et al.* [138], for example, make a strong argument that installed trusted hardware and appropriate execution protocols can effectively shield private application data from observation on untrusted remote servers—thereby mitigating trust issues. A principal can have different trust levels for different requirements, e.g., Alice may trust Bob to execute her agent without reverse engineering it (an expression of *code privacy*), but may not trust Bob to execute the agent without looking at previous results from other hosts (an expression of *state privacy*). When the desired trust level is not adequate between parties in the agent application, parties require that security mechanisms enforce specific security requirements before allowing agent/host execution.

**Table 8: Agent Security Requirements and Related Mechanisms**

<b>SI</b>	<b>state appraisal, encrypted functions, SMC, detection objects, executing tracing, reference states, intermediate result protection, state transition verification, group hosts, TPH, environmental key generation, undetachable signatures</b>
<b>SP</b>	<b>TPH, SMC, encrypted functions, obfuscation, sliding encryption, phoning home, MAS</b>
<b>HV</b>	<b>state appraisal, path histories, sandboxing, safe interpreters, policy management</b>
<b>AV</b>	<b>time-limited black box, phoning home, cooperating agents, MAS</b>
<b>HI</b>	<b>path histories, sandboxing, safe interpreters, proof carrying code</b>
<b>CP</b>	<b>TPH, SMC, encrypted functions, obfuscation, MAS</b>
<b>CI</b>	<b>digital signatures, clone detection</b>
<b>CF</b>	<b>sandboxing, proof carrying code, state appraisal</b>
<b>IP</b>	<b>anonymous/onion routing, bidirectional dispatch</b>
<b>II</b>	<b>itinerary recording, replication and voting</b>
<b>CA,HA,HN</b>	<b>digital signatures, TTP</b>
<b>AA,AZ,AN</b>	<b>digital signatures</b>
<b>HP</b>	<b>SMC, TTP</b>
<b>AY,HY</b>	<b>TTP</b>

Both application owners and potential agent execution environments have stake in the mechanisms we use to enforce security—whether they prevent or detect malicious behavior and what requirements aspect they enforce. Certain mechanisms are *preventative* in nature, not allowing malicious behavior a priori. Other mechanisms rely on a posteriori information to *detect* whether unauthorized actions occurred to either the agent or the host. Some mechanisms readily fit into both categories and the clear delineation remains unimportant. In general, we desire preventative mechanisms over detection mechanisms when available because they are stronger, but they usually come with more overhead, limited use, or complicated implementation. We consider detection as a less stringent security method because an adversary has already violated system security in some way when the mechanism identifies the malicious activity.

No single security *mechanism* can address every security *requirement* for a mobile agent system. Application level security brings together a process for selecting security mechanisms that achieve a desired trust level within a mobile agent system. Claessens *et al.* [360] delineate

several application level combinations for requirements/mechanisms that enforce desired security properties. Even when using mechanisms that *establish* trust in untrusted environments (such as tamperproof hardware), agent applications must taken into account other assumptions in order guarantee all desired security requirements are met. Trusted hardware or multi-agent secure cryptographic protocols may be warranted or even feasible given certain application environment factors. When such mechanisms are not available or are not practical to implement, higher trust levels are necessary; we require a more policy-driven approach to make dynamic decisions about agent execution. We describe the ability to express such interactions formally next.

### 4.3 Trust Framework

Mobile agent systems deal with environments where partial knowledge and blind trust are common; therefore, subjective determinations are appropriate for incorporation into security mechanisms and agent execution decisions. We consider trust as complex because mobile agent principals possess one-way trust for different security requirements. To formalize a mobile agent application, we define first the principals that can be assigned trust properties, define next the trust relationship nature between principals, and finally formulate what trust relationships can accomplish in mobile applications settings (the trust algorithm).

#### 4.3.1 Defining Principals

Table 9 fully defines principals within our model, using extended BNF form<sup>7</sup>, and we discuss each composition individually. We find three distinct *principal* groups in mobile agent systems: agents, hosts, and entities. We define an agent as a composition that includes static software (*code*) and a set of dynamic states (*state*) representing the agent’s migratory results.

**Table 9: Principals in Mobile Agent Systems (expressed in extended BNF notation)**

```

<agent> = <code>, <state>+, <itinerary>, <id>, <log>, <policy>
<host> = <resource>+, <id>, <log>, <policy>, <host-type>
<host-type> = <dispatching> | <executing> | <trusted>
<entity-type> = <code developer> | <application owner> | <host manager>
<entity> = <organization>, <entity-type>
<principal> = <agent> | <host> | <entity>
<trust> = <level>, <foreknowledge>, <timeliness>
<application> = <principal>+ , (<principal>, <principal>, <security requirements>+ , <trust>)*

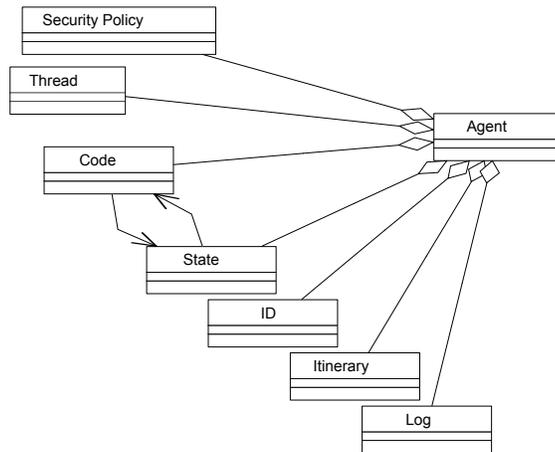
```

We describe agents by their migration path (*itinerary*), any unique identifiers (*id*), a record describing agent or host activity (*log*), and a security specification (*policy*) that includes any historical trust information for other principals in the agent application. We can create agents using reusable components where each component has its own associated dynamic state and trust level. For simplification, we define an agent to have only a single static code. The agent “id” encompasses more than one identity as well. For agent naming, we use Roth’s agent kernel (see Figure 10, p. 14) that uniquely binds a specific mobile agent’s dynamic state to its static code. This identification eliminates the possibility for cut/paste and oracle-style attacks that plague certain posed security mechanisms. The static code, the application owner, and the code developer all possess unique identities as well—which we capture in the *id* component. Figure 37 depicts the agent composition in a traditional Unified Modeling Language (UML) class notation, where we abuse the notation slightly for representational viewing.

Hosts provide an execution environment for the agent. They encompass the underlying physical hardware, runtime services, and middleware necessary for agent migration and execution. Agents see a host as a collection of computational, communicational, informational, and management resources. Hosts offer services provided by local (non-itinerant) agents, advertise host based software processes, offer host-based physical resources such as memory

<sup>7</sup> ISO/IEC 14977:1996(E), see <http://www.nist.fss.ru/hr/doc/mstd/iso/14977-96.htm>

and processor, and provide any information necessary for the agent to accomplish its task. Hosts also have security policies that support the trust formation and decision process.



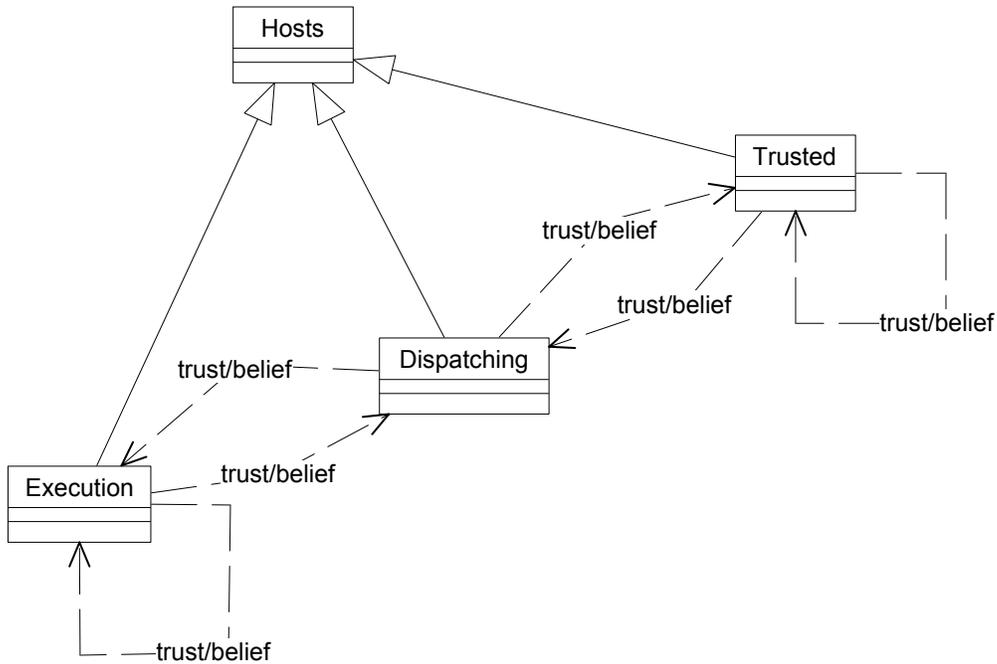
**Figure 37: Defining the Agent**

Three host types are relevant to mobile computations: the *dispatching host (DH)* associated with the application owner that launches mobile agents, the *executing host (EH)* where mobile computations occur, and *trusted hosts (TH)* which have ability to change trust relationships among other principals based on services they offer. The *DH* owns one or more agents that are acting on its behalf and remains *partially* responsible for agents under its control. For simplicity, we assume that a mobile agent application has only one dispatching host. One or more *executing hosts* comprise an agent itinerary: each host acts upon the agent code/state to produce a local host output and updated agent state. *Trusted hosts* conceptualize servers that provide security benefit for agents during their lifetime. They assume trusted third party status in many different security mechanisms—such as extended execution tracing [39] or multi-agent secure computation schemes (Chapter 3).

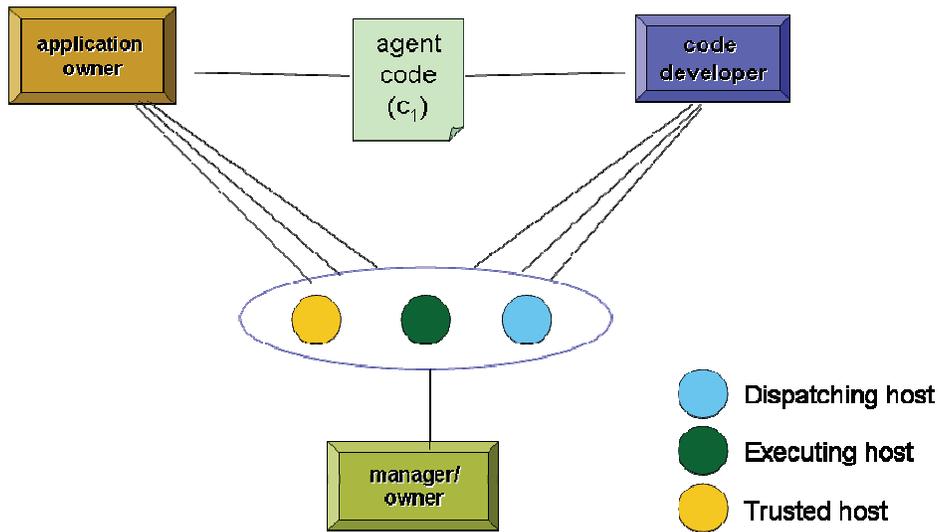
Figure 38 depicts in UML form the relationships between these three host types: each host type can have a one-way trust/belief relationship with another host. Dispatching hosts have trust/beliefs regarding executing hosts and every executing host has some trust/belief about agents received from a dispatching host. Likewise, dispatching hosts have trust/belief relationships with trusted hosts and each trusted host has a distinct relationship with every dispatching host. In every case, trust relationships do not have to be the same. Executing hosts and trusted hosts may also have trust/belief regarding other executing and trusted hosts as well.

As Figure 39 depicts, three entities have bearing on security relationships in mobile settings. We define the static agent code creator as the *code developer (CD)* and define the code user as the *application owner (AO)*. The *CD* and *AO* may be the same. The computer owner, the systems manager, and computer user can be the same person, or can be separate individuals with different trust levels. For simplicity, we view the *host owner, manager, and user* as synonymous and apply the term *host manager (-M)* to refer to all three responsible parties. In human terms, we trust machines (hosts) and software (agents) in some cases because we trust the manager associated with the environment or the developer for the software. We equate the trust that we have in the host manager (*DH-M, EH-M, TH-M*) as the trust we have in any other host (*DH, EH, TH*), realizing that the host manager for the dispatching host (*DH*), the code developer (*CD*), and the application owner (*AO*) can all be different entities or the same entity. In Appendix B, we provide Figure 140 to conceptualize in UML form the trust relations between host and agent and Figure 139 to depict the associations between the application owner, agent developer, and dispatching host with the agent.

<p><b>trust(DH) ≈ trust(DH-M)</b>  <b>trust(EH) ≈ trust(EH-M)</b>  <b>trust(TH) ≈ trust(TH-M)</b></p>
---



**Figure 38: Defining Executing/Dispatching/Trusted Hosts**



**Figure 39: Three Entities Affecting Trust in Mobile Agent Environments**

We define an *application* as the collection of all possible hosts involved in the agent task and the set of uniquely identifiable agents that implement a user function. This intuition captures single agents and multiple collaborating agents including those with the same static code and different itineraries and those with different static code. Applications, not agents, therefore become the focal point for trust determination, security requirements, and security mechanisms. We now define trust relationships shared among principals in our model.

### 4.3.2 Defining Trust Relationships

One security task is to attribute rightfully the observed actions within the system to a given party. The code developer, the dispatching host, and all visited hosts influence the agent's data state and code-making attribution for malicious behavior difficult. For simplicity, we equate the trust in the agent code with trust in the code developer and we will equate trust we have in the dispatching host as the trust we have in the application owner. We define a trust relationship  $\delta: P \rightarrow P \rightarrow S \rightarrow (L, F, M)$  as a mapping  $\delta$  between two principals ( $P_x, P_y$ ) and some number of security requirements ( $S$ ) with three associated parameters: trust level ( $L$ ), foreknowledge ( $F$ ), and timeliness ( $M$ ), defined in Table 10.

$$\text{trust}(A) \approx \text{trust}(CD)$$

$$\text{trust}(DH) \approx \text{trust}(AO)$$

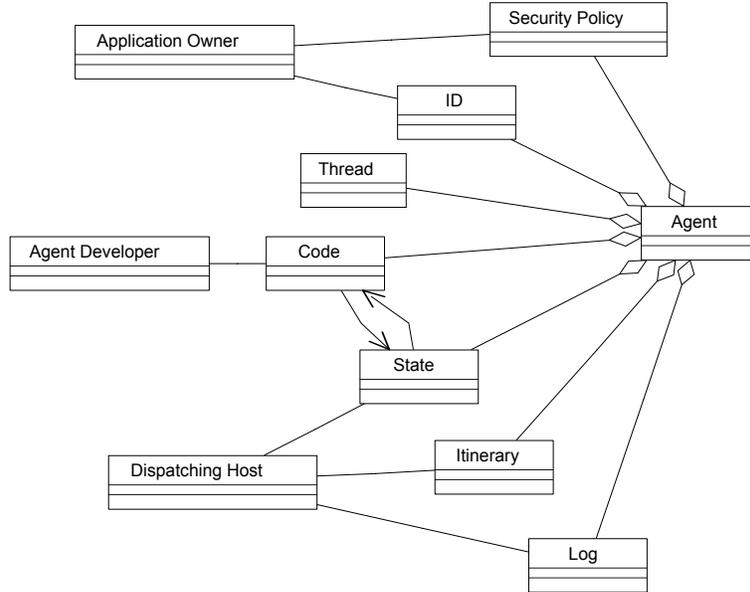


Figure 40: Principals and Entities Associated with an Agent

We categorize trust levels ( $L$ ) in a range from highly untrusted ( $HU/U$ ) to highly trusted ( $HT/T$ ), where in some instances we assume trust is not determined ( $ND$ ). Trust levels are non-Boolean and reflect a one-way subjective belief that one party will behave towards another party at some perceived malicious intent level ( $HU, U, ND, T, HT$ ). Trust can be discretely categorized negatively and positively as ranges between  $[-1, 1]$ : ( $HT, T$ )  $> 0$ ,  $ND = 0$ , ( $U, HU$ )  $< 0$ . Our model considers levels in the range  $[0, 1]$ .

Foreknowledge ( $F$ ) defines prior interaction between principals. Agents traveling in a dynamic free-roaming itinerary can encounter unknown hosts. Likewise, hosts are likely to encounter agents with no prior experience. We describe the foreknowledge held by a principal as either well-known ( $WK$ ), known ( $K$ ), or unknown ( $UK$ ). *Well-known* principals have established histories while we identify *known* principals for possible agent execution/migration.

Timeliness ( $M$ ) characterizes information currency and we express it with values *expired*, *stale*, or *fresh*. We establish timeliness by mechanisms such as timestamps [366] and we use age comparisons to determine whether we can rely on recommended or delegated trust decisions. Given the same interaction volume, trust is higher when the interaction period is longer. When the elapsed time since the last interaction is short, we can place higher confidence in interactions that are more recent.

For a specific mobile agent application, we derive principals using all sets for possible concerned parties. Based on simplifying assumptions, which we depict in Figure 41, we define four possible principal sets: dispatching host/application owner ( $DH/AO$ ), all executing hosts ( $EH$ ), all trusted hosts ( $TH$ ), and all agents/code developers ( $A/CD$ ). If a more precise trust

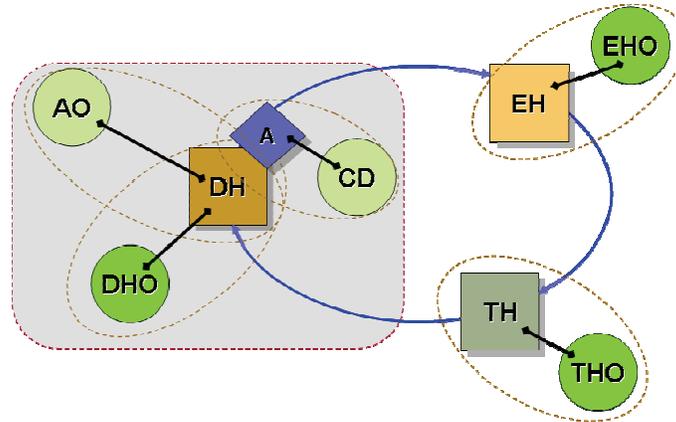
relationship needs expression for these specific entities, we can treat code developers and application owners separately.

$$\langle \text{application} \rangle = \langle \text{principal} \rangle^+ , (\langle \text{principal} \rangle, \langle \text{principal} \rangle, \langle \text{security requirements} \rangle^+, \langle \text{trust} \rangle)^*$$

Trust level, foreknowledge, and timeliness bind trust from two *principals* (an application owner, an executing host, a dispatching host, an agent, etc.) with one or more security requirements (elaborated in Table 7). Though we represent foreknowledge, trust level, and timeliness discretely, they can be converted to continuous ranges  $[-1, 1]$  or  $[0, 1]$  for example) to accommodate different trust algorithms.

**Table 10: Trust Relationships**

$\langle \delta \rangle =$	$\langle P_x \rangle, \langle P_y \rangle, \langle S \rangle^+, \langle L \rangle, \langle F \rangle, \langle M \rangle$
$\langle L \rangle =$	$\langle \text{highly untrusted} \rangle \mid \langle \text{untrusted} \rangle \mid \langle \text{non-determined} \rangle \mid \langle \text{trusted} \rangle \mid \langle \text{highly trusted} \rangle$
$\langle F \rangle =$	$\langle \text{well known} \rangle \mid \langle \text{known} \rangle \mid \langle \text{unknown} \rangle$
$\langle M \rangle =$	$\langle \text{expired} \rangle \mid \langle \text{stale} \rangle \mid \langle \text{fresh} \rangle$
$\langle S \rangle =$	$\langle \text{agent code privacy} \rangle \mid \langle \text{agent code integrity} \rangle \mid \langle \text{agent code safety} \rangle \mid \langle \text{agent code authenticity} \rangle \mid \langle \text{agent itinerary privacy} \rangle \mid \langle \text{agent itinerary integrity} \rangle \mid \langle \text{agent state integrity} \rangle \mid \langle \text{agent state privacy} \rangle \mid \langle \text{agent authenticity} \rangle \mid \langle \text{agent authorization} \rangle \mid \langle \text{agent non-repudiation} \rangle \mid \langle \text{agent availability} \rangle \mid \langle \text{agent anonymity} \rangle \mid \langle \text{host authenticity} \rangle \mid \langle \text{host non-repudiation} \rangle \mid \langle \text{host data privacy} \rangle \mid \langle \text{host anonymity} \rangle \mid \langle \text{host availability} \rangle \mid \langle \text{host integrity} \rangle$



**Figure 41: Simplifying Trust Assumptions in Mobile Agent Application**

## 4.4 Trust Algorithm

We now discuss how our model supports trust-based decision making in mobile agent contexts. Given an application  $G$  with associated hosts set  $H_x$ , a set containing uniquely identifiable agents  $A_y$ , and a set with trust relationships  $T_{ij}$ , we can formulate the trust actions and decisions that are possible in the application space based on trust relationships among all principals. Appendix B provides elaboration of various agent scenarios using the trust model.

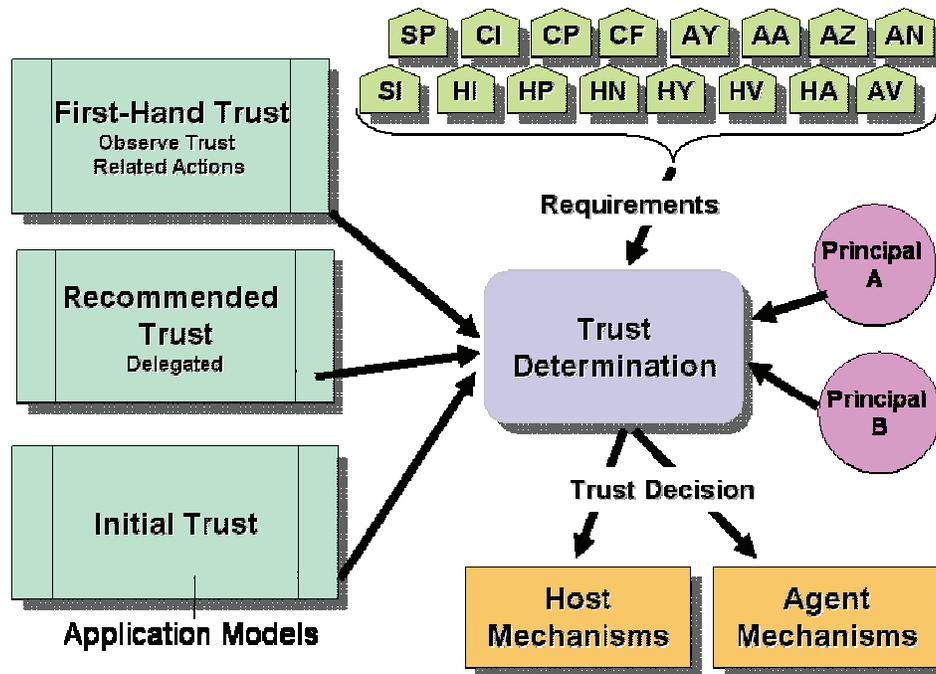
### 4.4.1 Trust Decisions

Trust decisions in pervasive scenarios come from two primary information sources: personal observations (previous interactions) and recommendations from other parties (transitive or

delegated trust). We can use Spy agents [153] to build and maintain a trust profile by validating behavior in small interactions. Trust-earning actions build relationships and principals use these to determine which security mechanisms will meet the application owner’s desired security level.

Trust in the mobile agent environment affects *security mechanism selection*, *agent itinerary*, *policy decisions*, and *code distribution*. For example, if the application owner (AO) has non-determined (ND) or low (U) trust toward any prospective host, the owner may require detection mechanisms to guarantee agent state integrity or agent state privacy. If no trust (HU/U) exists at all, the AO may require stringent mechanisms that prevent agent state integrity/privacy. If the AO has full trust (T/HT) in prospective hosts, no security mechanism may be required to allow agent migration and execution.

To link agent security mechanisms with application requirements, our framework processes *initial*, *recommended*, and *first-hand* trust to render a mechanism-based decision that meets the security objectives for involved principals. Highly trusted and trusted principals will tend to yield no requirement for security mechanisms. Non-determined trust will tend to require detection-oriented mechanisms while untrusted relationships will tend to demand prevention-oriented mechanisms. Migration decisions are also determined based on trust level.



**Figure 42: Trust Decisions for Mobile Agent Security**

Figure 42 depicts the inputs and outputs to our trust determination process for mobile applications with the outcome selecting one or more mechanisms that will meet bidirectional requirements between principals defined by the trust relationships. The trust algorithm produces an appropriate host-based mechanism set and an appropriate agent-based mechanism set. We generalize the initial trust set based on the agent’s application environment. We define the *initial trust component* for the trust-based decision seen in Figure 42 by a relationships set (*the application model*) which is context-dependent on the application. We define three different application model scenarios in Section 4.5.

Several decisions are possible based on principal interactions within the mobile agent application. We summarize the interactions that are possible in the agent lifecycle in Appendix B, Table 29, but discuss two in detail here: *agent dispatch* and *agent migration*. Figure 43 highlights the trust relationships required for agent dispatch while Figure 44 highlights trust and security for

agent migration. For simplicity, we consider the *application owner (AO) / dispatching host (DH)* synonymous and consider the *code developer (CD) / agent (A)* synonymous in trust expectation.

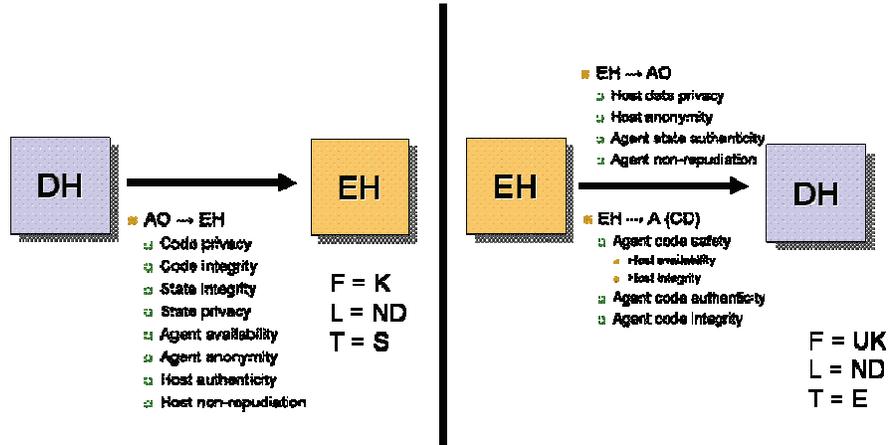


Figure 43: Trust/Security Decisions on Agent Dispatch

The first *executing host (EH)* in the agent itinerary has trust expectations (possibly different) from those expressed by the application owner. Recalling the definition for a trust tuple,  $\delta: P \rightarrow P \rightarrow S \rightarrow (L, F, M)$ , there exists tuple forms such as  $(AO, EH, S_{EH}, (L, F, M))$ ,  $(EH, AO, S_{AO}, (L, F, M))$ , and  $(EH, DH, S_A, (L, F, M))$  in a fully populated trust database. The set of security requirements  $S_{EH}$  that an application owner (AO) wishes to enforce for any prospective executing host (EH) may include code privacy, code integrity, state integrity, state privacy, agent availability, agent anonymity, host authenticity, and host non-repudiation. The set of security requirements  $S_{AO}$  an executing host (EH) may specify towards an application owner (AO) include host data privacy, host anonymity, agent state authenticity, and agent non-repudiation. The set of security requirements  $S_A$  an executing host (EH) may specify towards an agent/code developer (A/CD) may include agent code safety (to ensure host availability and host integrity), agent code authenticity, and agent code integrity.

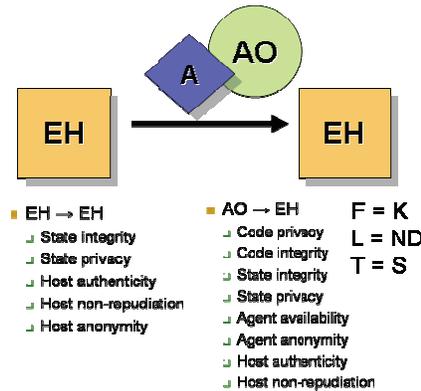


Figure 44: Trust/Security Decisions on Agent Migration

A single tuple exists in a policy database for every security requirement desired, each mapping to an allowable foreknowledge level (F), trust level (L), and timeliness level (T). Figure 43 for example indicates that an application owner can specify security requirements for executing hosts who are known (K) to have non-determined trust (ND) with stale (S) timeliness on their information. Executing hosts, likewise, specify security requirements for application owners who are unknown (UK) with non-determined trust (ND) and with expired (E) timeliness on their information. Figure 44 depicts the trust relationships that our trust framework evaluates during agent migration. Here, principals include two executing hosts (EH), the agent (A), and the

application owner (AO). Because a previous host may corrupt the agent, state integrity and privacy become paramount.

Required security mechanisms (*none, detection, prevention*) for a particular principal are determined based on trust level. For example, if an unknown agent requests migration to a host that requires agent code integrity and no mechanism enforces that requirement, the executing host refuses agent migration according to the policy database. If a well-known agent requests migration and no tuples specify policy otherwise, the executing host allows the migration and execution. Trust exists as a unidirectional property: what one principle (an executing host) considers allowable for security mechanisms may not be adequate for the other principal (application owner).

Given a trust relationship set that exists between principals, a policy engine using this framework can make several trust-based decisions:

- which agent/host security mechanism to use;
- which hosts an agent can include in the itinerary;
- which agent code parts are executed by the host;
- which agents are allowed host resources;
- which principals can share policy information;
- whether trust levels can increase (whether you are allowed to recover from negative trust);
- whether or not trust recommendations from other parties should be given merit to include how much weight they are given.

Figure 45 pictorially summarizes our trust framework components for mobile agent applications using standard UML generalization and compositional notation.

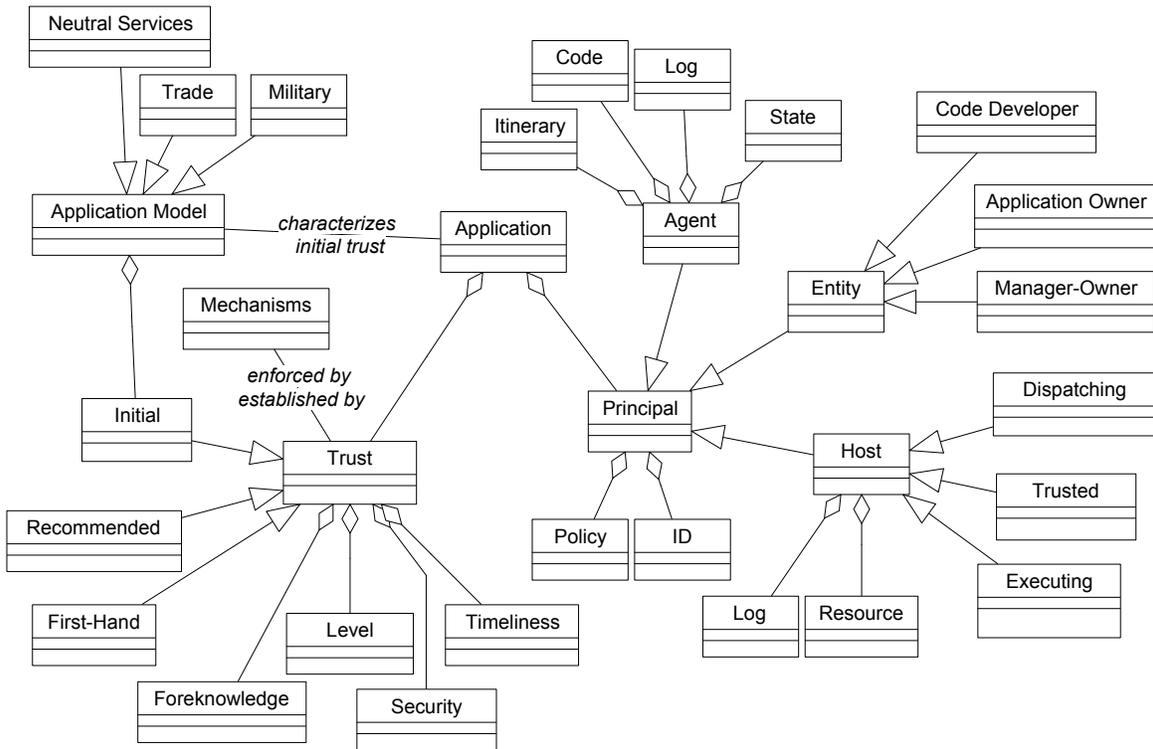


Figure 45: Trust Framework

#### 4.4.2 Trust Acquisition

We link trust relationships to one or more security mechanisms in our model. Given principal, trust, and application definitions, we can exercise security decisions based on requirements. We enforce trust using security mechanisms; applications link the principal's trust expectations through security requirements to a trust level, foreknowledge, and timeliness.

To formulate trust, our model supports three different acquisition modes: *initial trust*, *first-hand trust*, and *recommended trust*. Initial trust is the relationship set belonging to an agent or host before interaction history takes place over time. We argue that such an initial trust relationship set can be generalized based on the agent's application environment and pose at least three such models. Next, principals gather first-hand trust over time through specific interactions with other principals. We gain recommended trust when we accept or rely on trust levels offered by other principals. When the initial binding of trust at various stages of the mobile agent, we define the following lifecycle points for trust expression:

- (1) **creation** and **development** of code bind trust to a *code developer*
- (2) **ownership** of an agent binds trust to an *application owner*
- (3) **dispatching** an agent binds trust to a *dispatching host*
- (4) **execution** of an agent binds trust to all prior *hosts* an agent has visited plus its *dispatcher*
- (5) **migration** binds trust to the next *host* in the agent itinerary
- (6) **termination** binds trust of the entire *application* to the entire set of *execution hosts* and the network environment

Our model allows a principal to earn trust or degrade trust based on actions observed over time. Figure 46 illustrates the trust cycle where an agent execution using one or more executing hosts affects trust among all principals in a policy database. Observable trust-related actions during execution can change trust levels among mobile agents and hosts. Trust relationships evolve from initial trust according to predefined rules—which represent a security policy. In previous work on trust in ad-hoc networks [38], four different trust acquisition categories are formulated which we apply in the mobile application context: trust-earning actions over time, trust-earning actions by count, trust-earning actions by magnitude, and trust-defeating actions.

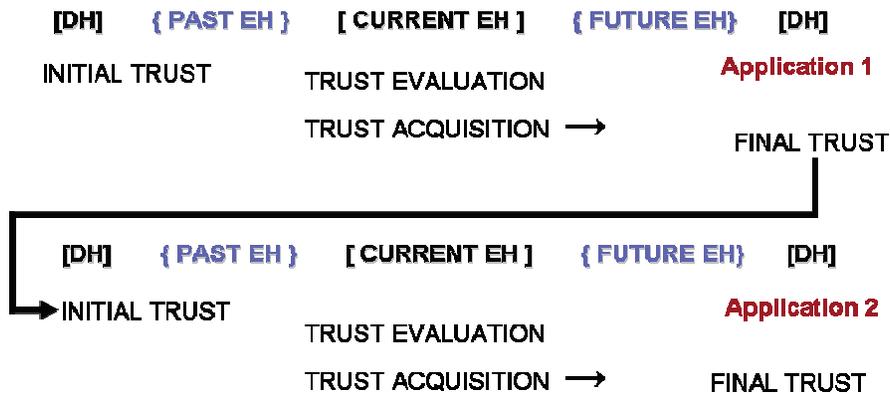


Figure 46: Acquired Trust over Multiple Applications

Because many methods already exist for evaluating delegated and acquired trust in mobile agent systems [39, 40, 41, 42, 152], we leave identification of specific delegated algorithms open for implementation. We focus instead on the novel aspects found in our approach—which include representation of mechanisms/requirements and how to express the role of trusted hosts.

#### 4.4.3 The Role of Trusted Hosts

*Trusted hosts* (TH) are distinguished from *dispatching* (DH) or *executing* hosts (EH) in an agent application and we pose a novel expression for their role. Trusted hosts conceptualize

properties normally associated with trusted third parties (TTP) in various agent security mechanisms and have specialized, static, and pre-determined trust relationships with principals. TTP trust levels do not change with agents or hosts that interact with them, though we describe the concept of *partially* trusted third party in Chapter 3. If TTPs do not have full trust in an application environment, we represent them as an *executing host (EH)* with normal trust relationships.

*Execution hosts* in the agent itinerary that have a trust level equal to “highly trusted” or “trusted” can be used to detect malicious behavior such as verifying intermediate data integrity. A *trusted host*, on the other hand provides, a third-party service such as information lookup, mediation, brokering, communication service, or middle-agent hosting. We are not concerned with whether an agent communicates statically or migrates dynamically to the trusted host. We capture the primary intuition that trusted hosts provide a means for either increasing or decreasing trust levels for other principals—with all parties in the application giving confidence and agreement for them to do so.

For example, in extended execution tracing (EET), the trusted server verifies agent execution integrity for both executing hosts and the dispatching host [33, 39]. Trusted servers in EET facilitate the migration process and become the only means by which agents can move from one executing host to another. When a host violates agent integrity, the trust policy framework lowers the trust level for that executing host and communicates the violation to other principals in the system. A host can delegate trust to another via the trusted server chain and a trust acquisition methodology.

High trust levels in mobile applications derive from several possibilities: having tamperproof hardware (TPH) installed, having a good reputation, being under the same management domain, and having an established trusted/non-malicious interaction history. An application owner, for example, may trust *highly* an executing host in its own management domain that has TPH such as a smart card reader. Yee [74] points out that we routinely use TPH to offset trust levels when remote hosts have non-determined trust or are assumed untrusted. Host-installed TPH that supports agent execution can allow the application owner to assign a *trusted* or *highly trusted* status not possible otherwise.

*Trusted hosts* normally implement a particular security mechanism, such as in EET or multi-agent secure computation. The trusted host can therefore notionally increase or decrease trust among principals in an application based on their services. TTPs may affect trust level relationships between host and agent, agent and host, host and host, or agent and agent. When trusted hosts service an agent via migration, they can inspect the agent or otherwise interact with the agent like any other host. When agents interact with the TTP by static communication, they can pass information to the trusted host for data logging, result protection, integrity checks, or phoning information back to their dispatching host.

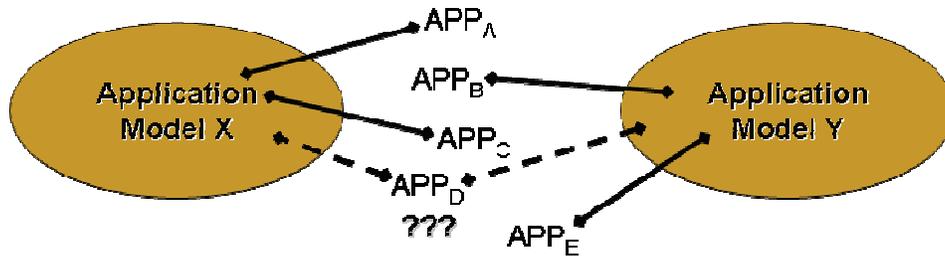
Trusted hosts introduced in a mobile application can thus change trust relations among principals and enforce particular security requirements for executing hosts and application owners. When agents and hosts interact with the trusted host, TTPs adjust trust mappings based on particular security requirements and their interaction. Services provided by trusted hosts can alter an agent’s state, itinerary, or security policy. An agent, for example, may use a trusted host to determine the next host to visit and alter its itinerary based on the interaction. No matter which principals are involved in the transaction, we assume trusted hosts to act in the best interest for *both* agents and hosts and therefore achieve guaranteed application level security goals for all parties involved.

An application environment *model* generalizes the adversarial nature that exists among principals. The next section gives our novel concept for such models in defining initial mobile agent security relationships.

## 4.5 Application Security Models

Models come in many shapes and sizes. In all cases, we use them to focus and detail a particular problem aspect. Security models help test whether security policies are complete or

can verify whether an implementation fulfills a requirements set. Application models for multiple agent systems describe how agents accomplish tasks based on an underlying pattern such as publish/subscribe, courier/broker, and supervisor/worker. The application context determines security responsibilities for principals and limits trust award to occurring only through specific interactions. As we illustrate in Figure 47, applications ( $APP_A$ ,  $APP_B$ ,  $APP_C$ ) typically have one application model that describes the behavioral aspects of parties within their particular environment. In some cases, we can use applications ( $APP_D$ ) in more than one application model setting by adjusting security requirements, mechanisms, or trust assumptions about parties within the environment.



**Figure 47: Application Security Models**

In our security framework, we establish trust three ways: initial trust, acquired trust, and recommended trust (see Figure 42 and Figure 45). Over time, trust will change based on observed actions and delegated decisions. Every application has unique security requirements, but many applications share a common trust environment that can be the starting point for trust-enhanced security decisions. Application scenarios dictate how we derive principals and how they act towards each other. We define scenarios to set boundaries on whether we can acquire trust over time—whether we can promote principals from untrusted ( $U$ ) to non-determined ( $ND$ ) or from non-determined ( $ND$ ) to trusted ( $T$ ).

We leverage the notion that initial trust relationships exist in a mobile agent application (between agents and hosts) based on a common trust environment. This initial trust is the starting point for trust-enhanced security decisions and we define the set of initial trust relations as an application security model. We provide three real world environments that reflect mobile agent applications and that share common trust assumptions: the *military model*, the *trade model*, and the *neutral services model*. These initial trust relationships couple the security requirements and trust levels from various participants. As a result, agents in an application can initially determine which security mechanisms they are willing to support and hosts can initially specify their required security mechanisms.

#### 4.5.1 The Military Model

We base the military model on the notion that a wall or “Maginot Line” exists between friendly and adversarial entities. Within friendly borders, entities typically know each other as authenticated, highly-trusted principals. At some point, however, an adversary may take a given principal captive. This captured entity (whether a host or agent) may continue to function passively in order to discover information or leak secrets on the capturer’s behalf. Captured entities may become overtly malicious by delaying and denying service, corrupting friendly communications, and attacking privacy and group integrity operations.

The military model formulates common application characteristics between principals and helps focus security requirements. For instance, although hosts might be ad-hoc or mobile, a managerial entity verifies their identity within the environment. Figure 48 illustrates how initial trust relationships capture this notion: every dispatching host/agent originator ( $DH/AO$ ) has a “known” and “trusted” relationship with every executing host ( $EH$ ) to begin with. We indicate this by “ $K/T$ ” in the row for  $DH/AO$  that intersects the  $EH$  column in Figure 48. In the military model, trust

relationships are implicit as long as the management domain can verify the principal's identity. We commonly know all principals beforehand in the military model, so we grant more trust to authenticated agents and hosts as a result.

	DH AO	EH	TH	A CD
DH AO	k / HT	k / T	k / HT	k / T
EH	k / T	k / T	k / HT	k / T
TH	k / HT	k / HT	k / HT	k / HT
A CD	k / T	k / T	k / HT	k / T

**k = Known**  
**uk = Unknown**

**HT = Highly trusted**  
**T = Trusted**  
**ND = Non-determined**  
**U = Untrusted**  
**HU = Highly untrusted**

Given military model application  $G$  with principal set  $P$ , dispatching host set  $DH$ , execution host set  $EH$ , trusted host set  $TH$ , application owner set  $AO$ , code developer set  $CD$ , and trust relation set  $v$ :

1.  $DH \subseteq EH$
2.  $TH \neq \emptyset$
3.  $\forall p_i, p_j \in P: i \neq j$  and  $\forall \delta (p_i, p_j, s) \in v$ :  
 $F = \langle \text{known} \rangle$ , or  
 $F = \langle \text{well known} \rangle$

**Figure 48: Military Model Initial Trust Relationships**

The military model fits requirements and trust relationships where using trusted third parties, trusted hardware, group security operations, multiple security levels, multiple trust levels, and distinct organizational structures exist. We find this environment in many corporate infrastructures (as well as the military itself) where a trusted computing base is financially possible or mandated. Implicit trust among principals allows hosts to work efficiently in cooperation with agents to provide mutual prevention and detection services.

The military model also suggests common agent characteristics exist within an environment where a centralized authority designs, develops, and deploys agents. In industry, corporations may delegate development to outsourced teams or an information technology department with in-house programmers. The military model reflects programming environments where only authorized mobile agent applications are used and agents act as peers. Other initial trust models can reflect agents that take on adversarial roles with one another. Even corporate divisions have proprietary and sensitive information that may require protection.

In the military model, agents may still have requirements for integrity and privacy, but we can verify their identity, safety, authorization, and authentication within a circle of trust. The military model also places less emphasis on distinction between executing and dispatching hosts. Agent servers play the role interchangeably in some cases as the dispatcher and in other cases as the

execution host. Initial trust in this model reflects many real-world computing paradigms where agent-based applications accomplish group collaboration, systems management, and information gathering. Centralized management domains exist and form a key feature in the military trust model.

Figure 48 summarizes the military model initial trust relationships and describes how we initialize a set of trust relations  $v$ . We illustrate two simplifying assumptions: the application owner (AO) and the dispatching host (DH) are equivalent for trust purposes and the code developer (CD) has equivalent trust to the agent (A). The matrix also depicts, for example, the dispatching host (DH) / application owner (AO) initially knows and trusts (K/T) all executing hosts (EH). It also illustrates how the AO knows and trusts all agents (A/CD) it will use.

Based on this initial trust relationship set, the trust algorithm dynamically determines acquired or recommended trust. Acquired trust mechanisms (where we define negative trust) facilitate discovery of infiltrators. These relationships also determine the security mechanisms required by each host or agent. In the military model, we assume that some agents or hosts will eventually fall under “enemy” control. Two primary security-related tasks consume the majority of time in the military model: 1) protecting insiders from outsiders and 2) detecting whether or not an adversary has compromised or captured an agent or host.

The latter security task becomes detecting anomalous or malicious behavior and removing malicious parties from the circle of trust. This scenario best represents application environments with a peer (non-adversarial) relationship among agents and hosts. As Figure 48 illustrates, the initial trust relationships among all principals in the system begin at a known and trusted level and when trusted servers are used (TH), they are “highly trusted” (HT).

Trusted third-party and trusted hardware roles, as well as coalition security mechanisms, focus their attention on identifying principals that have violated the circle of trust or are attempting to gain access to the circle of trust. A **strong** military model may require that all executing hosts be equipped with tamperproof hardware. Other application scenarios are better suited for expressing e-commerce interactions, discussed next.

#### 4.5.2 The Trade Model

A second model we define is the trade model: it captures the intuition for a competitive interaction among actors that are all bargaining for resources. Such an environment could also be termed an economic model, a buy/sell model, or a supply/demand model where we consider economic benefit as the chief motivator. This application scenario represents the Internet computing model where prospective buyers deploy E-commerce mobile agents. It describes applications where disjoint communities of mobile agent dispatchers want to use services or obtain goods from a set of host commodity or service providers. Agent literature routinely represents such a model as an agent dispatched to find an airline ticket among a group of airline reservation servers. The agent accomplishes the transaction autonomously while finding the best price within user constraints.

Figure 49 illustrates the initial trust relationships for security requirements in the trade model and depicts the adversarial relationship among principals. In this scenario, we express several trust facets: 1) buyers (application owners) do not trust sellers (hosts) to deal honestly with them; 2) sellers do not trust other sellers to work for their best interest; 3) buyers do not trust sellers to act non-maliciously; and 4) buyers are in competitive relationships with other buyers for the same goods and services. Initial relationships between dispatching hosts/application owners (DH/AO) and executing hosts (EH) thus have an implicit untrusted (U) relationship for parties that are known (K) and an implicit highly untrusted (HU) relationship for parties that are unknown (UK)-seen in the Figure 49 matrix. Executing hosts in the matrix (EH) have untrusted relationships (U/HU) with other executing hosts, whether known (K) or unknown (UK). We express the buyer's adversarial relationship by defining the initial trust between agents/code developers (A/CD) as non-determined (ND) or highly untrusted (UH) in the case of known/unknown parties.

The largest possibilities for perceived mobile agent applications typically fall into the trade model when describing security requirements. In this context, we do not necessarily know principals before interaction takes place. In most cases, no trust or foreknowledge exists between

users that want to execute agents and hosts that would like to execute agents. This model relies more on acquired or delegated trust decisions and reflects that executing hosts are as equally distrusting with agents as they are with other executing hosts. Application owners see hosts as implicitly untrusted in the sense that they can gain economic benefit if hosts alter agent execution integrity or maliciously collude together.

	DH AO	EH	TH	A CD
DH AO	k / HT uk / U	k / U uk / HU	k / T uk / ND	k / ND uk / U
EH	k / U uk / HU	k / U uk / HU	k / T uk / ND	k / U uk / HU
TH	k / ND uk / U	k / U uk / HU	k / T uk / ND	k / ND uk / U
A CD	k / T uk / ND	k / U uk / HU	k / T uk / ND	k / ND uk / HU

**k = Known**  
**uk = Unknown**

**HT = Highly trusted**  
**T = Trusted**  
**ND = Non-determined**  
**U = Untrusted**  
**HU = Highly untrusted**

Given trade model application  $G$  with principal set  $P$ , dispatching host set  $DH$ , execution host set  $EH$ , trusted host set  $TH$ , application owner set  $AO$ , code developer set  $CD$ , and trust relation set  $v$ :

1.  $DH \cap EH = \emptyset$
2.  $TH \neq \emptyset$
3.  $\forall p_i, p_j \in P: i \neq j \text{ and } \forall \delta (p_i, p_j, s) \in v:$   
 $F = \langle \text{unknown} \rangle$ , or  
 $F = \langle \text{known} \rangle$ , or  
 $F = \langle \text{well known} \rangle$

**Figure 49: Trade Model Initial Trust Relationships**

### 4.5.3 The Neutral Services Model

As a third notion to capture application-level security requirements, we define the neutral services model with the intuition that one or more agents acquire a service (or set of information) from providing hosts. Service providers do not themselves have an adversarial relationship with each other, but we view them as having disjoint trust communities. The primary difference in the neutral services model and the trade model is that host communities exist with no adversarial relationship among themselves. These communities are essentially neutral regarding their commitments to each other—neither friendly nor hostile. Figure 50 gives the initial trust relations for this model.

This model fits application environments designed around information or database services. Information providers typically have no economic gain from altering the results or influencing the itinerary for agents that they service. Hosts provide services honestly in the sense that they would not alter the path or intermediate data results for an agent or induce service denial. Service providers can and in most cases do charge a small fee for using their service, however. A

dispatching application owner in this model may be concerned with whether a host bills it correctly after agent interaction. In this respect, if information providers charge for their service, it is to their benefit to alter an agent's execution integrity and illegally charge an agent for more than was legitimately received.

Adversarial relationships exist between agents from the "client" community and hosts in the "server" community, but trusts within the same community do not necessarily trust or distrust towards each other (they tend to be neutral to one another). Neutral hosts see no benefit from altering an agent that might be carrying results from other hosts or from preventing them from visiting other hosts. Hosts in this realm are in essence a "one-of-many" information provider. This paradigm may not fit a search engine model where a mobile agent visits and collates search results from engines such as Google, Yahoo, and Alta Vista. In this case, one of these engines (who get benefit from every agent hit since advertisers might pay more for a more frequently visited search engine) may have desire to alter the itinerary or search results from agents that visit other hosts. It might also benefit a search engine in this example to maliciously alter search results from other engines carried by the agent and make them "less useful"; as a result, the malicious engine looks "better" to the application owner by making their competitor look "worse". For cases like this, the trade model would fit better to describe initial security requirements among principals.

	DH AO	EH	TH	A CD
DH AO	k / HT uk / U	k / ND uk / U	k / T uk / T	k / T uk / ND
EH	k / ND uk / U	k / ND uk / ND	k / T uk / T	k / T uk / ND
TH	k / ND uk / ND	k / ND uk / ND	k / T uk / T	k / ND uk / ND
A CD	k / ND uk / ND	k / U uk / HU	k / T uk / T	k / T uk / ND

**k = Known**  
**uk = Unknown**

**HT = Highly trusted**  
**T = Trusted**  
**ND = Non-determined**  
**U = Untrusted**  
**HU = Highly untrusted**

Given neutral services model application G with principal set P, dispatching host set DH, execution host set EH, trusted host set TH, application owner set AO, code developer set CD, and trust relation set v:

1.  $DH \cap EH = \emptyset$
2.  $TH \neq \emptyset$  or  $TH = \emptyset$
3.  $\forall p_i, p_j \in P: i \neq j$  and  $\forall \delta (p_i, p_j, s) \in v:$   
 $F = \langle \text{unknown} \rangle$ , or  
 $F = \langle \text{known} \rangle$ , or  
 $F = \langle \text{well known} \rangle$

**Figure 50: Neutral Services Model Initial Trust Relationships**

The protection required for applications falling under the neutral services model revolves primarily around the agent's execution integrity. To that effect, hosts that bill customers for usage might be tempted to cheat and wrongly charge agents for resources they did not use. Likewise, agents may want to convince a host falsely that it did not provide a service or information, when in fact it did. Trusted relationships between neutral third parties are also more conducive in this environment and trusted third parties may interact with various communities to provide services themselves.

## 4.6 Chapter Summary

When we consider application development, we desire methods that help transform requirements into implementation. We present in this chapter a trust model for mobile agent application development that supports trust-enhanced security decisions. We implement at least three novel concepts in our framework: we unify security requirements with security mechanisms, we address initial trust requirements at an application-specific level, and we define the relationships for trusted third parties.

Our formalized model remains more robust and comprehensive than current trust models for mobility in defining principals and their possible trust relationships. We also give the first definition for an application security model that seeds a trust framework. We give three model examples and characterize how to generate initial trust relationships based on the trust assumptions between parties involved in mobile agent interactions. Both developers and researchers benefit from this model because they can reason about security requirements and mechanisms from an application level perspective; the model allows them to integrate trust-based decisions into the mobile agent security architecture and define allowable security mechanisms. In the next chapter, we deal specifically with *mechanisms* that address the hardest problems in agent protection: how to protect the agent from meaningful alteration at the remote execution site.

## CHAPTER 5

### PROGRAM ENCRYPTION

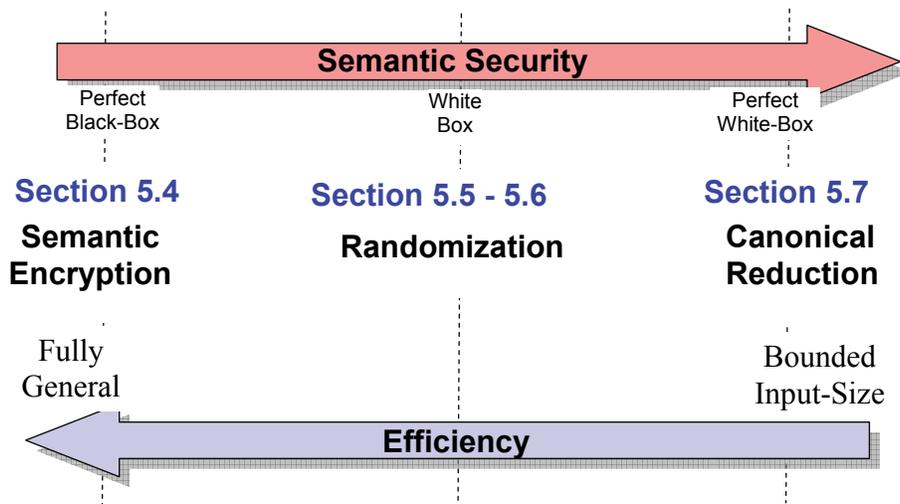
This chapter contains material from a collection of published papers [44, 154, 155, 156] and manuscripts under review [157, 158]. We first give a chapter overview and motivate the question for why we want to “executably encrypt” a program afterward.

#### 5.1 Chapter Overview

In their seminal work on homomorphic encryption in mobile agent settings [25], Sander and Tschudin challenge the idea that a program running on a remote host requires trusted hardware in order to guarantee integrity or privacy. They ask three specific questions that provide context for our thesis:

- (1) *Can a mobile agent protect itself against tampering by a malicious host? (code and execution integrity)*
- (2) *Can a mobile agent conceal the program it wants to have executed? (code privacy)*
- (3) *Can a mobile agent remotely sign a document without disclosing the user’s private key? (computing with secrets in public)*

We present results in this chapter that make positive contributions towards answering these questions affirmatively. In Figure 51, we summarize these significant results related to protecting software (generally) and protecting mobile agents (specifically) and show their relationship to both efficiency and perfect semantic security.



**Figure 51: Results in Program Encryption**

Section 5.2 gives the motivational basis for several program classes that benefit from executable encryption. Section 5.3 defines specifically what we mean by program encryption, its relationship to obfuscation, and gives definition for several program protection related metrics. Section 5.4 presents our results for achieving perfectly secure black box program protection. Section 5.5 defines a novel method for measuring obfuscation security strength based on random programs. Section 5.6 presents our results for defining randomizing obfuscators. Section 5.7

presents our methodology for achieving perfectly secure white-box protection for bounded-input size programs.

As a foundational result, we demonstrate that you can only securely obfuscate a program by first translating the program's semantic meaning (input/output relationships) into a one-way function (assuming the program is not a one-way function itself). We discuss this approach for black box protection in Section 5.4 and give a provable methodology for semantic encryption transformation. Given this translation basis (which securely hides the original program's input/output mappings) and given an associated recovery process (which reproduces the original program's intended output), we then consider how to hide the white-box information associated with a circuit's gate structure or a program's source code. In Section 5.5, we give an alternative obfuscation security model that finds applicability with (practical) obfuscation techniques currently in use today. In Section 5.6, we present our results for semantic security based on randomization techniques similar to those found in traditional data ciphers, representing a (more) efficient general approach to white-box protection. White-box protection can also be perfectly secure and generalized for programs with bounded input-size, and we give our methodology for this using canonical circuit reduction in Section 5.7.

## 5.2 Motivating the Question

We believe that the future distributed computing success depends upon securing intellectual rights found in software (in general) and protecting mobile program integrity and privacy (in particular). The malicious host problem (Section A.2) in mobile agent settings provides an interesting case for analyzing what is possible with program protection. In particular, a remote host has complete control over any code that it executes—creating a scenario where malicious parties may alter a program's execution without detection.

Methods for preserving *code privacy* in such environments have included multi-party secure computation (Chapter 3 and Section A.5), computing with encrypted functions (Section A.3.20 and Section A.3.21), homomorphic encryption [25, 159, 291], tamperproof hardware (Section A.3.19), server-side execution, time-limited black boxes [34], tamper resistance/tamperproofing [160, 161, 162, 163, 164, 165], software watermarking [166, 167], and obfuscation [168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186]. Many researchers *seek* to define obfuscation in cryptographically strong ways that show provable security according to some adversarial model or based on specific heuristic techniques. We seek methods for how to “*executably encrypt*” a program and show positive results in this thesis toward that goal. We borrow the term *program encryption* from traditional cryptography and define its properties in the next section. We distinguish *obfuscation* from *program encryption* based on provable cryptographic properties associated with the latter.

The goal of program *obfuscation* is to disguise programs so that a user can execute them but cannot determine their intent. It essentially entails constructing programs so that they are unrecognizable in a definable way. When an adversary cannot discern what a program is trying to accomplish, the adversary gains no benefit by copying the program or attempting to change the program in any *meaningful* way. Two questions naturally arise:

- (1) *How can running an unrecognizable program provide anything valuable?*
- (2) *Why would anyone risk running a program they do not understand?*

Consider a military application where the enemy captures a hand held device. If an adversary captures a device with an open session, they can observe input and output relationships. If we protect the program against black box analysis, the enemy cannot determine the device's function from an arbitrary number of input-output pairs. However, if we assume a sophisticated adversary, they may be able to analyze the device from a white box perspective; that is, they can execute arbitrary input and analyze the control flow and data manipulations in the program code as they occur. If we protect a program from white box analysis, we prevent the enemy from learning the program's intent by watching its execution or analyzing its code. Finally, in the military

environment, trusted parties would typically configure and load software to the handheld devices, using trusted hardware, so we have little concern regarding executing programs that we do not understand.

Although the mobile agent security field directly benefits from developing provable tamperproofing techniques, our results for program encryption apply to the computer science field in general and include protection for other significant program classes. Program classes exist containing applications with relatively small (bounded) input size and, in Section 5.7, we define an end-to-end methodology to protect these programs specifically. Using our methodology, we demonstrate how to protect several relevant program classes with perfect semantic secrecy. We give six specific applications next that benefit directly from the results described in this chapter—though there are certainly other candidate applications.

### 5.2.1 Mobile Agents

Mobile agents visit untrusted host environments and they do not always know or trust the security of their executing environment. Instead, agents must provide their own application layer security or the host middleware must enforce security on the agent's behalf. The best we can achieve for security in such environments is to reduce adversary's power from effective tampering to blind disruption. When we apply our black box protection mechanism in Section 5.4 to general programs, we can protect agents running in trusted hardware environments—as long as the TPH enforces virtual black box security (see Appendix A.3.19). Executably encrypted agents can hide their purpose/results in such a way that an adversary gains no benefit from trying to game their input/output or from altering the agent's code to their advantage.

### 5.2.2 Sensor Nets

Sensors are canonically resource-constrained devices that typically process small sized input data, e.g. 16 bits. A manufacturer could executably encrypt the embedded sensor code to protect their intellectual property. Take for example a sensor that we deploy in a remote operating location as illustrated in Figure 52. The sensor output is a broadcast stream of binary digits (64 bits at a time) that we carry by some means (satellite uplink possibly) to a remote processing facility. If an adversary captures the sensor and utilizes the capability to disassemble the sensor and look at its internal structure, the adversary may become aware (after some reverse hardware engineering process) that the sensor uses temperature readings and motion sensor related data. For temperature, the sensor uses an 8-bit input size (capturing a range from  $-100^{\circ}$  C to  $100^{\circ}$  C) and, for motion sensor data, the sensor requires 24 bits. The software inside the sensor thus takes 32 bits of input and outputs 64 bits of data every time it takes a reading (all of which are observable by the adversary).

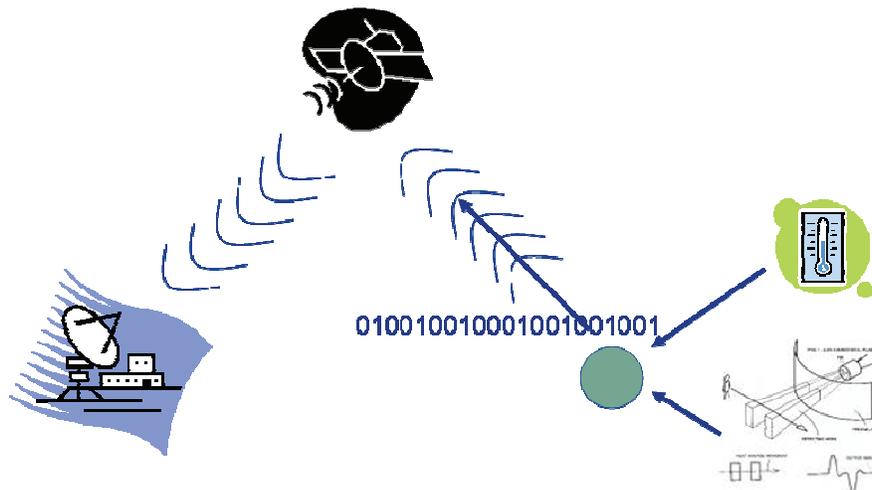


Figure 52: Application Example for Program Encryption

We want to protect the application intent for the software embedded in the sensor so that the adversary cannot foil the detection properties of the sensor. We want to prevent the adversary from understanding (based on the input) what processed information the sensor relays back to its processing facility. In other words, we want to protect provably the input/output (black box) relationships of the sensor and the algorithmic (white-box) information contained in the sensor's embedded circuitry.

### 5.2.3 Geo-Positional Location Information

Positioning devices utilize numerically intensive functions. We can often represent mathematical input very efficiently. Thus, location finding or tracking devices are potential program encryption applications.

### 5.2.4 Financial Transactions

There is a clear need to protect programs that compute financial data. Many important financial programs take small mathematical input and, thus can be target applications for perfectly secure obfuscation. The ability to hide small pieces of security information (bank PIN, account number) embedded in a user-specific financial application also becomes a reality under this construction.

### 5.2.5 Protecting Embedded Keys in Programs

A major contribution of our research includes a methodology to protect embedded key encryption algorithms contained in executing program code. For an application with suitable input size, we give a methodology to mask its input/output relationships and effectively protect its source code representation from leaking information. We can provably hide the key (and the seam between the application and the encryption algorithm) within the obfuscated program.

### 5.2.6 Transforming Private Key Crypto-Systems into Public Key Systems

Our approach to program encryption lays the foundation for solving the more (long-standing) problem in computer science of how to transform any private-key cryptosystem into a public-key system. Specifically, Alice can take a private-key data cipher with encryption algorithm  $E(K,M)$  and decryption algorithm  $D(K,C)$ , embed a specific private key  $KA$  in the encryption algorithm, obfuscate  $E(KA,M)$  to produce  $E'_{KA}(M)$ , and then publish the obfuscated cipher  $E'_{KA}$  as a virtual public key. Alice distributes  $E'_{KA}$  while keeping the private key  $KA$  secret. Bob can use  $E'_{KA}$  to send Alice encrypted messages that only she can decrypt (using  $D(K,C)$  and her secret key  $KA$ ). Diffie and Hellman [187] considered this idea in their original seminal work on passing secrets in public. As they point out, any encryption algorithm candidate  $E(K,M)$  must be complex enough so that input/output pair analysis does not easily reveal the decryption algorithm  $D(K,C)$ .

## 5.3 Defining Program Encryption

These motivating examples provide building blocks for the possibility of protecting mobile code from malicious execution environments and effectively protecting programs from malicious intent. Ultimately, we desire to prevent the adversary from *knowing* the purpose of a program in order to reduce attacks from *effective tampering* to *blind disruption*. Sander and Tschudin make this similar observation:

*“... if we can execute encrypted programs without decrypting them, we automatically have a) code privacy and b) code integrity in the sense that specific tampering is not possible. Attacks by a malicious host would then be reduced to actions “at the surface of the mobile agent”: denial of service, random modifications of the program or of its output as well as replay attacks.” [25]*

We posit that the research community already recognizes important opportunities for obfuscation *applications*, but they have yet to find a precise security definition with *positive* results that apply to current commercial obfuscation implementations. Our work leverages existing obfuscation techniques, but we move past traditional obfuscation to establish a baseline definition for the field of *program encryption*. In doing so, we hope to provide the community a practical yet theoretical basis for protecting programs while giving greater clarity to researchers for analyzing existing obfuscation techniques.

### 5.3.1 Measuring Cryptographic Security

Two approaches exist for measuring cryptographic security strength [188]: *information-theoretic* and *computational-complexity*. We base information-theoretic security on whether breaks are *possible* (unconditionally) and base computational measures on whether breaks are *feasible*. Concerning *data* ciphers, cryptographers deem an encryption scheme insecure in the information-theoretic sense if the ciphertext contains *any* information about the plaintext. In the computational-complexity model, cryptographers only care whether an adversary can efficiently extract information about the plaintext contained in the ciphertext. With information-theoretic secrecy, we use an *ideal* security model to show that any candidate security solution is nearly as good as the ideal one. This implicit approach is quite different from the explicit complexity method that must define an adversary task and then show that the task is computationally difficult.

### 5.3.2 Heuristic Views of Obfuscation

Heuristic techniques and some computational approaches represent a form of “fuzzy” security (neither well defined nor precise) because they rely on capturing all possible adversarial actions. These techniques also rely on less formal security properties that gauge an adversary’s mental or cognitive state concerning software (i.e., whether software is “hard to understand”). Table 11 summarizes several metrics that Collberg and his colleagues [167, 169, 170, 183, 185] use to define and analyze complexity. Defining adversarial actions requires ad-hoc definition and computational/heuristic approaches suffer typically from a use/break/tweak/use cycle as a result. These foundational differences in defining security apply directly to program obfuscation security.

**Table 11: Heuristic Obfuscation Metrics**

Metric Name	Definition
<b>Cyclomatic Complexity</b>	Function complexity increases as number of predicates increase
<b>Nesting Complexity</b>	Function complexity increases as conditional structure nesting levels increase
<b>Data Structure Complexity</b>	Function complexity increases as static data structure complexity increases
<b>Potency</b>	Measures the complexity of the obfuscated program versus that of the original program
<b>Resilience</b>	Measures the ability of an obfuscation to withstand a deobfuscation attack
<b>Overhead</b>	Measures the time or space increase of an obfuscation
<b>Stealth</b>	Measures the recognizable difference between obfuscated code and normal code within a program
<b>Quality</b>	Measures the combined qualities of potency, overhead, and resilience

Heuristic approaches for obfuscation include techniques based on the hardness of interprocedural analysis [183], key-based generation of pseudorandom encrypted code (decrypted just prior to execution) [164], and applying cryptographic primitives for constant hiding [175]. Drape [184] characterizes obfuscation as a refinement/proof process on data structures (versus algorithms). In almost all heuristic cases, the adversary has an ability to recover the original source code only related to the relative complexity of the obfuscated code version. As Table 11 illustrates, we can derive software complexity measures based on numbers of predicates [189], conditional structure nesting [190], and data structure complexity [191] and utilize them for obfuscation measurements. Many obfuscation techniques leverage known hard problems such as inter-procedural and control flow analysis [162, 163, 179] to provide complexity

increase. Other researchers use hardware supported program security [161, 192] and protecting embedded keys [146, 147].

Most obfuscators prevent an adversary from effective decompilation and re-assembly; practical obfuscation implementations invariably appeal to confusion or complexity as a measure of security. Specific obfuscation techniques vary greatly and we summarize the common ones in Table 12. Commercial obfuscators [193] use only a few of these techniques and we list several current products with their respective protection techniques in Table 13. Despite the lack of provable security properties, commercial vendors relate the security of their products to the inability of an adversary to reverse engineer, decompile, or effectively recover the original source code of an original program.

**Table 12: Heuristic Obfuscation Techniques**

Technique	Methodology
<b>Opaque Predicates</b>	Using predicates with known values at obfuscation time: always true, sometimes true, always false
<b>Variable Renaming</b>	Renaming variables and data structures to cognitively meaningless names
<b>Control Flow Mangling</b>	Reordering normal program control and execution flow to prevent decompilation and disassembly
<b>Memory Mangling</b>	Adding or reversing the order of addressing/dereferencing operations
<b>String Encryption</b>	Encrypting sensitive data strings using a data cipher and decrypting them prior to use
<b>Multiple Functions</b>	Introducing additional functions into code to obscure the original (intended) function
<b>Code Encryption</b>	Encrypting parts of the code using a data cipher and decrypting them prior to execution
<b>Loop Unrolling</b>	Confusing the normal logic of a loop by altering indexes or executing some number of loop runs
<b>Array Merging / Splitting</b>	Splitting an array into two arrays or merging two arrays into one large one in order to confuse the index logic
<b>Method Cloning</b>	Creating different versions of the same method
<b>Code Interleaving</b>	Merging two pieces of code in parallel and using specific means to distinguish the original methods. Interleaving unrelated code segments increases deobfuscation complexity
<b>Code Concatenation</b>	Merging two pieces of code serially by taking the output of one and using it as the input of the other: $f(x), g(y) \rightarrow g(f(x))$
<b>Code Outlining</b>	Taking a statement sequence and creating a separate function
<b>Code Inlining</b>	Replacing a function call with its actual code
<b>Random Statements</b>	Inserting execution neutral statements with proper characteristics in random and pre-selected places
<b>Randomized Ciphers</b>	Altering well-known data ciphers in random ways to produce embedded key-based encryptions unique to a particular application
<b>Code Morphing</b>	Creating self-modifying code that changes the runtime and static code structure of the obfuscated program on execution

### 5.3.3 Theoretical Views of Obfuscation

For some time, obfuscation researchers have found results based on both computational and information-theoretic models. The security characterization of obfuscation has been described as NP-easy [174], derivable in limited contexts [163, 175, 176], and proven to be NP-hard [182, 183, 186] / PSPACE-hard [179] based on specific protection mechanism. Yu and his colleagues have recently found several positive results for completely hiding circuit topology in the information theoretic sense [180, 181]. In Section 5.4, we introduce a secure black box program protection mechanism similar to Ostrovsky and Skeith's recent work [171] based on public-key obfuscation that produces encrypted, recoverable program output.

One definition of obfuscation is the ability to rewrite a program efficiently so that an adversary who possesses the obfuscation gains no advantage beyond having observable program

input/output behavior. This intuition has received substantial research and applied attention, yet a gap currently exists between practical and theoretical obfuscation security. The current de facto standard theoretical obfuscation model is the Virtual Black Box (VBB) paradigm [172]. Barak *et al.* prove that there is an unobfuscatable family of functions under VBB, and thus that efficient, *general* obfuscators do not exist. Wee [178] proves that we can obfuscate particular classes of point functions, whose result is true on one and only one input and false otherwise, under VBB given certain complexity assumptions. Lynn *et al.* [176] provide variations for protecting point functions based on random oracles while Canetti [177] demonstrates cases where hash functions replace oracles. Goldwasser and Kalai [173] show that you cannot efficiently obfuscate functions families with respect to a priori information given to adversaries—giving unconditional impossibility results under VBB unrelated to one-way functions.

**Table 13: Examples of Commercial Obfuscators**

Product	Company	Obfuscation Techniques
Dotfuscator (.NET) DashO (JAVA)	PreEmptive Solutions	Uses class/field/method renaming, string encryption, and control-flow confusion Available: <a href="http://www.preemptive.com">http://www.preemptive.com</a>
SourceGuard (JAVA)	4thPass	Uses class/field/method renaming, removes debug meta-data, and introduces control-flow confusion Available: <a href="http://www.4thpass.com">http://www.4thpass.com</a>
RetroGuard (JAVA)	RetroLogic Systems	Uses class file symbol renaming Available: <a href="http://www.retrologic.com">http://www.retrologic.com</a>
yGuard (JAVA)	yWorks	Uses class/field/method renaming Available: <a href="http://www.yworks.com">http://www.yworks.com</a>
Salamander (.NET)	RemoteSoft	Uses variable renaming and method overloading, removes debug meta-data Available: <a href="http://www.remotesoft.com">http://www.remotesoft.com</a>
JCloak (JAVA)	Force5 Software	Uses class file symbol renaming Available: <a href="http://www.force5.com">http://www.force5.com</a>
Smokescreen (JAVA)	Lee Software	Uses variable renaming, control-flow obfuscations (shuffles stack operations), and fake exceptions Available: <a href="http://www.leesw.com/smokescreen">http://www.leesw.com/smokescreen</a>
Klassmaster (JAVA)	Zelix	Uses variable renaming, string encryption, and control flow obfuscation (breaks up loops using gotos) Available: <a href="http://www.zelix.com/klassmaster">http://www.zelix.com/klassmaster</a>

We review briefly the first proof given by Barak *et al.* in [172] that no 2-Turing Machine (2-TM) or 2-Circuit obfuscator exists, as we reference the proof in our later constructions. Informally, we define an obfuscator  $O$  as an efficient, probabilistic algorithm that takes a program (or circuit)  $P$  and produces a new program or circuit  $P' = O(P)$ . Candidate obfuscators must exhibit the following properties in relation to  $P$  and  $P'$ :

- (1) **functionality**,  $\forall x, P(x) = P'(x)$ , where  $P' = O(P)$ ,
- (2) **polynomial slowdown**, which says  $O(P)$  is at most polynomially slower than  $P$  (for circuits the requirement is that the size of  $O(P)$  is at most polynomially greater than  $P$ ), and
- (3) **virtual black box (VBB) property**.

We define the virtual black box property uniquely for the class of programs (TM) or circuits we wish to analyze. Definition 1 and Definition 2 describe the requirements for 2-TM and 2-circuit constructions. The generalized VBB property mathematically states that you should not be able to learn more from the obfuscated version of a program ( $O(M)$ ) than from a simulator ( $S^{<M>}$ ) for the original program with oracle access. Equation 1 gives the formulation as follows:

**Equation 1.**  $|\Pr[A(O(M)) = 1] - \Pr[S^{<M>}(1^{|M|}) = 1]| \leq \text{neg}(|M|)$

**Definition 1. (2-TM Obfuscator)** A probabilistic algorithm  $O$  is a 2-TM obfuscator if the following three conditions hold:

1. (functionality) For every TM  $M$ , the string  $O(M)$  describes a TM that computes the same function as  $M$ .
2. (polynomial slowdown) The description length and running time of  $O(M)$  are at most polynomially larger than that of  $M$ . That is, there is a polynomial  $p$  such that for every TM  $M$ ,  $|O(M)| \leq p(|M|)$ , if  $M$  halts in  $t$  steps on some input  $x$ , then  $O(M)$  halts within  $p(t)$  steps on  $x$ .
3. (VBB property) For any PPT  $A$ , there is a PPT  $S$  and a negligible function  $\alpha$  such that for all TMs  $M, N$ :

$$|\Pr[A(O(M), O(N)) = 1] - \Pr[S^{<M>, <N>}(1^{|M|+|N|}) = 1]| \leq \alpha(\min\{|M|, |N|\})$$

We say that  $O$  is efficient if it runs in polynomial time.

**Definition 2. (2-Circuit Obfuscator)** A probabilistic algorithm  $O$  is a 2-circuit obfuscator if the following three conditions hold:

1. (functionality) For every circuit  $C$ , the string  $O(C)$  describes a circuit that computes the same function as  $C$ .
2. (polynomial slowdown) There is a polynomial  $p$  such that for every circuit  $C$ ,  $|O(C)| \leq p(|C|)$ .
3. (VBB property) For any PPT  $A$ , there is a PPT  $S$  and a negligible function  $\alpha$  such that for all TMs  $M, N$ :

$$|\Pr[A(O(C), O(D)) = 1] - \Pr[S^{C, D}(1^{|C|+|D|}) = 1]| \leq \alpha(\min\{|C|, |D|\})$$

We say that  $O$  is efficient if it runs in polynomial time.

**Proposition 1.** Neither 2-TM nor 2-Circuit Obfuscators exist.

In [172], the proof for Proposition 1 that 2-TM/2-Circuit obfuscators do not exist illustrates the nature of the contrived functions used in all their proofs. Specifically, they contrive two functions:  $C$  is a point-function that takes in a string of size  $k$  and returns a string of size  $k$  and  $D$  is a TM/circuit *decider* that takes in the description of a TM/circuit and outputs a Boolean answer  $(0, 1)$ . Both  $C$  and  $D$  depend on parameters  $\alpha, \beta \in \{0, 1\}^k$  where  $k \in \mathbb{N}$  in the following manner:

$$C_{\alpha, \beta}(x) \stackrel{\text{def}}{=} \begin{cases} \beta & x = \alpha \\ 0^k & \text{otherwise} \end{cases}$$

$$D_{\alpha, \beta}(C) \stackrel{\text{def}}{=} \begin{cases} 1 & C(\alpha) = \beta \\ 0 & \text{otherwise} \end{cases}$$

In essence,  $D_{\alpha, \beta}$  is a decider for some circuit  $C_{\alpha', \beta'}$ . If  $(\alpha = \alpha')$  and  $(\beta = \beta')$  for  $C_{\alpha', \beta'}$ , then  $D_{\alpha, \beta}$  returns 1. Otherwise,  $D_{\alpha, \beta}$  always returns 0.  $Z_k$  is a TM machine that always returns a string of  $k$  zeros,  $0^k$ . In order to prove the claim that 2-TM/2-Circuit obfuscators do not exist, we can show the VBB property violation in the following manner. First, we define an adversary PPT  $A$  that receives the *description* or *source code* of two circuits as input. The adversary simply runs the second circuit on the first circuit:  $A(C, D) = D(C)$ . If the adversary were given  $A(C_{\alpha, \beta}, D_{\alpha, \beta})$ ,  $D_{\alpha, \beta}$  always returns a 1 when given  $C_{\alpha, \beta}$  as input. Thus, the probability that an adversary, when given any equivalent version of  $C_{\alpha, \beta}$  and  $D_{\alpha, \beta}$  (which of course includes obfuscated versions of  $C_{\alpha, \beta}$  and

$D_{\alpha,\beta}$ ),  $D(C)$  **always** returns 1. Equation 2 states this relationship. Equation 3 shows that when we give the adversary a description for an all zero function  $Z_k$ , then  $D_{\alpha,\beta}(Z_k)$  **always** returns 0. Thus the probability that  $D_{\alpha,\beta}(Z_k)$  returns a 1 is **always** 0.

**Equation 2.**  $\Pr[A(O(C_{\alpha,\beta}), O(D_{\alpha,\beta})) = 1] = 1$

**Equation 3.**  $\Pr[A(O(Z_k), O(D_{\alpha,\beta})) = 1] = 0$

The contradiction for Definition 1 and Definition 2 arises when we consider computing the  $(0,1)$ -predicate concerning programs  $C, D$ , and  $Z$ . If we execute *any* version of the source code for  $C, D$ , and  $Z$ , we can compute predicates (in Equation 2 and Equation 3) absolutely. However, when given PPT simulators of  $C, D$ , and  $Z$ , we cannot compute the same predicates with better odds than guessing. A poly( $k$ )-time algorithm  $S$  which has oracle access to  $C_{\alpha,\beta}$  and  $D_{\alpha,\beta}$  (represented by  $S^{C_{\alpha,\beta}, D_{\alpha,\beta}}$ ) cannot be distinguished from another algorithm  $S$  which has oracle access to  $Z_k$  and  $D_{\alpha,\beta}$  (represented by  $S^{Z_k, D_{\alpha,\beta}}$ ). We express this in Equation 4 and show that having oracle access is less powerful than having (obfuscated) source code access. Therefore, under VBB, *no amount* of obfuscation / confusion ever overcomes this inherent limitation for defining semantic security.

**Equation 4.**  $|\Pr[S^{C_{\alpha,\beta}, D_{\alpha,\beta}}(1^k) = 1] - \Pr[S^{Z_k, D_{\alpha,\beta}}(1^k) = 1]| \leq 2^{-\Omega(k)}$

The strength of the impossibility proofs depend on large  $k$  and the contrived examples assume we must provide the decider  $D$  a TM (or circuit) description. As we point out by the constructions in Section 5.7, given a small  $k$  with associated polynomial bound, a simulator with oracle access can construct a circuit based on the enumeration of all truth table inputs in polynomial time, even though the algorithm it uses is exponential. Using our methodology, we demonstrate the possibility of perfect semantic encryption for a relevant class of programs.

### 5.3.4 Why We Need a Different Security Model

The VBB model of measuring *obfuscation* security essentially levies an information theoretic requirement: an adversary should learn no more when given the obfuscated version (i.e., executable ciphertext) of a program than it should when given black box access to the original (executable plaintext) version of the program. Because of the impossibility results under VBB, it has been very hard (impossible) for any practical implementations of obfuscation to demonstrate *measurable* security properties.

Barak *et al.* claim that the virtual black box paradigm is “inherently flawed”. Since the VBB model is unsuitable for reasoning about program obfuscation, we require a new model if we hope to effectively hide program properties for security. Researchers suggest that these foundations leave us two directions to pursue:

- (1) *Are there weaker or alternative models for obfuscation that provide meaningful results?*
- (2) *Can we construct obfuscators for restricted but non-trivial/interesting classes of programs?*

In other words, can we prove practical obfuscation methods secure against some threats and attacks, but not necessarily all? We believe an alternative model for describing obfuscation security strength based on the complementary notions of random programs and black box semantic transformation give an affirmative answer to these questions. We provide a basis for understanding intent-protected programs using this paradigm in Section 5.5 and consider obfuscators that make random selections from a set of black box protected programs in Section 5.6. As a result, we relax both the hiding property and the program classes considered for

obfuscation. We purposefully produce obfuscated programs that are not semantically equivalent to the original version so that  $M(x) \neq O(M(x))$  and we show that a general obfuscator exists in our model that is not subject to Barak's impossibility proof.

### 5.3.5 Program Understanding

We propose to protect programs from tampering by hiding their intent—essentially preventing intruders from understanding a program. The direct implication is that if malicious parties do not know what the program is trying to do, they cannot perpetrate attacks that achieve a predictable manifestation. Thus, their interference is limited to blind disruption, or at least to a subset of well-known, non-application specific attacks (e.g. buffer overflow attacks that have no semantic application relationship).

We consider four distinct, but related program-understanding paradigms. In the first, we consider the generic, intuitive notion of “understanding” that an adversary's ability to anticipate a program's operational manifestation(s) reflects their program understanding. Secondly, an adversary may gain intent indications by comparing the obfuscated code, or segments, to known code libraries. Third, we recognize VBB's theoretical and practical importance. Finally, information content in program code is our primary focus.

While we motivate our work by using program prediction for malicious purposes and obfuscation for security, the notion of program clarity for maintenance applies in a direct way. A maintenance programmer must be able to understand program intent in order to make purposeful changes, e.g. to fix bugs, improve performance, and port code to a different environment. In the same sense, a malicious host must understand what a program is doing (in some sense) to effectively copy, modify, run, or forward the program to accomplish a semantic-oriented purpose.

Side effects are an example of an unintended outcome of a program, segment, or construct, or at least an outcome that is not clearly intended. Some programmers consider their code elegant because of their stylistic use of obscure approaches to accomplish intended function in ways that are not obvious. When programs with obscure mechanisms are changed, the maintenance programmer is unlikely to recognize all the impacts of the change. Our review of heuristic obfuscation techniques and commercial obfuscators bears witness that understanding programs precisely is a naturally hard problem.

For example, attackers may not require precision; i.e. they may only need a high-level understanding of program function or be able to recognize a subset of the functionality in order to accomplish their intended malice. Once again, there is little in the literature that quantifies or qualifies the level of understanding necessary to maintain, or attack, a program. We offer our formalization in this regard and begin first with the intuition behind it.

The foundation for our approach is that an adversary only understands a program if they are able to predict its operation in one of two ways. First, an adversary that understands a program can predict a program's output with any given input. For example, for the program that computes the simple function given in Equation 5, an adversary need not run the program to know that its output is 7 on input 2. As a more complex example, consider a program  $P$  that implements a small degree polynomial. Even if an adversary is unable to expose  $P$  itself, but can plot a graph based on gathered input-output pairs, they may be able to guess output for a given, arbitrary input without running  $P$ .

**Equation 5.**  $y = x + 5$

The second notion regarding program understanding is that an adversary that understands a program is able to reason about the input required to produce a desired semantic result. For the program  $P$  that implements Equation 5, an adversary that understands  $P$  and desires that  $P$  produce an output of, say 19, knows to feed 14 into the program. This "one-way" property captures the important intent quality we focus on. A common threat to mobile code is that the adversary desires the query to produce a favorable result from their perspective. Accordingly, their goal is to modify the input or code to produce a result with these properties. If we protect the

mobile code's intent, the adversary can only guess with low probability at the input necessary to produce the desired result.

We mention intuition and graphing as ways that an adversary may come to understand a program's intent, but there are many others. For example, an adversary may be able to guess the output of  $P$  by determining that  $P$  is equivalent to another program<sup>8</sup>,  $P_i$ , that the adversary recognizes (i.e. understands its intent). Essentially, the adversary could run  $P_i$  as their "prediction process" as long as they are confident that for any arbitrary input  $x$ ,  $P(x) = P_i(x)$ .

We formalize program understanding in Definition 3 as an entity's ability to derive the input corresponding to an arbitrary output based on their program understanding. While we speak in terms of functional response, we recognize the broader notion of any persistent state change or information transfer to another process or device as output<sup>9</sup>.

**Definition 3. (Program Understanding)** *Alice understands terminating program  $P: X \rightarrow Y$ , iff given arbitrary output  $y \in Y$ , Alice can guess  $x \in X$  such that  $y = P(x)$  in polynomial time on the length of  $P$ , with probability greater than  $\varepsilon$ , where  $\varepsilon$  is a small constant.*

We consider *Program Understandability (PU)* to be Boolean. That is, given an arbitrary program  $P$ , there may exist an algorithm  $A_{PU}(P)$  that returns either true or false if it understands  $P$ . It is possible that *PU* is Boolean, yet that no efficient algorithm exists that distinguishes between programs that are understandable and those that are not. It is also possible that the Boolean viewpoint is too narrow. For example, there may be programs that have no notion of understandability, i.e. programs that have no overriding intention<sup>10</sup> or pattern (possibly created with that in mind to confound potential intruders<sup>11</sup>).

If *PU* is Boolean, we can use this to reason about what it means to understand a program. Consider the set of all programs,  $P$ . We can partition  $P$  into two subsets, the set of all understandable programs ( $R$ ) and the set of all non-understandable programs ( $U$ ), where  $P = R \cup U$ . We observe that many functions are fundamentally understandable, and therefore we cannot securely obfuscate them. For example, for any program  $P$  that implements the function  $y = x^2$ , the input/output patterns of  $P$  reveal its function clearly. No matter how random the code implementing this function may be, an adversary need not look at the code to know what the program is doing. It need only conduct black box (input/output) analysis.

Since  $P$  is infinite, one or both of the sets  $R$  and  $U$  are infinite. It is also reasonable to ask if either  $R$  or  $U$  is empty. In the former, we may argue that ALL programs have unintended impacts at some level of abstraction, or even that our ability to articulate intentions precludes any program from comprehensively meeting them. In the latter, we may point to the Barak result as sufficient to ensure that  $U$  is empty. We know that simple polynomials are not good candidates for intent protection, but we posit that strong encryption functions are excellent candidates. Specifically, we know that cryptographically strong data ciphers are not susceptible to black box analysis. However, all well-known encryption algorithms have program structures that betray their intent to a sophisticated adversary with white box analysis capabilities.

We have a strong intuition regarding what it means to *understand* a program from Definition 3. However, we have not formalized what it means for a program to be *understandable*. Following the security paradigm of data encryption, we define secure *obfuscation* only if an obfuscated program leaks no intention-relative information, i.e. it is indistinguishable from a *random* program. We argue that this notion is sufficiently strong to preclude intentioned attacks, though we recognize that weaker formalizations may prevent some (or even most) intentioned attacks. Thus, a conservative protection goal is to generate "*executably-encrypted*" code that is indistinguishable from random programs, which we define in Section 5.6. The manifestation of this outlook is that if

---

<sup>8</sup> We consider program equivalence issues separately

<sup>9</sup> Obviously, programs that do not have output in this sense are not necessarily suitable to our obfuscation approach.

<sup>10</sup> Random programs or programs that have no impact on the environment.

<sup>11</sup> We presently ignore the self contradiction of having programs whose purpose is to have no purpose.

we effectively obfuscate (intent protect) a program, an adversary cannot predict or guess the program's behavior (as in Definition 3).

### 5.3.6 Program Context

A major challenge to protecting a program's intent is the role that contextual information plays. In most mobile applications, it is impossible to protect all contextual information from the executing host. Items such as program size, execution time, controlled input performance and resource use variations, response to injected errors, and many other operational program aspects are under the executor's control. It is a prerequisite for protecting a program's intent that the adversary has limited contextual information available. Thus, inherently, we cannot obfuscate many programs using our approach.

Consider an agent program that comes from a vendor known to provide travel plans, and the computer we access contains only flight information and pricing for a known airline with very limited availability dates (e.g. last minute flights). In this case, even a casual observer may infer that the program is gathering flight information to prepare imminent travel plans for the dispatcher's client. If an adversary knows too much context, intent protection is unlikely. Therefore, we assume context-independent protection suffices for our methodology.

### 5.3.7 Protecting Program Intent using Program Encryption

The VBB flaws result from the breadth the approach seeks, essentially to be a comprehensive model for all program obfuscation. Our goal is comparatively modest. We reduce the goal from general obfuscation to protecting *program intent*, under our narrower definition, against specific attacks. As we have illustrated, we limit our model by recognizing that there are programs that we cannot obfuscate (securely). There are also programs that we can obfuscate but the approach we describe in this chapter may not be appropriate for them.

For our purposes, we consider intent protection a game between an originator and an adversary or intruder (we use these terms interchangeably). We consider that intruders desire to understand or recognize programs (discern their intent) for three purposes:

- (1) *To manipulate the code in order to attain a known output effect*
- (2) *To manipulate input to attain a known output effect*
- (3) *To understand the input/output correlation for use with contextual information*

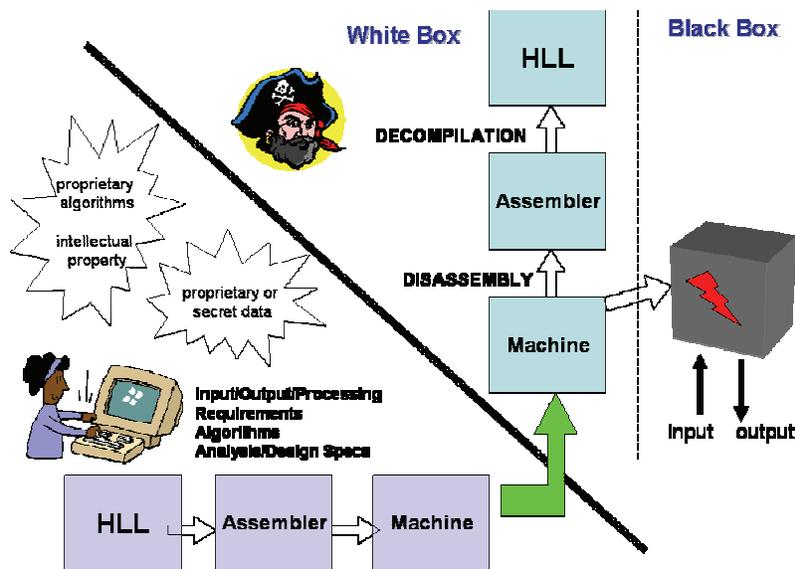
We illustrate the first two of these by considering an Internet purchase application where a mobile agent gathers bids for a product or service. If the adversary residing on a visited host recognizes the program, they may manipulate the data they provide to the agent or they may locally modify the agent code in order to elevate their opportunity to win the bid falsely. Intent protection (hiding) does not prevent an intruder from changing input or code, but reduces this type of tampering to blind disruption by preventing the intruder from being able to predict the effect of an input or code change.

In the third objective, we envision adversarial environments where parties gather information or intelligence about one another. In an Internet purchase scenario, adversaries may operate with modified purposes. Here we anticipate that the adversary may gain important information, not so much about the specific transaction that is underway, but about the underlying business practice or strategy that the agent executes. If the adversary is able to understand what the program is doing, it may be possible to infer fundamental business information from the transaction. Conversely, if the program does not divulge its intent, an intruder is unable to gather any information about the dispatcher's activity.

Program intent may become evident through repeated execution and observation of the input-output pairs, so programs that hide their intent must protect against *black box analysis*. Malicious parties that acquire code or can corrupt hardware may be able to examine executing code with automated tools such as debuggers. As Figure 53 depicts, there are three primary approaches to **context-independent program intent detection**:

- (1) *Input-output (black box) analysis*
- (2) *Static analysis*
- (3) *Run-time analysis*

We recognize that malicious parties are likely to attack intention protection using hybrid methods that combine static analysis, black box testing, and dynamic analysis. We collectively term latter activities as *white box analysis*. Program Recognizability (PR) is a classic concept in computer science and relates to Program Understandability (PU). Classic PR refers to the context-free notion of being able to determine whether a string is a member of a particular language. This represents a form of static analysis. Compiler optimization techniques refine the class of languages that automata can recognize, allowing program segment identification through signature analysis. Combined with reverse engineering techniques, compiler optimization techniques complicate hiding program intentions.



**Figure 53: Adversarial Program Intent Detection**

Static analysis involves actions that an adversary takes without executing that code. Static approaches include inspection, parsing, optimization, pattern matching, etc. These actions can give the adversary hints about the nature of the data, control structures, resources used by the program, etc. Dynamic analysis occurs as the program is executing. Run-time tools such as debuggers reveal control flow, data manipulations and evolution, and resource access and consumption. If either static or dynamic analysis or the two applied collaboratively can reveal a program's intent, the program is white box understandable.

Traditional obfuscation applies data and control flow confusion techniques to complicate these attacks, with little or no measurable protection. Only imagination and resources limit the number of methods that a motivated and sophisticated adversary can employ to reveal a program's protected intent. Nonetheless, the literature point to black box and white box analysis as the classical approaches for defeating program obfuscation. Without loss of generality, we address these attacks by assuming an adversary maliciously examines programs off line, where they exploit software on computers with large, but polynomially bounded resources. In practice, the adversary may only be able to employ on line attacks (especially for time-dependent mobile agents). In either case, while the adversary can glean much of the information gathered in off line attacks just as easily from on line attacks, off line attacks reflect the stronger adversarial model.

Consider an industrial application on a stolen laptop. An adversary may desire to know how the laptop owner generates business or financial estimates, how their decision process works, or

other business or organizational information. For a **black box protected** program, the enemy cannot determine the device's function from an arbitrary number of input-output pairs. However, if the enemy is sophisticated, they examine the executable code structure or analyze the application's control flow and data manipulations as they occur. Programs that are **white box protected** prevent the enemy from learning the program's intent by watching its execution.

We formalize the intuition and ideas for understandability, obfuscation, and intent protection in the following definitions. Because we do not attempt to formalize the definition for white box analysis approaches (static and dynamic adversary analysis), we give only an introductory, informal white box definition. In Section 5.5.2, we give a more formal definition for white box protection based on the existence of a random program oracle.

**Definition 4. (Black Box Understandable/Obfuscated)** Program  $P \rightarrow \{X, Y\}$  is black box understandable if and only if, given an arbitrarily large set of pairs  $IO = (x_i, y_i)$  such that  $y_i = P(x_i)$  and  $y_j$  an arbitrary element of  $Y$  (not an element of  $IO$ ), an adversary can guess [compute]  $x_j$  such that  $y_j = P(x_j)$  in polynomial time on the length of  $P$  with probability  $> \epsilon$ . Otherwise, we say  $P$  is black box obfuscated.

**Definition 5. (White Box Understandable/Obfuscated, Informal)** Program  $P$  is white box understandable if it is understandable (under Definition 3) through static or dynamic analysis of  $P$  or a collaboration of the two. Otherwise, we say  $P$  is white box obfuscated.

**Definition 6. (Intent Protected)** Program  $P$  is intent protected if and only if it is black box protected, white box protected, and protected from any composition of the two. If  $P$  is both black box obfuscated and white box obfuscated, then  $P$  is also intent protected. We refer to an intent-protected program  $P$  as an executably encrypted program or as a program that implements program encryption.

## 5.4 Creating Perfect Black Box Obfuscation

We naturally encapsulate program functionality by pairs that map pre-image to image. Thus, a natural way to try to identify a program's intent is to analyze known input/output pairings. Traditionally, obfuscation has considered producing different versions of the same program, where one version is (or likely is) understandable, but the obfuscated version of the same program is not understandable (more complex to understand). Barak *et al.* show this form of obfuscation is impossible in the general, efficient case. We now build an alternative model for defining obfuscation under the narrow definition of intent protection using the existence of one-way functions and strong cryptographic data ciphers.

### 5.4.1 One-Way Functions and Black Box Obfuscation

Since protection of programs that retain their semantic equivalence is impossible, we appeal to a class of functions that have known (strong) cryptographic properties and apply their use to the obfuscation problem. We begin first by stating function definitions used in traditional cryptographic arguments [188, 194].

**Definition 7. (One-Way Functions and Permutations)** A function  $f$  with domain  $X$  and range  $Y$ ,  $f: X \rightarrow Y$ ,  $x \in X$ ,  $y \in Y$ , is called a one-way function if,  $\forall x \in X$ ,  $f(x)$  is easy to compute and if,  $\forall y \in Y$ , it is computationally infeasible given any  $y$  to find  $x$  such that  $y = f(x)$ . A one-way permutation is a bijection from the set of all binary strings with length  $n$  to itself, whose image is easy to compute, but whose inverse pre-image is difficult to compute:  $f: \{0, 1\}^n \rightarrow \{0, 1\}^n$ .

There are some cases where for some values  $y \in Y$ , it is easy to find an  $x \in X$  such that  $y = F(x)$ . One may compute several values for  $y = F(x)$  for a small number of  $x$  and find an appropriate inverse based on table look up. We specify, normally, that the inversion process is

hard for any  $x$  chosen randomly from  $X$ . Cryptographers have given the subject of one-way functions rigorous treatment. Goldreich [188] lists several candidate one-way functions including integer factorization, Rabin quadratic residue functions, discrete logarithms in a finite field (RSA), and the DES function families. “Hard to invert” normally means that an upper bound of success exists for some efficient inverting algorithm. Therefore, proving one-way functions exist implies that complexity classes  $P \neq NP$ . We assume one-way functions exist (as most cryptographers do) and appeal to their strength in describing cryptographically strong obfuscation (program encryption).

For strong cryptographic data ciphers, encryption is an algorithm that computes (efficiently) the functional output (ciphertext) of any given input (plaintext), but the inverse of any functional output (the ciphertext) is hard (infeasible) to compute. Of course, the purpose of data ciphers is that recovery of the plaintext from the ciphertext is feasible given some piece of special knowledge (normally termed the key). We capture this notion by defining a trapdoor one-way function in Definition 8. In Definition 9, we express the definition for cryptographically strong data ciphers that are not breakable (systematically), apart from brute-force key discovery.

**Definition 8. (Trapdoor One-Way Functions)** *A function  $f: X \rightarrow Y$  is trapdoor one-way if  $f$  is a one-way function and, given some additional information termed the “trapdoor” and given any  $y \in Y$ , it is feasible to compute  $x$  such that  $y = f(x)$ .*

**Definition 9. (Strong Data Encryption)** *An encryption scheme is breakable, if an adversary (without prior knowledge of the encryption or decryption keys) can systematically recover plaintext from a ciphertext efficiently (within a specified time). Encryption schemes that are not breakable (apart from brute-force key search) exhibit strong data encryption security and are representations of trapdoor one-way permutations.*

Based on these definitions, we now pose the possibility of black box obfuscators that support intent protection. Barak *et al.* show that general, efficient obfuscators do not exist if one-way functions exist. Unlike their impossibility result based on a contrived function family, we demonstrate here that *unless* one-way functions exist, secure obfuscation that guarantees intent protection is impossible. We focus first on black box obfuscation.

**Proposition 2.** *If (efficient) trapdoor one-way functions exist, then general (efficient) black box obfuscators exist (under Definition 4).*

Cryptographically strong *program* obfuscation results from the nature of strong *data* encryption algorithms. We prove Proposition 2 with two lemmas and one theorem. Lemma 1 states that given an arbitrary ciphertext output  $y$ , an adversary cannot efficiently compute the corresponding input to a semantically strong encryption program  $E$ . This represents the property of a strong data encryption algorithm under Definition 8 and Definition 9. We define such properties to be the fundamental characteristics of *any* strong *program encryption* algorithm. In Lemma 2, we use black box obfuscated programs as the starting point to consider situations where adversaries are able to extract executing code for out-of-band, white-box analysis.

**Lemma 1.** *Any program that implements a cryptographically strong data encryption algorithm is black box obfuscated (under Definition 4, Definition 9).*

**Proof:** Arbitrarily select the cryptographically strong data encryption algorithm  $E$ , a plaintext message  $M$ , and choose encryption key  $K$  randomly from the uniform distribution of possible keys in the keyspace. Assume  $E$  is black box understandable. Then there exists  $y = E(M, K)$  where an adversary can guess  $M$  given  $Y$  with negligible probability. This violates the definition of cryptographically strong data encryption. Similar to Lemma 1, If an adversary can efficiently guess the cipher text for one plaintext message it can easily distinguish that cipher text from the cipher text of another message. This contradicts the encryption algorithm's strong semantic security.

**Lemma 2.** Programs that are not one-way functions cannot be intent-protected obfuscated (under Definition 6) by any obfuscator  $O$  where  $O(P) = P'$  and,  $\forall x, P'(x) = P(x)$ .

**Proof:** Follows directly from Definition 4 and Definition 7. Given the family of programs  $\mathcal{P}$ , and for all programs  $P \in \mathcal{P}$ , assume  $P: X \rightarrow Y$  is *not one-way*. Assume obfuscator  $O$  is a black box obfuscator for the class of programs  $\mathcal{P}$  such that  $O(P) = P'$  and,  $\forall x, P'(x) = P(x)$ . Therefore,  $\forall x \in X, P(x)$  is easy to compute and  $\forall y \in Y$ , it is computationally feasible given any  $y$  to find  $x$  such that  $y = f(x)$ . Given any program  $P \in \mathcal{P}$ , an adversary can guess [compute]  $x_j$  such that  $y_j = P(x_j)$  in polynomial time on the length of  $P$  with probability  $> \epsilon$ . Therefore,  $P' = O(P)$  is black box understandable for all  $P$  in  $\mathcal{P}$ . This contradicts the statement that  $O$  is a black box obfuscator for the class of programs  $\mathcal{P}$ .

We stipulate by these two lemmas that all obfuscators  $O$ , where  $P' = O(P)$ , must semantically change the I/O mappings of any candidate program  $P$  into strong, one-way functional relationships in order to achieve black box obfuscation. Therefore, we alleviate all black box analysis threats as a foundational (first) program encryption step.

An interesting and important side effect is that this property simply and absolutely insulates our model against the impossibility result in [172]. Their elegant impossibility proofs rely on the existence of a Turing machine decider ( $D$ ) that, given a program or circuit description, can appropriately detect a particular function type. In our model, this proof technique cannot apply, since all obfuscations we create are one-way functions. There are no point functions (the type of function that Barak *et al.* used in their proof), nor are there any other functional program categories. Thus, the only relevant decider is one that detects one-way functions; such a decider will return “true” on all obfuscations under our model.

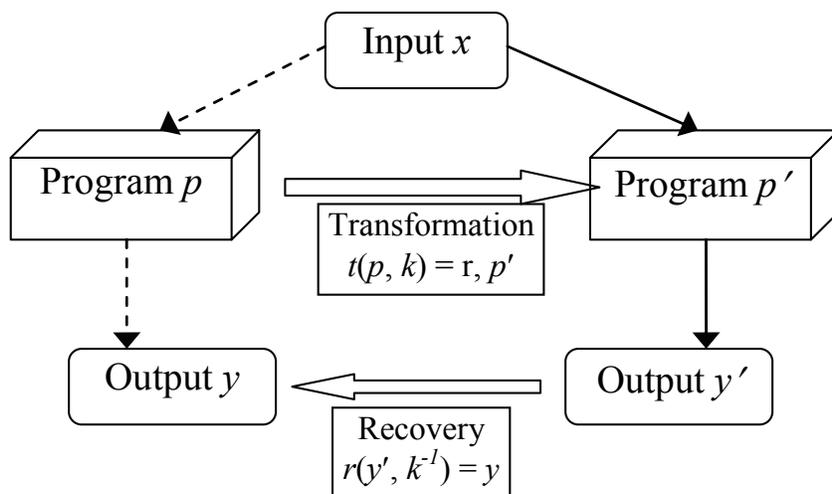
#### 5.4.2 Implementing Perfect Black Box Obfuscation

We now consider obfuscators that deviate from the semantic equivalence rule under VBB and implement Lemma 2. Sander and Tsichudis [159], Ostravsky and Skeith [171], and Adida and Wikström [195] all adopt similar non-semantic equivalence approaches in their respective program protection models. The fundamental property of our model, shown in Figure 54, is that the output of the obfuscated program ( $p'$ ) is not equivalent to the output of the original program ( $p$ ). In other words, if  $t(p,k)$  produces  $p'$ , then  $\forall x, p'(x) \neq p(x)$ .

We define the semantic encryption transformation (SETS) process  $t$  that generates a program version ( $p'$ ) that is not black box understandable (under Definition 4). The program  $p'$  must have recoverable (invertible) functionality with respect to the output of  $p$ ,  $y = p(x)$ . We accomplish this by creating  $p'$  as the concatenation of the original program  $p$  with a strong encryption algorithm  $e$  so that for all  $x \in X, p'(x) = e(p(x))$ . Equation 6 describes the output of obfuscation process  $t(p,k)$ : given a program  $p$ , we generate a new program ( $p'$ ) and a recovery program ( $r$ ) with the properties that  $p(x) = r(p'(x))$  and where  $r$  is simple to compute and output of  $p'(x)=y'$  is simple to invert given knowledge of special information ( $k^{-1}$ ).

$$\text{Equation 6. } t = \begin{cases} p' = e(p, k) \\ r := d(y', k^{-1}) \end{cases}$$

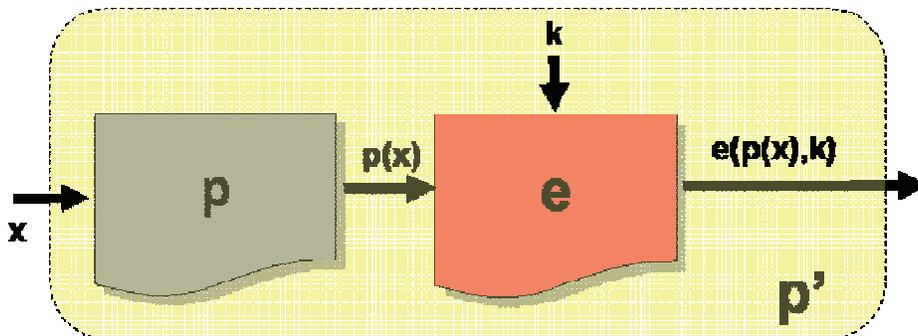
The obfuscation process uses a key that provides security control and allows correlation with data encryption paradigms. To be cryptographically strong, the obfuscation method must be public and its strength dependent only on knowledge of the key. We express the protection properties of such a transformation process formally in Theorem 1 and illustrate the black box obfuscated program in Figure 55.



**Figure 54: Black box Obfuscation with Recovery Model**

**Theorem 1.** Let  $t(p, e, k) = (p', r)$  be a process that creates program  $p'$  by composing the output of program  $p$  to the input of a black box obfuscated data encryption program  $e$  (defined under Lemma 1). Then  $p'$  is black box obfuscated.

**Proof:** Follows directly from Definition 4 and Lemma 1. If  $e$  is black box obfuscated, then  $p'$  is also black box obfuscated since the output of  $p'$  is [also] the output of  $e$ :  $y' = p'(x) = e(p(x), k)$ .



**Figure 55: Black box Obfuscated Program**

We emphasize that Theorem 1 is the foundation for any further (white-box) obfuscation approaches. If an adversary can interpret or understand a program through black box analysis, the program is not obfuscatable. This approach overcomes the primary weakness of the Virtual Black Box (VBB) paradigm and ensures us that black box protection is not only possible for general programs, but it is easy to accomplish. Furthermore, it gives us insight into why obfuscation is meaningful. The notion of intentional manipulation precisely captures an important intrusion category and limits blind disruption to sophisticated intruders. Moreover, it provides a foundation to expand our research into situations where adversaries are able to extract executing code for out of band, white box, analysis. To define such white box attacks more formally, we discuss in the next section a program-intent protection model that uses random bit string properties as a measurement basis. We term this the *random program security model* and illustrate its usefulness for analyzing white box protection next.

## 5.5 Defining the Random Program Security Model

To summarize the main result of Section 5.4, black box obfuscated programs must be the foundation to protect against out-of-band, white-box analysis. Most commercial and heuristic-based obfuscators focus attention on white-box unrecognizability and complexity. Most have no theoretical basis apart from appeals to provably hard (NP-complete) problems that increase complexity of adversary analysis. We too seek to make programs unrecognizable in some sense, but introduce in this section a formal, theoretical framework to measure such confusion. As we appealed to the cryptographic properties of strong data ciphers for black box obfuscation, we appeal now to the cryptographic properties associated with randomness and pseudo-random number generators. Instead of using random *data strings*, however, we use random *programs* as a security baseline. To outline our model, Section 5.5.1 briefly introduces random programs and shows their similarity with random data, Section 5.5.2 gives a formal definition for white box protection using random programs, Section 5.5.3 gives proof for the existence of random programs, and Section 5.5.4 describes the existence and construction of random circuits.

### 5.5.1 Random Data and Random Programs/Circuits

We believe provable program protection can only find its security characterization by comparison with provable data protection. When evaluating cryptographic data ciphers, we assume that mechanisms exist to simulate truly random bit strings. We can compare encrypted data to the output of a pseudo-random number generator—that we assume to mimic a truly random number generator given an appropriate seed. Program ciphers, likewise, need to have a baseline for comparison; we refer to this baseline as the “random program”.

Several cryptographic constructions rely on the existence of (pseudo) random *data* strings that are indistinguishable from truly random data strings. In measuring data randomness, we concern ourselves with sets of binary strings with the same length  $n$ ; the set of all strings of the same length we term an ensemble. A pseudo-random string is close enough to random if a polynomial distinguisher cannot tell it apart from a truly random string efficiently. We give an intuitive definition for a pseudo-random generator in Definition 10 and relate the traditional formal definition for indistinguishability in Definition 11 [194].

**Definition 10. (Pseudorandom Generator)** *A deterministic program that, when given a short random sequence of bits (termed the seed), generates a long sequence of bits which look like random bit sequences.*

**Definition 11. (Polynomial Time Indistinguishability)** *Two bit string ensembles  $X = \{X_n\}_{n \in \mathbb{N}}$  and  $Y = \{Y_n\}_{n \in \mathbb{N}}$  are indistinguishable in polynomial time if for every probabilistic polynomial-time algorithm  $D$ , every positive polynomial  $p(\cdot)$ , and all sufficiently large  $n$ :*

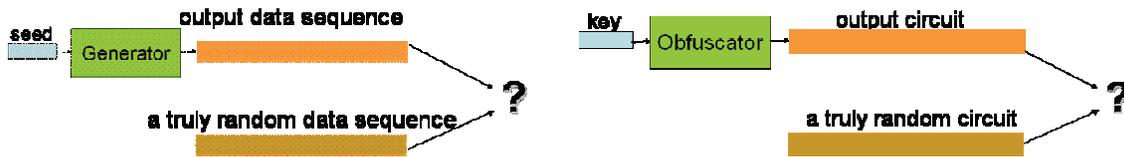
$$\left| \Pr[D(X_n, 1^n) = 1] - \Pr[D(Y_n, 1^n) = 1] \right| < \frac{1}{p(n)}$$

*where the probabilities are taken over the relevant distribution ( $X$  or  $Y$ ) and over the internal coin tosses of algorithm  $D$ .*

According to this definition, the probability that  $D$  accepts (outputs 1 on input) a string taken from the first distribution ( $X_n$ ) compared to the probability that  $D$  accepts a string taken from the second distribution ( $Y_n$ ) is negligibly different. In other words, if the two probabilities are close, we say that  $D$  does not distinguish the two distributions. For cryptographic algorithms, this reflects the foundational concept for “efficient” procedures that have the ability to distinguish the output of two different algorithms. The VBB proofs appeal to similar indistinguishability arguments concerning source code access and simulator/oracle access. In information theoretic arguments, strong data ciphers produce strings that are indistinguishable from random data.

We consider the question of what properties accompany an encrypted *program*. Unlike encrypted *data*, an executable encrypted *program* must be intelligible to some underlying interpreter or execution engine. We assume that random *programs* follow the rules of an underlying interpretive architecture and that random (combinational) *circuits* follow legal construction rules (current inputs only derive from previous inputs).

In Figure 56, we compare the notion of random *data* generated by a pseudorandom generator and a random *circuit (program)* produced by a pseudorandom *obfuscator*. A strongly pseudorandom number generator, based on some seed, will produce data sequences that are indistinguishable from truly random data sequences. We envision obfuscators that, based on some key/seed information, produce data sequences (specifically, data sequences that are circuit or program descriptions) that are computationally indistinguishable from random circuit (random program) descriptions. We prove shortly that pseudorandom descriptions for both programs and circuits exist.



**Figure 56: Considering Random Data and Random Circuits/Programs**

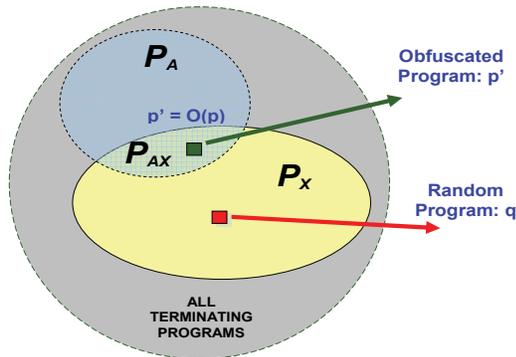
Like computational indistinguishability, unbiased selection is another way to think of randomness. Generally, if we select an element from a population without bias (i.e. each population member is equally likely for selection), that element is a randomly selected element of the population. The element itself is no more “random” than any other element; only the unbiased selection gives the element the random property. More specifically, we can only select a random bit if we can construct an unbiased selection process, where we select 1 and 0 with equal likelihood. Unfortunately, this problem is impossible in practice since we cannot create a “perfect coin”. The theoretic idea that strongly pseudorandom generators exist represents our best (scientific) attempt to produce simulators that provide nearly random bit selection.

We can produce random selections only when we choose without bias absolutely; if those selections are bits, we refer to their conglomeration as a *random bit stream*. Perfect data encryption rests on generating cipher text that is indistinguishable from a random bit stream of the same length. The [accurate] intuition here is that cipher text that closely simulates randomness is unlikely to give away any hints about the corresponding plaintext. We extend that notion to expect that streams with strong randomness properties also have high entropy and low information content. Such random bit streams reveal only confusion under inspection and cryptanalysis. We leverage this paradigm and transfer its notions from *data encryption* (where we protect information secrecy) to *program encryption* (where we protect program intent).

Random *programs* are similar to randomized *data* produced by strong data encryption algorithms. Digitized random data, for example, has no discernible patterns and has typical bit representations where each bit is equally likely to be zero or one. Considering all random data bit strings of size  $n$ , a non-linear, random selection should produce, on average, a string with roughly equal 1s and 0s and no repeating patterns. Similarly, given the infinite set  $P_A$  that contains all programs that implement some functionality  $A$  and the large, but finite set  $P_X$  that contains all programs of length  $X$ , the intersection set  $P_{AX}$  contains all programs that are of size  $X$  and that implement  $A$ . If we randomly choose  $q$  from  $P_X$ , we consider  $q$  to be a *random program*. Figure 57 illustrates the relationship between  $P_X$ ,  $P_A$ ,  $P_{AX}$ , and the selection of a random program  $q$ .

Random selection is only valuable if it provides or ensures entropy in some form. Just as randomness properties for strings (no patterns or lengthy uniform sections, similar number of zero/one, etc.) only emerge as string length increases, so program entropy only emerges as program size increases. Intuitively, there are many *more* ways to write an unrecognizable

program, e.g. to write in unintelligible spaghetti code, than there are to write versions that reveal their intent through static analysis.



**Figure 57: Random Program Selection**

Intuitively, if a maintenance programmer statically inspects two programs that compute the same function (one with more source lines of code than the other), we expect the programmer will need to process more “information” cognitively for the larger program version than they will need to for the smaller version. Of course this makes no assumptions regarding the information content of any one line of source code when compared to another (obfuscated code competitions illustrate the cognitive complexity of compressed code). We state this intuition as Proposition 3 and relate program size with entropy; we provide an initial proof sketch for programs here and in a later section give empirical reasoning to support the exponential growth claim based on circuit constructions. No universal argument for complexity and entropy exist, but Proposition 3 is similar to the notion of Kolmogorov-Chaitin complexity [196, 197], which provides a definition for algorithmic information theory. The theory states loosely that objects are “simple” if they require small quantities of information to describe them and “complex” if they require much. Specifically, the information content of a string increases as the randomness in the string increases and therefore entropy increases as a measure of the randomness of a string. The Kolmogorov metrics, however, remain uncomputable.

**Proposition 3.** Entropy of randomly selected programs increases (exponentially) on the program size.

**Proof Sketch:** Randomness properties emerge as string length increases and therefore as program description length increases. Given two program ensemble families  $P_n = \{P_n\}_{n \in \mathbb{N}}$  and  $P_m = \{P_m\}_{m \in \mathbb{N}}$ , where  $n < m$ , select programs randomly  $p_1$  and  $p_2$  where  $p_1 \in P_n$  and  $p_2 \in P_m$ . Based on Kolmogorov-Chaitin complexity, since the [random] string  $p_1$  is smaller than  $p_2$ ,  $p_2$  has greater *emergent* randomness and therefore more entropy than  $p_1$ .

We also point to researchers like Harrison [198] that define entropy-based metrics for software complexity (based on empirical probability distribution of operators within a program). We do not contend that delivering a *random program* (chosen in the method we describe) guarantees program intent protection; only that an adversary is highly unlikely to discover the intent of a random program through static analysis. Our intuition, however, is that random selection also provides strong dynamic analysis protection, and we give deeper insight into this with the circuit construction methodology in Section 5.6.

Given Proposition 3, we may characterize program encryption strength as its ability to select a program randomly from a set of equivalent, bounded implementations. Classically, such mechanisms are measured based on an adversary’s ability to distinguish an executably encrypted (i.e. randomly selected) program  $p'$  of size  $x$  that implements  $A$  from a random program

$q$  of size  $x$ , that does not implement  $A$ . If the adversary can distinguish  $p'$  from  $q$ , then the obfuscation (program encryption) technique may leak the intent of  $p$ . This leads us to a better (formal) definition for white box obfuscation.

### 5.5.2 Random Program Oracles and White Box Obfuscation

To fully protect intent under white box protection, an obfuscator must systematically confuse  $p'$  so that an adversary cannot learn anything about program intent by analyzing the static code structure or by observing program execution. The confusion must make the code and all possible execution paths that it produces display random program properties. For example, if a sophisticated adversary can distinguish between the functional program and the composite encryption program, they may be able to extract valuable intent information. Definition 6 extends and incorporates white box protection to define full intent protection as preventing all combined means of analysis that discover programmatic intent.

Recall that input/output behavior is the primary means to discover programmatic intent based on black box understandability (already established under Definition 4 and Theorem 1). A secure (white box) obfuscation produces a program  $p'$  that is indistinguishable from a random program selected from the set of all programs the same size as  $p'$ . White box security encompasses both cases. White-box security is the ability to shield program intent from code analysis. Thus, white box protected obfuscations protect against analysis intended to reveal embedded data, seams between functions, the number of functions, or other such program properties. We specify a more formal definition of white box protection in Definition 12 and illustrate the random program oracle interaction in Figure 58.

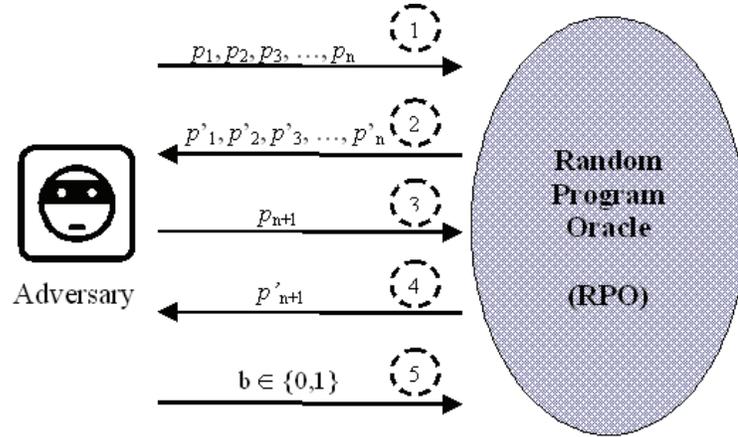


Figure 58: Random Program Oracle

**Definition 12. (White Box Understandable, Formal)** Let PPT algorithm  $E$  be a white box program obfuscator let random program oracle  $RPO$  take any program  $p$  and computes  $p' = E(p)$  as an executably encrypted version of  $p$ . Let PPT algorithm adversary  $A$  be able to query  $RPO$  and receive any encrypted program  $p'_x$  in the following manner: after knowing any  $n$  pairs of original / executably encrypted program pairs  $\{(p_1, p'_1), (p_2, p'_2), \dots, (p_{n-1}, p'_{n-1}), (p_n, p'_n)\}$ , adversary  $A$  supplies a subsequent program  $p_{n+1}$  and receives  $p'_{n+1}$  from  $RPO$  which is either: a random program ( $P_R$ ) or the encrypted version of the program  $p'_{n+1} = E(p_{n+1})$

$$p'_{n+1} = \begin{cases} P_R & \Pr[p'_{n+1} = P_R] \leq \frac{1}{2} + \varepsilon \\ E(p_{n+1}) & \Pr[p'_{n+1} = E(p_{n+1})] \leq \frac{1}{2} + \varepsilon \end{cases}$$

The program  $p'_{n+1}$  is white box understandable if the probability is greater than or equal to  $\frac{1}{2} + \epsilon$  (where  $\epsilon$  is the negligible error probability), that the adversary  $A$  is able to distinguish whether  $p'_{n+1}$  is either  $E(p_{n+1})$  or is a random program  $P_R$ . Therefore, the program encryption algorithm  $E(p)$  provides white box obfuscation if and only an adversary is able to distinguish the encrypted program ( $p'_{n+1}$ ) from a random program ( $P_R$ ) with probability less than or equal to  $\frac{1}{2} + \epsilon$ , where  $\epsilon$  is the negligible error probability.

We define formally white box strength for executably encrypted programs based on the existence of a random program oracle. We demonstrate the existence of random programs with a contrived architecture in Section 5.5.3 and give two algorithms for constructing random circuits in Section 5.5.4. We assume the existence of a random program oracle because of the general existence of a random oracle that simulates creation of random data strings. If we can simulate the creation of random data strings (with a pseudorandom generator), then we can simulate the creation of random programs.

Figure 58 illustrates our model where the oracle performs two functions: when given a program, it can generate an encrypted version of that program based on a predefined algorithm  $E(p)$  or it can generate a random program  $P_R$  from the set of all random programs with polynomial description size to  $p$ . The adversary sends an original program  $p$  to the oracle and the oracle executably encrypts using the underlying algorithm,  $E$ . The algorithm can be any tamperproofing, obfuscation, or piracy prevention mechanism. The oracle returns the corresponding encrypted program  $p' = E(p)$ . As we depict in via the top two arrows labeled Figure 58-1 and Figure 58-2, the adversary may build some polynomial history of  $n$  pairs. After building the history, Figure 58-3 shows that the adversary then sends program  $p_{n+1}$  to the oracle and the oracle then returns  $p'_{n+1}$  (Figure 58-4).

The adversary must decide whether the program  $p'_{n+1}$  given by the oracle is the encrypted version of program  $p'_{n+1} = E(p_{n+1})$  or if it is a random program  $P_R$ . The adversary attempts to make a prediction by returning bit  $b \in \{0, 1\}$  corresponding to the guess of either  $P_R$  or  $p'_{n+1}$  as shown by Figure 58-5. Under this definition, we measure the security strength of *any* obfuscation approach as a computational indistinguishability question based on random programs and a random program oracle model. We now define the nature of the random program,  $P_R$ , generated by the random program oracle and demonstrate their existence.

### 5.5.3 Proving Random Programs Exist

To review, the canonical notion of a *random*  $x$  is that of a randomly selected member of a target space of items, say group  $X$ . The problem of whether or not a random  $x$  exists reduces to 1) the ability to define the group  $X$  and 2) creating a method to select one item randomly from it without bias. The mechanism must select items with an equal probability.

**Proposition 4.** Random programs exist.

We state in Proposition 4 our belief that random programs exist and give proof in the following text. We establish first a well-defined set  $\mathbb{P}$  that is the set of all legal programs and then propose mechanisms for randomly selecting a legal program from  $\mathbb{P}$ .

**Legal Programs.** Selecting a random program is impossible without knowing the maximum program length, since it is impossible to select randomly an item from an infinite set. We first bound the program length to  $n$  statements, words, bits, or any other meaningful metric that bounds program size at  $n$  units. Choosing bits as our metric, there are  $2^n$  possible programs and we let  $\mathcal{P}_n$  represent the function ensemble of programs of length  $n$  bits or less.

To select without bias, we must ensure that we can identify *legal* programs, with many related considerations towards legality. For example, a legal program is syntactically and grammatically correct. The selection mechanism must guarantee that we do not select programs with illegal

symbols or illegally constructed words or phrases, if our programming language allows such constructs. Parsers and compilers routinely facilitate this filtering process.

A second illegality category includes programs with runtime failures, e.g. dividing by zero. Identifying runtime flaws is a difficult problem; if we could completely solve it, the software engineering field would be essentially obsolete. We can solve this problem for very simple architectures and go on to propose heuristics for dealing with the dilemma in real world applications. Similarly, legal programs must terminate on all input. Termination may be dependent on the underlying interpreter or environment and can range from reaching the last program statement, executing a HALT instruction, reaching a final state, or reaching a maximum number of instructions. While the general halting problem is well beyond our scope, we again propose a solution in a simple environment. We address all of these problems more simply under the circuit model, discussed in the following subsection, because legal circuit descriptions are much easier to produce.

Table 15 summarizes legal program characteristics (for our purposes) from which we can make some simple observations. For example, there are  $2^n$  possible programs that are  $n$  bits long. For a given architecture, we count the number of programs that are illegal in each category ( $h, i, j, k$ ) respectively and assign  $m = h+i+j+k$  as the possible number of illegal programs. We therefore know  $2^n - m$  legal programs exist whose length is less than or equal to  $n$ . We assume that categories are mutually exclusive such that all grammatically incorrect programs are syntactically correct, all programs with runtime failures are syntactically and grammatically correct, and so forth.

**Table 14: Legal Program Considerations**

Legality Category	Metric
$h$ : Syntactically correct	Lexical Analysis
$i$ : Grammatically correct	Parsing
$j$ : No runtime failure	Testing/verification
$k$ : Halts	Proof

We have not shown how to identify all of these categories for *any* architecture yet, but we can show them for specific architectures under specific rules. Moreover, for the method to work, we must guarantee that selection considers *every* possible legal program with equal likelihood. In this case, any PPT algorithm  $D$  that decides legality of a program  $x$ , where  $x \in P_n$ , has probability of success  $Pr[D(x, 1^n)=1] = (2^n - m)^{-1}$ . In the next section, we demonstrate random program selection that meets these criteria and give a concrete example using a contrived program space termed the *Ten-Bit Instruction Architecture*.

**Random Bit Stream Programs.** We address first the belief stated in Proposition 4: *do random programs exist?* As we establish in the preceding sections, we consider digitized programs as data with special syntactic rules governing their construction. *Random programs* are indistinguishable from a random bit stream, i.e. that they have no discernable bit patterns, each bit is equally likely to be zero or one, and any sub-string of any reasonable length has approximately the same number of zero's as it has ones.

We illustrate this notion with an abstract machine (depicted in Figure 59) that uses a saturated instruction space. Our machine consists of four operations (instructions), sixteen four-bit registers, operations defined with two-four bit operands, and ten-bit instructions (Table 15). The contents of the registers upon program termination reflect the program output. In this architecture, any bit stream whose length is divisible by ten represents a legal program.

**Table 15: Ten Bit Instruction Set Architecture (TBIA)**

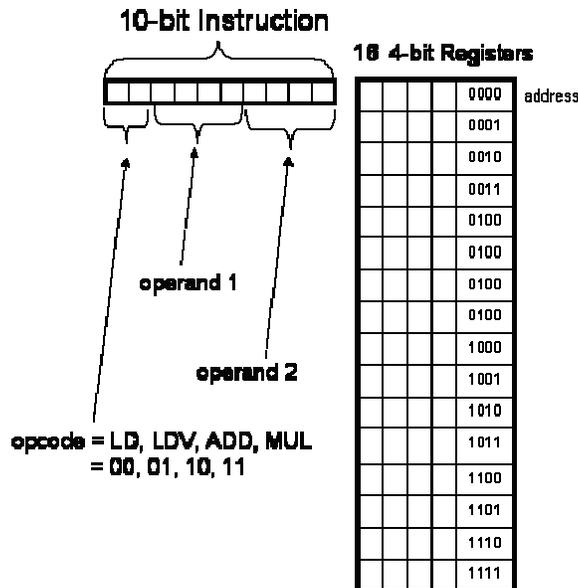
Op	Opnd 1	Opnd 2	Description
LD	<i>Rega</i>	<i>Regb</i>	Copy values fm regb to rega
LDV	<i>Regb</i>	<i>Opnd</i>	Copy values fm opnd to regb
ADD	<i>Rega</i>	<i>Regb</i>	Add regs a&b, trunc result in rega
MUL	<i>Rega</i>	<i>Regb</i>	Mult regs a&b, trunc result in rega

**Theorem 2.** Random programs exist in the Ten Bit Instruction Architecture (TBIA). Therefore, random programs exist.

**Proof.** Using a strongly pseudorandom generator with appropriate seed, generate  $p'$  as a random bit stream of length  $10 \cdot c$ , where  $c$  is a large constant. Then with instructions interpreted serially from beginning to end,  $p'$  is a random program in TBIA.

1.  **$p'$  is a legal program.** It is syntactically and grammatically correct, since
  - (a) There are no illegal instructions, therefore no syntactic or grammatical errors exist
  - (b) There are no vulnerable instruction sequences, therefore no execution failures exist
  - (c) There are no loops and programs are finite, therefore all programs halt
  - (d) The architecture can compute meaningful programs
  
2.  **$p'$  is a random selection from  $P_{10c}$ .** A strongly pseudorandom generator that produces a string  $\{0, 1\}^{10 \cdot c}$ , where  $c$  is a large constant, selects a string with equal probability from the ensemble consisting of all bit strings of size  $10 \cdot c$ . Since we represent the ensemble by the TBIA program set  $P_{10c}$ ,  $p'$  is an unbiased random choice with equal selection probability from all other programs in the ensemble.
  
3.  **$p'$  is a random string.** In every reasonable sense of the term, assuming the strong pseudorandomness of the generator,  $p'$  has no discernible pattern in:
  - (a) Static representation (otherwise, it would not be a random bit stream)
  - (b) Data representation
  - (c) Control flow

**QED,  $p'$  is a random program.**



**Figure 59: TBIA Machine Depiction**

**Corollary 1.** Assuming one-way permutations exist, pseudorandom *data* generators exist. Then, if one-way permutations exist and if random programs exist, pseudorandom *program* generators exist also.

We extend Theorem 2 by stating what many cryptographers such as Goldreich [188] assume to be true: if one-way permutations exist, we know that strongly pseudorandom *data* generators exist. By proving the existence of random programs, we also prove that if one-way permutations exist, pseudorandom *program* generators exist under the same assumptions. This again provides a much stronger cryptographic basis for which to define white box obfuscation under. It relies on well-known cryptographic primitives and proofs—and assumes nothing about cognitive or mental ability of the adversary.

**Composing Random Programs.** We plan to extend these results to more functional architectures in the future. In order to do so, it may be possible to use random program composition. Here we pose a second question: *can we create random programs from other random programs?* We conjecture that composition (concatenation) of two random programs always produces a random program, but we note that recursive composition produces patterns (possibly repeated segments). However, the program resulting from concatenation of atomic (independent) random segments is random.

**Random Instruction Selection Programs.** TBIA concretely illustrates that random programs exist. We now extend this notion to a more complex machine where the instruction space is not saturated. For example, we extend TBIA to include a fifth operator with a shift operator (SFT) that shifts the value in *Rega* left one bit and stores the result in *Regb*, as shown in Table 16.

**Table 16: Modified TBIA with New Instruction**

Op	Opnd 1	Opnd 2	Description
LD	<i>Rega</i>	<i>Regb</i>	Copy values fm regb to rega
LDV	<i>Regb</i>	<i>Opnd</i>	Copy values fm opnd to regb
ADD	<i>Rega</i>	<i>Regb</i>	Add regs a&b, trunc result in rega
MUL	<i>Rega</i>	<i>Regb</i>	Mult regs a&b, trunc result in rega
SFT	<i>Rega</i>	<i>Regb</i>	Shft Rega left 1 bit-> Store Regb

To accommodate the additional instruction, we increase the operator length to three bits. Thus, a random bit stream interpreted as a program in TBIA may contain *illegal* instructions. To address architectures where the operator space is not saturated, we may think of a random program as having the operators equally distributed across the program. In this scenario, we generate a random program by randomly selecting each operator from all possible operators and similarly selecting the operands. Programs generated in this way have random properties similar to those in TBIA, such as having a similar count of each instruction type, no patterns among operands, and no observable patterns between instructions. During execution, the data and control flow reflect the random properties of the instructions.

The examples in TBIA and its extension clearly illustrate that our model need not be complex or sophisticated to allow random programs. We consider next more sophisticated random program versions. We use random selection with TBIA to produce random programs. Such random selection allows a systematic way to generate random programs that avoid illegal instructions. We discuss the ramifications for more sophisticated architectures by considering different program representation schemes .

**Random Function Selection Programs (RFSP).** To extend the notion of random programs beyond the simple architecture of TBIA, we consider a random program as simply a collection of higher-level structures, composed with no discernable pattern or plan. A large library of random program segments (random programs which themselves may be incorporated unmodified into other random programs) can provide the possibility for composition. We can compose<sup>12</sup> selected

<sup>12</sup> In TBIA, composition consists of concatenation. We recognize the administrative actions necessary in higher level languages and posit that these are well understood and that segment compatibility issues are overcome easily.

segments to create another, larger random program, but it remains unclear which randomness properties we preserve in the composition.

Consider, for example, the one-bit architecture depicted by the four operations in Figure 60. In a simple architecture like TBIA, we can recognize usable patterns in segments even when they derive from random creation themselves. In a one-bit architecture, we define the functionality of every segment as one of the four operations in Figure 60. Thus, by purposefully selecting segments, the composition may not be random or may even have a usable function with obvious pattern. This concern diminishes rapidly as the architectural complexity increases, since randomly generated segments are less likely to have a usable, recognizable function. Still, we may also increase the confusion of generated RFSPs by governing segment selection.

1. 0→0, 1→0 2. 0→1, 1→0 3. 0→0, 1→1 4. 0→1, 1→1
--

**Figure 60: Simple One-Bit Architecture**

We retain reference to TBIA because it is sufficiently simple to illustrate our concepts, yet complex enough to give a flavor of its strength. Given a large constant  $cl$  (e.g.,  $cl > 100$ ) a small constant  $cs$  (e.g.,  $10 < cs < 30$ ), and an integer  $l$ , the following algorithm will generate RFSPs of length  $l * cs$  statements.

<ol style="list-style-type: none"> <li>1. Generate <math>(cl * cs)</math> random statements.</li> <li>2. Partition the statements into <math>cl</math> random segments of length <math>cs</math>. Number the segments from one to <math>cl</math>.</li> <li>3. Create a program <math>p</math> by randomly selecting <math>l</math> segments (without replacement) and concatenate them.</li> </ol> <p>Then, <math>p</math> is an RFSP.</p>
---

By virtue of the properties proved in Theorem 2, we can claim  $p$  is a random program. Were replacement allowed, it would be possible to include the same segment more than once, resulting in a discernable pattern and diluting  $p$ 's randomness. However, we observe that, because of the random construction, these patterns reveal very little about the program. This is easily seen if consider each segment as a named subroutine and replace each segment with its name and arguments to create  $p'$ . Then  $p'$  is a random program, since the repeated subroutines are randomly placed. A final extension of this notion is to consider randomly composing non-random segments. Clearly, this injects patterns into the code. Again, if we name and replace each of the segments in  $p$  with their name,  $p$  is a random program.

**Random Turing Machines.** Random programs may also be Turing machines. Consider a Turing machine  $T = \{Q, \Gamma, S, b, F, \delta\}$  where:

<p> <math>Q</math> is a finite set of states  <math>\Gamma</math> is a finite set of the tape alphabet  <math>S \in Q</math> is the initial state  <math>b \in \Gamma</math> is the blank symbol  <math>F \subseteq Q</math> is the set of final or accepting states  <math>\delta</math> is the transition function: <math>Q \times \Gamma^k \rightarrow Q \times (\Gamma \times \{L, R, S\})^k</math> </p>
--

Following our model, we construct a random Turing machine  $t$  using the following algorithm:

1. Randomly select a small number of states and number them 1- $i$ .
2. Similarly, select a small alphabet numbered 1- $j$ .
3. Randomly select the start state from the state space.
4. Define the transition function as follows: for each state and each alphabet member, randomly select:
  - a) A head movement from  $\{R, L\}$
  - b) An operation from  $\{\text{write, none}\}$
  - c) An alphabet element to write
  - d) A state to transition to

Then  $t$  is a random Turing machine.

By these constructions, we illustrate a consistent theme concerning random programs: each type of randomness has discernable properties, just like random (data) bit streams. The more we know about random program properties, the more likely we will be able to generate intentioned programs that reflect random program properties. We believe the random program security model represents a (better) theoretical basis for analyzing obfuscators that rely on complexity and confusion. The underlying tenet for white box security found in Definition 12 is the first model (that we know of) to consider protection based on cryptographic properties (as opposed to known hard NP-complete problems that assume complex adversarial workload).

#### 5.5.4 Proving Random Circuits Exist

The term “program” is less precise than traditional TM definitions. Therefore, we predominantly use combinational circuits to describe obfuscators that provide white box protection using randomization in Section 5.6. We introduce here the notion that random circuits, like random programs, exist. We discuss first our circuit description nomenclature, define a bit string language to construct circuits, and then give support for Proposition 5 that random circuits exist by elaborating three separate algorithms for their construction.

**Proposition 5.** Random circuits exist.

Circuits provide an alternative to Turing machines for considering computational complexity and defining functional operations. Literature abounds with references to circuit and complexity relationships, and we mention several known results that are detailed further in textbooks such as Wegener’s [199]. We can simulate the computation of a Turing machine  $M$  on inputs having length  $n$  with a single  $n$ -input circuit with size  $O((|M| + n + t(n))^2)$ .  $t(n)$  defines the bound on the running time of  $M$  in inputs of length  $n$ . Thus, a non-uniform family of polynomial-size circuits can simulate a non-uniform sequence of polynomial-time machines. Likewise, a non-uniform sequence of polynomial-time machines can simulate a non-uniform family of polynomial-size circuits. Machines with polynomial description lengths may integrate polynomial-size circuits and simulate their computations in polynomial (bounded) time.

There are several advantages for using circuit representations. Circuits only have one polynomial representation space (which is their size), while Turing machines have two (for their size and for their running time). Turing machines are uniform concerning input whereas circuits are non-uniform because each different input length may have a different associated circuit (gate). It is possible to show that we can construct all (physical) machines with bounded memory via (sequential) circuits and binary memory units. We can completely simulate machines whose computations terminate with circuits.

**Combinational Circuit Families.** In digital circuit theory, combinatorial or combinational logic represents circuits whose output is a function of only the present input. *Sequential* logic has

output that depends not only on the present input but also on historical input. Sequential logic supports memory while combinatorial circuits do not. Combinational circuits (whose gate values depend only on its current signals from any previous gate) can represent any straight-line programs, meaning those with no loops or branches. However, we can compute many important functions in a straight-line manner while conveniently describing their functionality as a circuit or directed acyclic graph.

Physical computer circuits normally contain a mixture of the two logic modes. For example, ALU components that perform mathematical calculations are typically combinatorial while the control signals for the ALU require sequential logic. At its lowest level, a computer is represented as (lots) of Boolean circuit combinations. However, hardware relies on synchronous time signals and clock signals to direct their activity.

We discuss families of Boolean circuits according to a set of common features that they share. We let  $C_{nms\Omega}$  indicate the set of all circuits with the same input size ( $n$ ), output size ( $m$ ), circuit size ( $s$ ), and basis ( $\Omega$ ). A circuit over  $\Omega$  is a directed acyclic graph (DAG) having either *nodes* mapping to functions in  $\Omega$  (*gates*) or having *nodes* with in-degree 0 being termed *inputs*. We also distinguish one (or more) as *outputs*. The basis  $\Omega$  is *complete* if and only if all functions  $f$  are computable by a circuit over  $\Omega$ . The basis sets {AND, OR, NOT}, {AND, NOT}, {OR, NOT}, {NAND}, and {NOR} are all known to be complete. By definition, the 6-gate basis  $\Omega = \{\text{AND, OR, NOR, NAND, XOR, NXOR}\}$  is complete and has basis size  $|\Omega| = 6$ .

Circuits that implement the same Boolean function,  $f: \{0,1\}^n \rightarrow \{0,1\}^m$ , must have the same input size ( $n$ ) and output size ( $m$ ), although a larger (padded) input size  $n'$  is possible as long as  $n' > n$ . Functionally equivalent circuits may differ according to size ( $s$ ) and basis ( $\Omega$ ). However, they share common characteristics (depending on the terminology you wish to use) such as having the same signature (truth table output columns), the same truth table representation, equivalent fully minimized Boolean formulae, and equivalent input/output mappings.

**Circuit Description Languages.** There are three ways of referring to circuit description languages: *textual description languages*, *graphical representations*, and *binary representations*. Because our interest is ultimately pseudorandom bit generation, the binary representation is more suitable for demonstrating provable white box security properties. We discuss the first two forms because they also apply to our research implementation efforts.

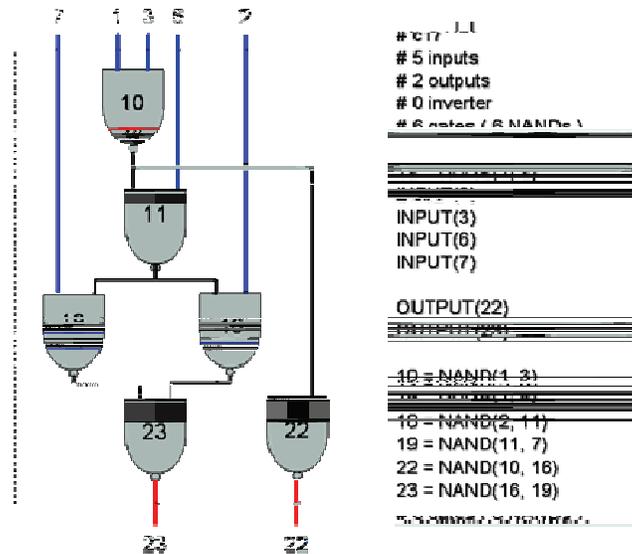
Textual description languages are similar to programming languages (whether machine level, assembly level, or high level). Textual circuit representation languages are regular grammars with syntactic rules for construction. Their syntax supports expression of gates, electrical signals, components, and gate interconnections. Over time, organizations like IEEE have developed libraries of standardized circuit definitions that support application testing, algorithm analysis, and integrated circuit optimization [200]. Researchers used a conference-style approach to develop and review the ISCAS (International Symposium of Circuits and Systems) and ITC [201] (International Technical Conference on Circuits) benchmark sets. Benchmark circuits come in many different textual formats as well<sup>13</sup>. BENCH, CKT, VERILOG, and VHDL are several just to name a few. Most formats typically reflect complex hardware components and languages such as VERILOG / VHDL facilitate direct hardware synthesis. The BENCH format is very close to a true Boolean circuit definition and we adopt this as textual description language in our (MASCOT) implementation architecture described further in Section 5.8 and Appendix D/E.

Finding direct translators of high-level language (HLL) to circuit representation is hard outside of government<sup>14</sup> or commercial applications [202] designed for digital circuit design. In order to facilitate our research and implementation activities, we identified existing definitions for combinatorial Boolean circuits with well-known functionality instead of attempting to convert from HLL to circuit representation directly. We use the ISCAS benchmarks in our implementation activities because they give us known functionality to start with and provide readily available Boolean logic. For future work, we plan to devote attention to the *HLL-to-Boolean-logic* conversion problem.

<sup>13</sup> ISCAS-85, ISCAS-89, ISCAS-99 circuits available at <http://www.fm.vslib.cz-kes/asic/iscas/> in 6 textual forms

<sup>14</sup> Wright Laboratories contracted a C-to-VHDL translator under BAA, <http://www.stormingmedia.us/51/5170/A517013.html>

We prefer to illustrate the notion of randomization for white box protection using pre-existing circuit definitions where possible and we conclude from our current work that it is much easier to work with known functionality than to convert high-level programs into Boolean logic. The ISCAS-85 set of benchmark circuits provide a rich source of both known functionality and research interest [203, 204]. Figure 61 illustrates a BENCH (textual) specification and schematic (graphical) diagram of the ISCAS-85  $C_{17}$  circuit. The ISCAS-85 benchmarks include definitions for functions such as a 27-channel interrupt controller, a 32-bit SEC circuit, an 8-bit ALU, a 16-bit SEC/DED circuit, a 12-bit ALU/controller, an 8-bit ALU, a 16x16 multiplier, and a 32-bit adder/comparator just to name a few. Appendix E describes both the ISCAS benchmark circuits and the BENCH circuit representation language in detail.



**Figure 61: The ISCAS-85  $C_{17}$  Benchmark Circuit in BENCH Notation**

Combinational circuits in BENCH notation represent textual circuit descriptions. Figure 61 represents the other (traditional) graphical representation form as a collection of binary Boolean gates connected with wires. We can also use directed acyclic graphs to represent equivalent circuit information as long as we associate gate types with each node. Formally analyzing circuit designs is a known hard problem that has led researchers to produce efficient, graphical representations for Boolean circuits that facilitate verification. Solving constraint satisfaction problems and formal verification have been catalyst to a myriad of graphical structures that support graph-based Boolean function manipulation: Binary Decision Diagrams (BDD), Reduce Ordered Binary Decision Diagrams (ROBDD), FDD, OBDD, ADD, MTBDD, BMD, KMDD, and BGD to name a few [205, 206, 207].

Boolean Expression Diagrams (BEDs), another extension to BDDs, represent any Boolean function in *linear* space and provide standard graph-based tools for dealing with combinational-level logic problems [204]. BEDs have been useful for efficiently determining whether two combinational circuits implement the same Boolean function [208] and come with several desirable features practical to our current work. Figure 62 shows the graphical BED-based definition for the  $C_{17}$  benchmark circuit described earlier in Figure 61. The creators of the BED library provide an ability to generate DOT-based<sup>15</sup> graphical notations for circuits, which we utilize for viewing circuit definitions extensively. Appendix F describes in detail how we integrate BEDs into our current implementation activities and gives several illustrative examples of BED diagrams for the ISCAS-85 circuit benchmarks.

<sup>15</sup> IBM AT&T Research Labs, <http://citeseer.ist.psu.edu/331854.html>

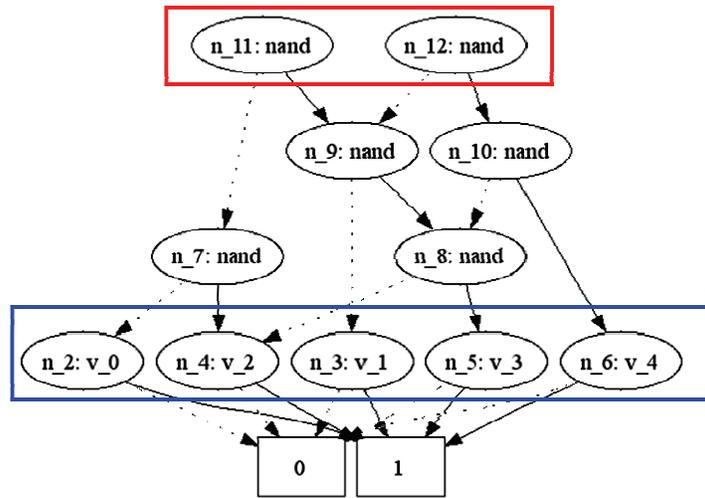


Figure 62: BED Definition of ISCAS-85 C<sub>17</sub>

The canonical theoretic representation for a binary circuit is a binary string. We envision circuit randomization techniques in Section 5.6 that manipulate binary circuit representations using traditional cryptographic cipher primitives such as confusion and diffusion. We describe our method for annotating Boolean logic circuits as binary strings here. In canonical representation, we can view a circuit  $C: \{0, 1\}^n \rightarrow \{0, 1\}^m$  with  $n$  inputs and  $m$  outputs as a collection of its  $(m)$  output subcircuits  $C_1, C_2, \dots, C_m$ . For all subcircuits  $C_i, 1 \leq i \leq m$ , and each subcircuit  $C_i$  corresponds to exactly the  $i^{\text{th}}$  output of circuit  $C$ . The canonical sum-of-products form expresses each output subcircuits  $C_i$  as a collection of minterm products related to each possible input of the circuit.

We refer to a circuit *signature* as the collection of truth table values associated with an output gate (i.e., subcircuit  $C_i$ ) corresponding to the canonical ordering of input values. For two input and one output gate, the signature for the gate corresponds to the inputs pairs  $(0,0), (0,1), (1,0), (1,1)$  and we represent the corresponding signature as  $[\{0, 1\}_1, \{0, 1\}_2, \{0, 1\}_3, \{0, 1\}_4]$  where  $G: (0,0) \rightarrow \{0, 1\}_1, G: (0,1) \rightarrow \{0, 1\}_2, G: (1,0) \rightarrow \{0, 1\}_3,$  and  $G: (1,1) \rightarrow \{0, 1\}_4$ . Figure 63 gives two examples of such signatures: one for a 4-input/1-output circuit and the other for a 3-input/2-output circuit.

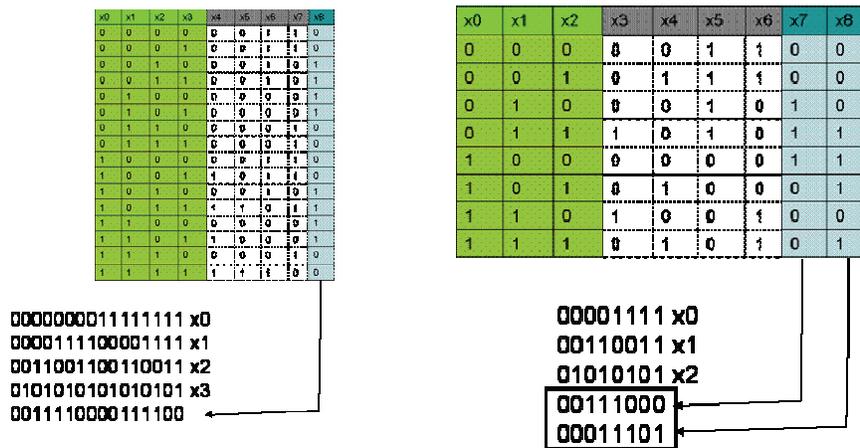


Figure 63: Examples of Circuit Signatures

For Boolean circuits over  $\Omega_2$  (which define 16 possible 2-input Boolean gates), every gate in a circuit  $C$  has function  $G: \{0,1\} \times \{0,1\} \rightarrow \{0,1\}$ . Table 17 lists the full set of gates under  $\Omega_2$  and the corresponding symbol which we use to identify traditional Boolean circuit gates types (AND, OR, XOR, NOR, NAND, NXOR). For the function values, we let  $x'$  indicate the negation of Boolean variable  $x$ ,  $\wedge$  indicate the logical AND,  $\vee$  indicate the logical OR, and  $\oplus$  indicate the logical XOR. As long as we have a complete basis  $\Omega$ , we can generate all functions over that basis and we do not require all 16 (gate) types to enumerate circuits within a functional family.

**Table 17: Gate Definitions Under  $\Omega_2$**

$G(x_1, x_2)$	Symbol	Signature	$G(x_1, x_2)$	Symbol	Signature
0	const 0	[0,0,0,0]	$x_1' \wedge x_2'$	NOR	[1,0,0,0]
$x_1 \wedge x_2$	AND	[0,0,0,1]	$(x_1 \oplus x_2)'$	NXOR	[1,0,0,1]
$x_1 \wedge x_2'$		[0,0,1,0]	$x_2'$		[1,0,1,0]
$x_1$		[0,0,1,1]	$x_1 \vee x_2'$		[1,0,1,1]
$x_1' \wedge x_2$		[0,1,0,0]	$x_1'$		[1,1,0,0]
$x_2$		[0,1,0,1]	$x_1' \vee x_2$		[1,1,0,1]
$x_1 \oplus x_2$	XOR	[0,1,1,0]	$x_1' \vee x_2'$	NAND	[1,1,1,0]
$x_1 \vee x_2$	OR	[0,1,1,1]	1	const 1	[1,1,1,1]

Recalling our circuit family definition,  $C_{nms\Omega}$ , we can give the encoding of any circuit  $C \in C_{nms\Omega}$  by a set of gates  $W = \{w_1, w_2, \dots, w_{n+s}\}$  set depicted in Table 18. This set includes  $n$  inputs,  $s$  total gates,  $s - m$  intermediate gates, and  $m$  output gates.

**Table 18: Circuit Encoding for Family  $C_{nms\Omega}$**

Structure	Size	Encoding
Inputs	$n$	$w_1, w_2, \dots, w_n$
Gates (intermediate)	$s - m$	$w_{n+1} = G_{n+1}(w_{x_1^1}, w_{x_2^1})$ $w_{n+2} = G_{n+2}(w_{x_1^2}, w_{x_2^2})$ ... ... $w_{n+s-m} = G_{n+s-m}(w_{x_1^{s-m}}, w_{x_2^{s-m}})$
Gates (output)	$m$	$w_{n+s-m+1} = G_{n+s-m+1}(w_{x_1^{s-m+1}}, w_{x_2^{s-m+1}})$ ... $w_{n+s} = G_{n+s}(w_{x_1^s}, w_{x_2^s})$

For all gates  $G_i$ , we assume  $x_1^i \leq x_2^i < n + i$  for the corresponding inputs  $w_{x_1^i}$  and  $w_{x_2^i}$ , for all  $1 \leq i \leq s$ . This states that gate inputs can be the same and insures us that gate inputs can only come from previously computed gates (thus guaranteeing the acyclic nature of the circuit). To characterize the binary string representation of a circuit, we need only replace the textual encoding for the circuit with a binary equivalent form (much as we would translate assembly language into binary). For example, given  $\Omega = \{NAND, NOR\}$ , we can represent the gate type with one bit. For  $n = 8$  inputs, we can represent each input with three bits. For  $s = 25$  gates, we can represent each gate with the upper bound logarithm on  $s$ , which is five bits. Table 19 summarizes the computations for determining the total binary string size of any circuit with  $n$  inputs,  $s$  gates, and basis  $\Omega$ . Definition 13 incorporates these computations as a point of reference.

**Definition 13. (Binary Size for Circuit Descriptions)** Given a circuit  $C$  of size  $s$  with input size  $n$  and basis  $\Omega$ , the upper bound size for a binary string representing the circuit is given by:

$$n \lceil \lg(n) \rceil + 3s \lceil \lg(s) \rceil + s \lceil \lg(|\Omega|) \rceil$$

**Table 19: Binary Size Representation for Circuit Encoding**

Representation	Number	Bit Size
Enumerate Each Input ( $w_i$ )	$n$	$\lceil \lg(n) \rceil$
Enumerate Each Gate ( $w_i$ )	$s$	$\lceil \lg(s) \rceil$
Enumerate Inputs for Each Gate ( $x1_i, x2_i$ )	2	$2\lceil \lg(s) \rceil$
Enumerate Function Type for each Gate ( $G_i$ )	$ \Omega $	$\lceil \lg( \Omega ) \rceil$
Representation	Bit Size	
All Inputs	$n \lceil \lg(n) \rceil$	
Each Gate	Gate ID: $\lceil \lg(s) \rceil$ Input ID: $2\lceil \lg(s) \rceil$ Function: $\lceil \lg( \Omega ) \rceil$ Total: $3\lceil \lg(s) \rceil + \lceil \lg( \Omega ) \rceil$	
All Gates	$s(3\lceil \lg(s) \rceil + \lceil \lg( \Omega ) \rceil)$	
Entire Circuit = All Inputs + All Gates	$n \lceil \lg(n) \rceil + 3s\lceil \lg(s) \rceil + s\lceil \lg( \Omega ) \rceil$	

The only other consider for circuit representation is its signature. We find it useful to classify circuits according to their function family (versus their representation size) and we use the smallest succinct (truth table) embodiment to do so. As we have described previously (see Figure 63), a circuit with  $m$  outputs will have a signature size  $2^n$  bits corresponding to the  $2^n$  possible input combinations that produce each 1-bit output of the signature. We assume a canonical ordering of inputs that reflects directly in the signature. Representing a (full) signature thus requires  $2^n \cdot m$  bits.

**Enumerating Circuit Descriptions.** We specify how to represent a single circuit as a binary string and how to characterize its size. We now consider how to characterize the number of possible circuit descriptions that are contained in a set of circuits with a specific size and basis. In Appendix C, we give full exposition for circuit enumeration possibilities and give only the summarized representation for number in Definition 14.

**Definition 14. (Total Number of Circuit Enumeration Possibilities)** Given a circuit  $C$  of size  $s$  with input size  $n$ , the number of possible  $s$ -gate circuits  $G_C$  possible under basis  $\Omega$  (assuming gates can have identical inputs) is given by the following product:

$$G_C = \prod_{i=1}^s C(n+i, 2) * |\Omega|$$

See Appendix C for the entire circuit enumeration possibilities.

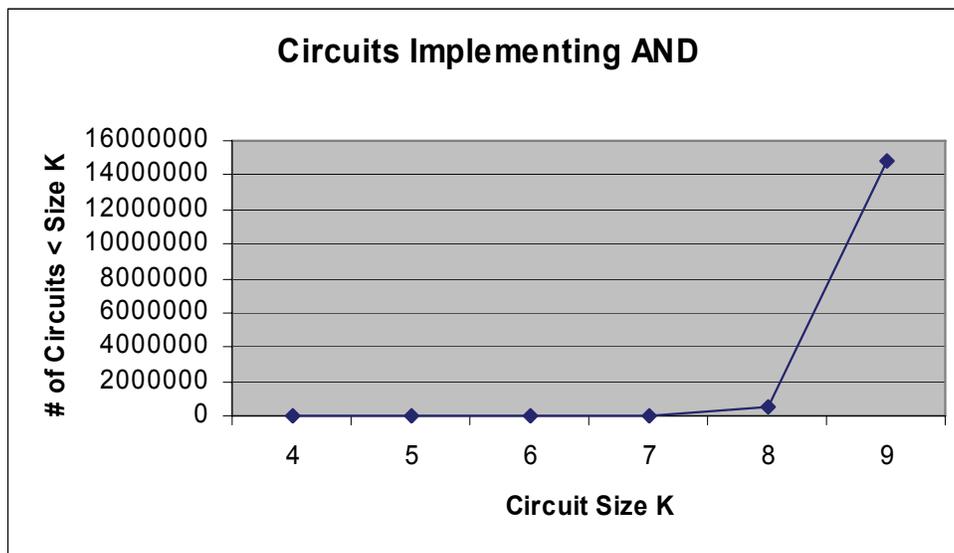
**Entropy and Circuit Size.** Given that we can specify a textual circuit description as a binary string and given that we know how many circuits we can generate based upon a given circuit size and basis, we can now characterize entropy as it relates to circuit size (from Proposition 3). Consider the set  $C_{nms\Omega}$  such that  $n = 2, m = 1, m \leq s$ , and  $\Omega = \{\text{AND, OR, XOR, XNOR, NOR, NAND}\}$ . Assuming we know the basis, we refer to this circuit family ensemble as  $C_{2,1,s}$ . Given this circuit family set, we consider the subset of circuits within  $C_{2,1,s}$  that implement the AND function or, in other words, have a signature of  $[0,0,0,1]$ . At a minimum,  $C_{2,1,s}$  contains circuits of (total) size  $K = 3$  or above, where total size  $K$  is the number of edges in a circuit DAG ( $K = \text{inputs} + \text{gates} = n + s$ ). We can enumerate all node arrangement possibilities and with  $K$  or fewer edges and determine which circuits have the characteristic AND signature.

We let  $C'_{n+s}$  represent the subset of all circuits in  $C_{2,1,s}$  that implement function and let  $|C'_{n+s}|$  indicate the cardinality of the set (the number circuits with AND functionality). Through experimentation, we generate a circuit family for various gate sizes ( $C_{2,1,1}, C_{2,1,2}, C_{2,1,3}, C_{2,1,4}, \dots$ ) and count the number of circuit representations that produce the characteristic  $[0,0,0,1]$  signature. In doing such experimentation, we demonstrate exponential blowup in the possible number of AND function representations as  $K$  increases. For example, when  $K = 4$ , there are 66 total possible circuit combinations (circuits of total size 4 composed of any legal combination basis gates) and three of these circuits have signature  $[0,0,0,1]$ :

$ C'_4  = 3$  $C'_4 = \{$ $(x_0, x_1, x_2=x_1 \text{ AND } x_0),$ $(x_0, x_1, x_2=x_1 \text{ XNOR } x_0, x_3=x_2 \text{ AND } x_1),$ $(x_0, x_1, x_2=x_1 \text{ XNOR } x_0, x_3=x_2 \text{ AND } x_0)$ $\}$	$s = 1, K = 3$ $s = 2, K = 4$ $s = 2, K = 4$
---	--

Figure 64 shows our observed increase as follows:

- $|C'_5| = 81 (K=5);$
- $|C'_6| = 81971 (K=6);$
- $|C'_7| = 8122,881 (K=7);$
- $|C'_8| = 581,203 (K=8);$
- $|C'_9| = 14,793,117 (K=9).$



**Figure 64: Exponential Blowup of Functional Representation**

The set  $C'_9$  of circuits total size 9 or less which implement AND functionality contains nearly 15 million circuits. Any *random* selection from this set gives a circuit with equivalent logical AND functionality. Because we represent larger circuits with larger binary strings (by Definition 13), we show empirical evidence here that entropy of circuits increases exponentially with size, given the

same functionality. This tends to support also our conjecture under Proposition 3 regarding program size and entropy. This illustrates that for complex functionality, the number of circuits implementing that functionality is large, but we can provide a bound on circuit size to keep numbers within (efficient) manageable reach. Assuming a linear increase to the circuit size, we can also increase the complexity or understandability of a Boolean circuit by converting all gates to an atomic gate type such as NAND or NOR.

**Generating Random Circuits.** We have established several foundational premises (as we did with programs) to now consider the existence of random circuits. We have given a concrete methodology for describing circuits as binary strings, characterized the size for such a description, characterized the size for a set of circuits with a specific gate size and basis, and characterized the entropy for functionality as circuit size increases. We have also given rules for constructing legal circuits and we now describe a (random) selection from a known population of circuits.

Considering our entropy example, we show how to concretely describe the family of 2-input / 1-output circuits with gate-size = 7 and total-size = 9. We also, by experimentation, generate a subset of this population with a specific functionality (AND) and refer to this circuit ensemble as  $C'_9$ . If we have a mechanism to select a circuit randomly from the subset  $C'_9$ , this selection constitutes an unbiased, equally likely representative from the population. That selection, by definition, is a random circuit. As with programs, however, we prefer to have a method for generating random circuits as validation basis for their existence. We provide three such algorithms, with varying efficiency.

**Algorithm 1: Random Circuit Generation by Enumeration**

*Initialization:*

1. Choose a complete basis  $\Omega$
2. Choose input size  $n$
3. Choose output size  $m$
4. Choose circuit size  $s$
5. Elaborate all possible combinations of circuits with size  $s$  or less.

*Create circuit ensemble  $C$  from this elaboration process.*

6. Assign each circuit in  $C$  a unique number,  $1 < x < |C|$

*Selection:*

7. Using a pseudorandom number generator, generate  $x$  such that  $1 < x < |C|$ . Pick circuit  $C_x$  where  $C_x \in C$ .

Then,  $C_x$  is a random circuit. Therefore, random circuits exist under Proposition 5.

Algorithm 1 exhibits super exponential run time because we must enumerate all possible circuit combinations, which involve multiple selection loops that cover all possible functions in the basis and all possible combinations of prior inputs for each gate in the circuit (Definition 14 defines a combinatorial bound). A second (more efficient) approach is to generate one circuit based on pseudorandom choices for each gate's function type and signals. This process involves direct enumeration of all  $s$  gates within the circuit in a straightforward manner.

We reflect in the third algorithm yet another approach using a set of all binary strings with length  $b$ . Given the string ensemble, we make an unbiased selection from the entire population. However, just as with programs, we must make sure that selection is a legal circuit description. Although conceptually easier with circuits (because we need not worry about termination), illegal circuits would only be encountered when any of the given circuit parameters do not fully saturate the binary space allocated for them. In other words, a basis with size 6 does not fully saturate the

3-bit representation space needed for representing gate types under that basis. A basis size of 4, however, would fully saturate its bit-representation space of 2 bits.

The other form of illegal circuit definition occurs when gate inputs derive from future inputs instead of only previous inputs. For this reason, legal (combinational) circuit constructions create directed acyclic graph representations. Such gate/input wire combinations are legal for sequential circuits, however, and in future models we may actually desire such configurations.

### Algorithm 2: Random Circuit Generation by Construction

*Initialization and Selection:*

1. Choose a complete basis  $\Omega$
2. Choose input size  $n$
3. Choose output size  $m$
4. Choose circuit size  $s$
5. Generate circuit  $C$  in the following manner using gate set  $W$ :

Set  $i := n+1$

For  $\forall w_i \in \{w_{n+1}, w_2, \dots, w_{n+s}\}$ :

- (a) Using a (pseudorandom) choice, pick gate type  $G_i$  from  $\Omega$  where there are  $1 \dots |\Omega|$  possible functions to choose from
- (b) Using a (pseudorandom) choice, pick input  $x1_i$  for  $G_i$  where there are  $1 \dots i-1$  possible previous gates to choose from
- (c) Using a (pseudorandom) choice, pick input  $x2_i$  for  $G_i$  where there are  $1 \dots i-1$  possible previous gates to choose from
- (d) Assign  $w_i = G_i(w^{x1_i}, w^{x2_i})$

$C$  is a legal circuit definition and exhibits properties of a random circuit in terms of each gate's inputs and Boolean function. Then,  $C$  is a random circuit and we affirm random circuits exist under Proposition 5.

### Algorithm 3: Random Circuit Generation by Test

*Initialization:*

1. Choose a string size ( $b$ )

*Selection:*

2. Repeat the following process until a legal circuit description is chosen
  - (a) Make a (pseudo)random string selection  $C$  from the population  $\{0, 1\}^b$
  - (b) Test to see if  $C$  is a legal circuit definition, returning yes or no
 

For specific  $n, m, s, \Omega$ : the test is efficient to compute

For unknown  $n, m, s, \Omega$ : the test is hard to compute

$C$  is a legal circuit definition (by decision). By its selection, it represents an unbiased choice from a population of all (possible circuit representation) strings for a specific class of circuits with size  $s$ , input size  $n$ , output size  $m$ , and basis  $\Omega$ . Then,  $C$  is a random circuit.

We complete this section by noting that random programs and random circuits are a foundational premise for proving white box protection attributes under the random program security model. We introduce next obfuscators that leverage this concept of randomization and polynomial time indistinguishability under our formal definition for white box obfuscation found in Definition 12.

## 5.6 Creating White Box Protection Based on Randomization

Classic security research reveals many reasons to seek strong program obfuscation theory and technology. Protecting the seam between two composed programs (black box protection under Theorem 1) is a canonical *white box* obfuscation goal central to our model. Recall that we compose a protected function ( $P$ ) with an encryption function ( $E$ ) to provide black box protection. If the adversary can identify the seam between  $P$  and  $E$  through white box analysis, the black box protection provided by  $E$  disappears. White box security encompasses both cases and gives ability to shield program intent from code analysis. Thus, white box protected obfuscations protect against analysis intended to reveal embedded data, seams between functions, the number of functions, or other such program properties. Our goal is to define a systematic, measurable defense against white box threats using the random program security model defined in Section 5.5.

The extensive history surrounding data encryption provides important insights into understanding information and its representation. We contend that programs (and circuits) are no more than a special information class with well-defined syntax and semantics. Moreover, scrambling techniques (for code) are limited because the final form must adhere to this rigid syntax and semantics. However, as we demonstrate in the previous section, program code and circuit descriptions possess information content equivalent to information content in any other type of bit stream. We present in this section a methodology for building white box obfuscators that attempts to meet both informal and formal protection specifications given under Definition 5 and Definition 12. We begin by comparing *data* and *program* encryption in Section 5.6.1 and point out the parallel lines of development we believe the (newer) field of program encryption will follow.

### 5.6.1 Comparing Data and Program Encryption

We introduce randomization to the obfuscation problem and make an appeal using traditional methods found in data encryption schemes. As we discuss in Section 5.3.1, we analyze data cipher security properties in one of two ways: 1) an information theoretic viewpoint, where data is secure regardless of computational resources; or 2) a complexity viewpoint, where data is secure based on limited resources. Data encryption strength reflects whether we can reduce possible breaks to known hard problems (e.g., factoring). Asymmetric ciphers use trapdoor one-way functions based on algebraic groups or rings. Symmetric cipher security proofs, on the other hand, do not rely on number theory. Confusion, diffusion, and composition operations form the foundation for the Data Encryption Standard (DES), AES, RC4, etc. Security proofs leverage Shannon's perfect secrecy [209], though security confidence relies on the fundamental theory of cryptography, i.e. that no *easy* attacks on symmetric schemes like DES have been found despite voluminous research efforts over the years<sup>16</sup>. Symmetric cryptosystems rely on brute force exhaustive search as their strength metric. Yet, symmetric ciphers are widely accepted as strong, despite absence of mathematical proof formulations.

There are two analogous threads in program obfuscation research. The Virtual Black Box is the de facto standard "provable security" approach, pitting the ability of a Turing machine given obfuscated code against one with only oracle access to the original function. Conversely, we use *random programs* as a baseline for measuring program intent protection through entropy. Figure 65 summarizes these notions.

Practical (heuristic) program obfuscation techniques are, in large part, observation generated. Software engineers have known for decades that certain program structures reveal more about

---

<sup>16</sup> Observation from RSA Security, <http://www.rsasecurity.com>

program intent than others. These intuitions led to obfuscation techniques such as adding ruse code, eliminating structured constructs, generating “elegant” algorithms, type casting, code reordering, code interleaving, and many others. The foundation was that if structured, concise code is easy to understand, then non-structured, elaborate code must be difficult to understand. Unfortunately, the software engineering model that seeks to understand code and the security model whose goal is to protect intent do not correspond well at their extremes. Specifically, to protecting intent against sophisticated intruders is fundamentally different from revealing intent to maintenance programmers. Thus, program obfuscation techniques focus on confusing code, with little theory or evidence that independent mechanisms are complementary, or even that they are not counter-productive.

<b>Asymmetric Data Encryption</b>	<b>Symmetric Data Encryption</b>
Based on mathematical algebraic primitives	Based on repetitive permutation/substitution
Provably secure relative to mathematical theory	Time-tested, secure based on limited resources
Key-based, systematic, recoverable	
Seeks to create ciphered data with discernible randomness properties	

<b>Program Obfuscation</b>	<b>Program Encryption</b>
Spurious, heuristic, limited	Based on repetitive, heuristic use of permutation/substitution primitives with composition
Not provably secure in the general case (VBB); secure in limited contexts	Time-tested / complexity (secure based on limited resources)
Mechanism-specific, non-generalized	Key-based, systematic, recoverable
Seeks to protect programs against specific attacks using specific techniques	Seeks to create ciphered programs with discernible properties of randomness

**Figure 65: Comparing Data Ciphers with Program Obfuscation / Encryption**

We contend that we can measure confusion by comparing our systematically obfuscated code (hereafter referred to as “encrypted code”) or circuits against random code or circuits. We adhere to Kerckhoffs’ security principle [210] and leverage substitution and permutation engines similar to symmetric key encryption techniques. We thus consider our methodology a form of program “encryption” rather than program “obfuscation”.

We specify obfuscators that generate key-based, white box secure software modules that remain executable. We reiterate the difference in our approach from using a *data cipher* to *encrypt code*, making the code a random data stream unintelligible to the code’s (originally) intended interpretive environment. In Aucsmith’s approach [164], he utilizes a key to generate pseudorandom blocks of encrypted code that are decrypted just prior to execution. Our approach is similar to homomorphic forms of encryption, but instead of mathematical group operations, we utilize executably semantic preserving primitives found in traditional symmetric data ciphers. We refer to these primitives as confusion and diffusion. We define and show the utility of random programs in Section 5.5 for measuring the (relative) randomization that any given obfuscator produces.

Program (or circuit) encryption mechanisms are key-based functions, with corresponding recovery mechanisms. These algorithms produce programs with well-understood randomness properties. Program protection algorithms that utilize confusion, diffusion, and composition strategies (like DES) are not necessarily *weaker* than mathematically based functional-transformations such as homomorphic encryption schemes or RSA. To illustrate, consider permutation and substitution data ciphers.

Data **permutation**, or **transposition**, shuffles the order of data, where the key dictates the shuffle order. When used alone as a data cipher mechanism, permutation **diffuses** data across ciphertext (as Figure 66 illustrates), but is not cryptographically strong alone. When applied in isolation (by itself), we may rightfully consider data permutation a data *obfuscation* method. Data **substitution**, or **replacement**, when used alone as a cipher technique, **confuses** bits within a

ciphertext but is not individually cryptographically strong either. We can rightly consider it a form of data *obfuscation* by itself.

However, cryptographers strategically compose permutation and substitution in round-based production block ciphers. In doing so, they can create strong encryption, evidenced in well-known symmetric ciphers like DES. Even though DES strength is difficult to mathematically express in other than brute force terms, most cryptographers recognize it as a strong cipher that has no known attacks *significantly* more efficient than brute force key discovery.

**Permutation**

H	E	L	L	O	A	N	D	G	O	O	D	B	Y	E	W	O	R	L	D	!	Plaintext
H	L	N	D	B	W	L	E	O	D	O	Y	O	D	L	A	G	D	E	R	!	Ciphertext

**Substitution**

H	E	L	L	O	A	N	D	G	O	O	D	B	Y	E	W	O	R	L	D	!	Plaintext
G	\$	Z	Z	B	^	E	R	<	B	B	R	B	P	\$	N	B	Q	Z	R	N	Ciphertext

**Figure 66: Example of Data Permutation and Substitution**

We leverage the *program encryption* analogy that uses confusion and diffusion that are not strong themselves, but when composed in systematic, round-based algorithms produce executably encrypted code. Analyzing program encryption security under our randomization model breaks any relationship with VBB security; instead, we appeal to the random program security model (Section 5.5). We next illustrate circuit indistinguishability as a program encryption security metric.

**5.6.2 Integrating Black and White Box Protection**

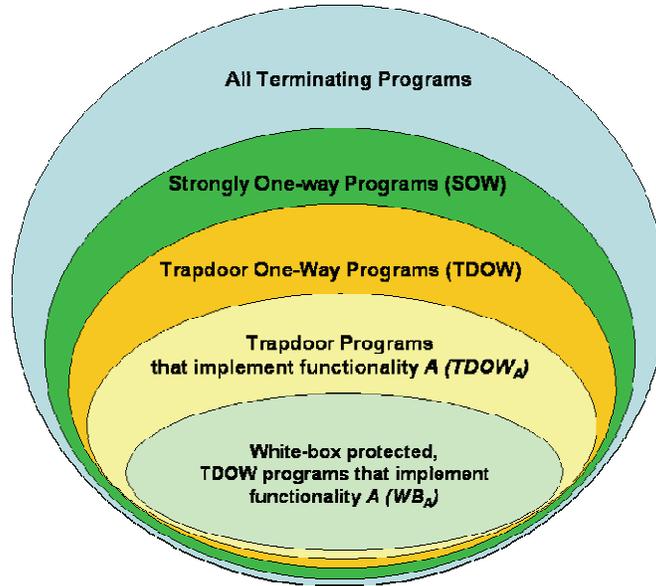
Under Definition 4, the goal of black box intent discovery by an adversary is to establish the I/O relationship that exists for an obfuscated program  $p'$ . If the adversary cannot find the functionality class  $A$  given runtime analysis of the obfuscated version  $p'$ , black box protection is achieved. By definition, the family of all programs that implement one-way functions consists of programs whose input/output behavior is hard to learn. The security game played with an adversary involves not knowing or being able to determine a program's I/O class or functional category.

In Definition 12, we express how to measure whether an adversary has an advantage when given the obfuscated program (code) or circuit over oracle-only access to the original program. We analyze whether the adversary distinguishes the obfuscated program from a randomly selected program of the same size. This includes an adversary who not only performs black box analysis but also performs static or dynamic analysis of the code itself, specifically to determine *program intent*. To reiterate, we do not attempt to prove *general* security against all-powerful adversaries—rather we seek a more narrowly defined goal of intent protection and a framework to evaluate security of practical obfuscation techniques.

To afford full intention protection (under Definition 6), we must protect against both black box and white box analysis. Figure 67 depicts the program families of interest that afford intent protection, beginning with the foundational program class of strongly one-way functions. Trapdoor functions, from Definition 8, are one-way functions where the inverse is easy to compute given the key, but hard otherwise. We assume that black box protection uses cryptographically strong, one-way trapdoor functions under Definition 9. We let  $E$  represent a trapdoor one-way encryption function that takes a plaintext message  $M$  and key  $K$  and returns a recoverable ciphertext  $C = E(M,K)$ . We assume an inverse function  $D$  related to  $E$  provides recovery of the plaintext given a ciphertext and symmetric key  $K$ :  $M = D(C,K)$ .

Our black box protection mechanism forms a special subclass of trapdoor one-way programs that has a special input/output relationship defined by the functionality class  $A$  and any program  $P \in A$ . Specifically, when we concatenate a program  $P$  that has a specific functionality  $A$  ( $P \in A$ )

with an encryption algorithm  $E$  from the trapdoor one-way function family  $\mathcal{E}$ , we have programs consistent with those described in Theorem 1. We show this family of programs in Figure 67 as the subset  $\text{TDOw}_A$ . Given the transformation process  $t$  of Theorem 1 that creates a specific subclass of programs in  $\text{TDOw}$ , a recovery algorithm  $r$  recovers the intended output  $y$  of any program  $P$ , given the output of  $y' = P'(x)$ , where  $P' \in \text{TDOw}_A$  in Figure 67. The set of programs  $\text{TDOw}_A$  is black box intent protected under Definition 4 with respect to the functionality class  $A$ .



**Figure 67: White Box Protected Programs**

From a compositional approach, any black box obfuscator  $O$  (under Definition 4) that implements Theorem 1 is a compiler that produce  $P' = O(P)$  from an original program  $P \in A$  and a strong, trapdoor one-way program  $E \in \mathcal{E}$  such that  $P'(x) = E(P(x), K)$ . Here  $P' \in \text{TDOw}_A$  and indicates that the set  $\text{TDOw}_A$  contains all programs whose input/output relationship accommodates the domain of  $A$  and produces the range of  $E$  such that  $P': \{0, 1\}^{|\text{in}P|} \rightarrow \{0, 1\}^{|\text{out}P|}$ . A black box obfuscator that meets Theorem 1 thus produces obfuscated programs whose input/output characteristics are consistent with  $E$  and are thus one-way functions. We clarify that the selection of the particular class of functions  $E$  is a key-based decision part of an overall obfuscation process. Thus,  $E$  is randomized along with other parameters and the functionality class  $A$  may itself include strongly one-way programs, trapdoor one-way programs, or data encryption algorithms.

### 5.6.3 Intent Protection with White Box Randomizing Transformations

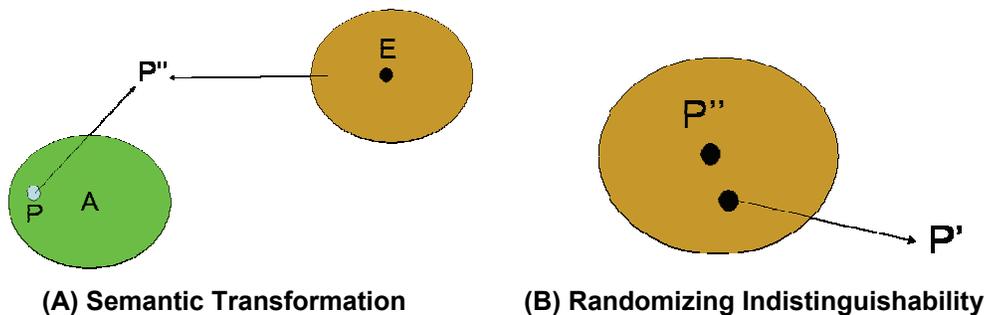
At this point we refer specifically to Boolean circuits (using  $\text{TDOw}_A$  to refer to a set of circuits) and of obfuscators that algorithmically manipulate circuits. As we elaborate in Section 5.5.4, circuits provide a better meeting point between theoretic limits and practical implementation and eliminate the need to worry about program termination. Considering both forms of intent protection (from Definition 4 and Definition 12), we now define obfuscators that perform systematic circuit *transformations* based on indistinguishability from a random circuit. Such white box obfuscators assume any candidate circuit  $P' \in \text{TDOw}_A$  as a starting point. Since  $\text{TDOw}_A$  is infinitely large, we bound the possibilities by specifying only circuits with a maximum size  $N$  or less. For example, if  $\mathcal{E}$  were the  $N$ -bounded family of Boolean circuits that implement the DES algorithm, all elements in  $\mathcal{E}$  are circuits of size  $N$  or less that produce the mapping  $E_{\text{DES56}}: \{0, 1\}^{64} \times \{0, 1\}^{56} \rightarrow \{0, 1\}^{64}$  based on 64-bit message size and a 56-bit key  $K$ . If we choose a specific 56-bit key  $K$ , then we have an embedded-key DES function defined as  $E_{\text{DES56}, K}: \{0, 1\}^{64} \rightarrow \{0, 1\}^{64}$ .

The specified maximum circuit size  $N$  represents the desired obfuscated circuit efficiency; we consider obfuscators that randomize a circuit in a way that produces exponential circuit blow up unless bounded otherwise. The lower bound size of circuits in  $TDOW_A$  is based on the size of the most efficiently reduced circuits that implement  $P'(x) = E(P(x), K)$ . A maximum circuit size  $N$  bounds the number of circuits that implement  $E$ . Likewise,  $N$  provides a bound on the number of circuits in the set of all trapdoor one-way functions.

We base white box protection on an indistinguishability argument. As Definition 12 states, we achieve white box intent protection if a circuit obfuscator (encryptor) produces an obfuscation of  $P$  that is indistinguishable from a random circuit  $P_R$ . We use the random program (circuit) model as the security basis and ask whether obfuscators exist that achieve this intent protection form. By Corollary 1, which affirms that pseudorandom program generators exist, we conjecture that pseudorandom program generators can exist also that reliably transform one program (circuit) form into a semantically equivalent / executably encrypted program (circuit) form.

We again leverage the well-understood notion of traditional data ciphers to illuminate our paradigm. Strong data encryption produces ciphertext that is indistinguishable from a string chosen randomly from the set of all strings of the same size. Cryptographically strong data ciphers that use permutation, substitution combinations accomplish this successfully. Our desire is to design or find obfuscators that utilize circuit permutation and substitution to produce randomized circuits; these randomized circuits are indistinguishable with respect to  $P$  from any other circuit of comparable size chosen randomly. If random circuit selection provides white box protection, as we contend, then our effort is reduced to finding mechanisms that produce suitably randomized "cipher code" (to coin a phrase).

An obfuscator  $O$  that provides full intent protection for a program (circuit)  $P$  (under Definition 6), such that  $P' = O(P)$ , can thus be seen as a two-step compiler.  $O$  first provides black box obfuscation by a semantic transformation on  $P$  to  $P''$  under Theorem 1 (depicted in Figure 68-A).  $O$  then provides static white box obfuscation by randomization of  $P''$  to  $P'$  (depicted in Figure 68-B).



**Figure 68: Full Intent-Protected Program  $P'$**

We have already demonstrated in Section 5.4 that we can produce obfuscators that satisfy black box protection. In order to create a full intent-protection obfuscator, we must ensure that any candidate  $O$  selects programs  $P'$  from the set  $TDOW_A$  in a uniform, random, key-based, and repeatable fashion. We conjecture that if we randomly select a circuit from  $TDOW_A$ , this selection is indistinguishable from a random selection from the parent set  $E$  (recall that by virtue of construction,  $P'(x) = E(P(x), K)$ , where  $E \in E$ ). We further investigate whether the selection of  $P'$  can be made indistinguishable from a random circuit selection taken from the parent sets of  $E$ , which include  $TDOW$  and from  $SOW$  (seen in Figure 67).

We have two goals based on these foundations. First, if an obfuscator randomly selects a bounded size circuit from  $TDOW_A$ , this selection is indistinguishable from a bounded size circuit randomly selected from  $E$ . Secondly, we investigate whether efficient obfuscators exist that randomly selects a circuit from  $TDOW_A$ . The secondary goal has to do with practical

implementation of the first and we discuss our initial results toward that aim next. In a sense, the second step corresponds with classic efforts to confuse code. While other approaches lack structure, in our approach, there is a well-understood goal (randomization) and a metric (non-linearity).

#### 5.6.4 Distinguishing Random Selections of $TDO\bar{W}_A$ from $P_R$

A circuit has input/output mappings that reflect its functional behavior. We summarize such mappings by either truth table or the characteristic Boolean function of the circuit in some reduced, canonical form. By definition, circuits in the set  $TDO\bar{W}_A$  are one-way functions and are therefore not analyzable by their input/output mappings—they are indeed hard to learn based on their membership in the set of all one-way functions. Given a circuit  $P'' \in TDO\bar{W}_A$ ,  $P'': \{0,1\}^{64} \rightarrow \{0,1\}^{64}$ , with an appreciably large input size (64 bits) and appreciably large output size (64 bits), the truth table for such a circuit  $P''$  has  $2^{64}$  rows. Without being able to analyze the input/output pairs of circuits in  $P''$ , no link to an original  $P$  is possible on the basis of input/output analysis alone, given that  $P'' = E(P(x),K)$ . An adversary must then analyze circuits that come from the family  $TDO\bar{W}_A$  using combined static and dynamic techniques.

There are uncountably many circuits in the unbounded sets  $E$  and  $TDO\bar{W}_A$ . Given only circuits of size  $N$ ,  $E$  and  $TDO\bar{W}_A$  are finite and allow possible uniform selection. Given a basis  $\Omega$ , there is a large but countable set of circuits of size  $N$  or less that implement encryption functionality  $E$  and that ultimately compose  $TDO\bar{W}_A$ . The set  $TDO\bar{W}_A$  has large, but finite, cardinality and  $E$  is by implication much larger. We stipulate that at least one element of the set  $TDO\bar{W}_A$  is selected as the first step of a full intent-protection obfuscator: the circuit created by black box protection using the Theorem 1 (semantic transformation of  $P$  to  $P''$  seen in Figure 68-A). However, by applying both sub-circuit confusion and diffusion to  $P''$  in a round-based, repetitive manner, we select an equivalent, random circuit from  $TDO\bar{W}_A$  which we refer to as  $P'$  (seen in Figure 68-B). Note also that  $P''$  and  $P'$  are semantically equivalent to each other (they come from the same set  $TDO\bar{W}_A$ ) whereas  $P''$  and  $P'$  are neither semantically equivalent to the circuit we are interested in protecting ( $P$ ):  $P'' = P' = E(P(x),K)$ .

Given a mechanism (obfuscator) that randomly selects a circuit from the set  $TDO\bar{W}_A$ , we assert that such a circuit is indistinguishable from a randomly chosen circuit from the set  $E$ . The group of all permutations of  $\{0,1\}^{64}$  is considered large enough to satisfy a brute-force discovery of the key (having  $2^{64}!$  elements), even though some attacks on DES slightly reduce the number of plaintext/ciphertext pairs required to be successful. We draw a parallel and say that the number of representations for circuits that implement  $P'' = E(P(x),K)$  with characteristically large input/output  $\{0,1\}^{64} \rightarrow \{0,1\}^{64}$  form a pool for random selection. Recall that selection from  $TDO\bar{W}_A$  does not preserve the original functionality of  $P$ , but preserves black box protected functionality. In Section 5.5.4, we make an argument for exponential entropy increase in circuits that have increasing size and increasingly complex functionality. We show by our empirical experimentation that for simple functionality (a single AND), the number of circuits implementing that functionality is (exponentially) large. As the complexity of a circuit's function increases (i.e., implementing a data encryption cipher versus simple AND), we also would expect a corresponding increase in the (exponentially) large number of circuit combinations that can implement that functionality. By observation,  $E$  and  $TDO\bar{W}$  are much larger than  $TDO\bar{W}_A$ . Our premise is that an adversary, when given a random circuit ( $P_R$ ) that implements the one-way function class of  $E$ , cannot distinguish that circuit from one that comes from the subset  $TDO\bar{W}_A$ . If we can create an obfuscator with such properties, we accomplish white box protection.

#### 5.6.5 Obfuscators that Randomly Select Circuits from $TDO\bar{W}_A$

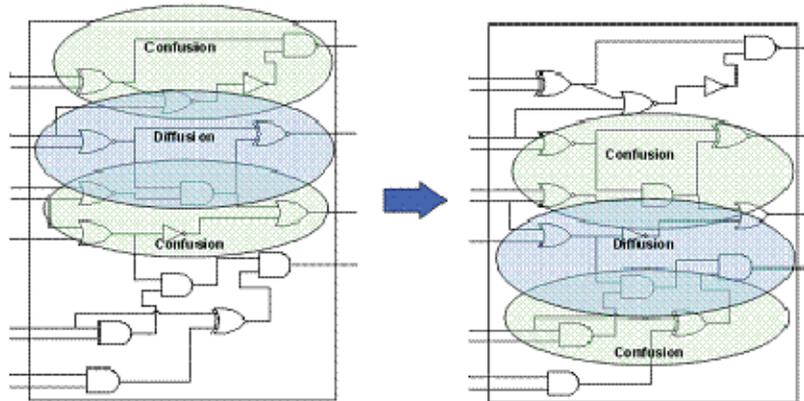
Given a binary string representing a circuit, we define a process that selects legal sub-circuit substitutions-permutations that preserve circuit functionality. The resultant binary representation reflects these transformations and mimics data plaintext replacement with an equivalent ciphertext substring. In symmetric, Feistel-based [211] data ciphers, security strength comes from the ability to perform key-based operations that are random and uniform across the plaintext. Ciphers normally accomplish this one plaintext block at a time and return recoverable ciphertext blocks. In

confusion-diffusion approaches, ciphers transform each block by a series of key-based operations that include some type of non-linear substitution on small portions of the string (4 bits for example) and then permutation across the entire string.

We leverage non-linear substitution for sub-circuit replacement within a parent circuit. Even though circuits in  $TDOw_A$  are large, we build the obfuscator to work with fixed (small) size sub-circuits and create sets (substitution boxes) of circuits that preserve functionality (i.e., produce the same truth table or signature). The intuition is that given a bounded size circuit, if sub-circuits are randomly chosen and replaced repetitively (up to some number of rounds), the resultant circuit has properties consistent with a randomly selected circuit from the pool of circuits  $TDOw_A$ .

As the basis for non-linear security properties in DES [212, 213, 214, 215, 216, 217, 218], S-boxes transform bit strings from larger to smaller sizes. In the case of circuits, we replace a circuit of some (small) size with an equivalent circuit of closely smaller, equal, or greater size that has equal number of inputs and outputs. We assume initially that circuit substitution boxes produce equivalent sized circuits. Cryptographic algorithms based on the strength of non-linear substitution also rely on a given number of confusion/diffusion rounds. We define a circuit substitution operation as a non-linear equivalent replacement of a sub-circuit and a circuit diffusion operation as a substitution that comes because of two different replacement operations.

Figure 69 shows a notional circuit transformation where two other sub-circuit replacements diffuse the original functionality. Beginning with the circuit  $P'' = E(P(x), K)$ , we apply round-based sub-circuit selection-replacement so that all gates are considered for replacement at least once. Each selection-replacement round within  $P''$  is key-based. Unlike block-ciphers, not all sub-circuit definition blocks are contiguous. This dictates multiple selection/replacement rounds using various (small) input size and output size sub-circuits. A one-time, up-front cost is required to create equivalence classes for circuits—much like the requirement to design S-boxes part of symmetric data ciphers.

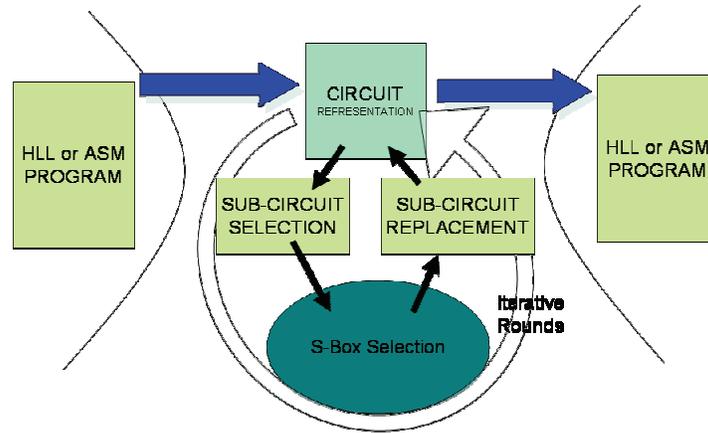


**Figure 69: Circuit Substitution and Permutation**

An obfuscator that takes a circuit  $P''$  and produces an equivalent circuit  $P'$  based on this process produces a string representation of  $P''$  with properties consistent to a random circuit. Such an obfuscator fulfills the requirements we lay out for full intent-protection (black box and white box) in Definition 4 and Definition 12. In particular, the binary string representations of  $P''$  compared to  $P'$  would map closely to the plaintext/ciphertext pair produced by a symmetric data cipher like DES. If the obfuscator functions in this manner, the resultant circuit is indeed indistinguishable from a random circuit. We envision incorporating such a white box protection approach into a higher-level algorithm that provides fully general program intent protection, illustrated in Figure 70. We discuss our implementation activities further in Section 5.8 and Appendix D.

The creation process for a randomizing white box obfuscator resembles the DES creation process in many ways: we must be able to analyze an implementation in order to measure its true resilience. On a theoretic level, we have provided arguments that fully intent-protected

obfuscators based on semantic transformation and circuit randomization would defeat combined black box and white box analysis attacks. Given that we pre-construct circuit substation boxes (an efficient operation for small circuit sizes), the algorithm for subcircuit selection and replacement on  $P''$  to create  $P'$  would have similar DES efficiencies for general circuits of reasonable size as well.



**Figure 70: Circuit Encryption in Context to HLL Code Protection**

By Kerckhoff's principle (*i.e.*, the adversary has knowledge of the encryption process), we prefer a tighter proof that an adversary cannot perform certain (specific) actions. Particularly, when an adversary is given a circuit that by its creation represents a random selection from a large population (making it indistinguishable from a random circuit), can we provably assert that this random circuit completely prevents the adversary from finding the “seam” that separates circuit (program)  $P$  from circuit (program)  $E$ ? Though fully implementing our randomizing circuit obfuscator requires additional future work (and thus future analysis in this regard), we are able to provide an alternative theoretical view of white box protection that *does* prove perfect semantic security (even when an embedded-key encryption algorithm is used). For bounded input-size programs and circuits, this methodology not only exhibits provably perfect semantic white box security, but also affords practical, real-world implementation (which we accomplish under our current work). We discuss this approach next.

## 5.7 Creating Perfect White Box Protection

In this section, we show how to produce a semantically secure obfuscation for  $\{P_n\}_{n \in \mathbb{N}}$ , which is the class of programs with input size  $n$ . Unlike other (obfuscation) results, the only definition we give for  $P_n$  is a polynomially related bound  $b$  on the input size such that  $n, b \in \mathbb{N}$  and  $2^n \leq n^b$ . Given such a bound, we show how to produce obfuscated circuits that are efficient, semantically equivalent, and virtual black box protected to the original program. The algorithmic complexity of the obfuscation is exponential, but, when bounded polynomially, is practical for a relevant class of programs (which we motivate in Section 5.2). In our formulation, we actually appeal to the VBB notion that (any) source code version should not leak more information than a simulator with oracle access to the source code.

To bridge the gap between theory and practice, we address the “best-case” of what can be achieved. Turing machines are not physically constructible, even though they represent the theoretical underpinning of computer science; any best-case implementation of a Turing machine would require a (bounded) limit on infinitely defined tapes. What does the best case, practical virtual black box circuit look like from a security perspective? We answer that question by considering obfuscators that are based only on oracle-access to a function  $P$ , and not the original function  $P$  itself. By definition, an obfuscated circuit  $P'$  should not leak any more information about  $P$  than the oracle of  $P$  reveals. This is our baseline for perfect white box protection as we

state in Definition 16 the notion of a bounded input-size program obfuscator. In Definition 15, we give a reminder definition for the negligible function.

**Definition 15. (Negligible Function)** Function  $\alpha: \mathbb{N} \approx \mathbb{R}^+$  is negligible if, for any positive polynomial  $p$ , there exists  $N \in \mathbb{N}$  s.t  $\alpha(n) < p(n)^{-1}$  for any  $n > N$

**Definition 16. (Bounded Input-size Program Obfuscator)** An algorithm  $O$  is an obfuscator for the class of  $b$ -bounded input size programs  $\{P_n\}_{n,b \in \mathbb{N}, 2^n \leq n^b}$ , where  $P \in \mathbb{P}_n$  if:

1. **Semantic Equivalence:**  $\forall x, P(x) = P'(x)$ , where  $P' = O(P)$
2. **Efficiency:** There is a polynomial  $l(\cdot)$  such that for every  $n, b \in \mathbb{N}$  where  $2^n \leq n^b$ , and for every  $P$  in  $\mathbb{P}$ ,  $|O(P)| \leq l(|P|)$
3. **Perfectly Secure Obfuscation:** For any PPT  $A$ , there is a PPT simulator  $S$  and a negligible function  $\alpha$  such that for every  $n, b \in \mathbb{N}$  where  $2^n \leq n^b$ , and for every  $P \in \mathbb{P}_n$

In their argument formulation, Barak *et al.* acknowledge a valid obfuscation exists for circuits in the following manner:

*“Note that if we had not restricted the size of the obfuscated circuit  $O(C)$ , then the (exponential size) list of all the values of the circuit would be a valid obfuscation (provided we allow  $S$  running time  $\text{poly}(|O(C)|)$  rather than  $\text{poly}(|C|)$ ).” [172]*

We explore this statement and define explicitly the constructions related to this possibility. The VBB impossibility proofs in general deal with (contrived) functions where the input size is too large for practical truth table enumeration—therefore a simulator with oracle access to an original program  $P$  (defined as  $S^P$ ) can do no better than guessing based on oracle-queries. We consider instead the family of functions whose input size is small and therefore whose input/output behavior is not prohibitive for a simulator to enumerate.

Barak *et al.* also state that the foundation of (all) of their proofs derive from the “fundamental difference between getting black box access to a function and getting a program that computes it, no matter how obfuscated” [172]. They go on to state that this difference disappears if the function is learnable completely from oracle (black box) queries. Our interest in bounded input-size programs/circuits is that we (or a simulator) can obtain their truth tables efficiently when they have a sufficiently limited input size.

### 5.7.1 Existence of 2-TM Obfuscators for Bounded Input-Size Programs

Since we have already introduced the Barak impossibility proofs earlier in Section 5.3.3, we proceed immediately to our interest regarding them: defining a version of the VBB property related to a bounded input-size parameter. Regarding Equation 4 (p. 72), when we bound the input size  $k$  polynomially, the probability that we can compute the predicates in Equation 4 is much different—and this corresponds exactly with what Barak *et al.* state. Under a bounded  $k$  assumption, we can distinguish a  $\text{poly}(k)$ -time algorithm  $S$  that has oracle access to  $C_{\alpha,\beta}$  and  $D_{\alpha,\beta}$  from another algorithm  $S$  that has oracle access to  $Z_k$  and  $D_{\alpha,\beta}$ . This is because both simulators ( $S^{C_{\alpha,\beta}, D_{\alpha,\beta}}$  and  $S^{Z_k, D_{\alpha,\beta}}$ ) can enumerate the truth table for  $C_{\alpha,\beta}$  or  $Z_k$ , create a circuit from that truth table, and get a decision from  $D_{\alpha,\beta}$  accordingly (we describe how obfuscators do exactly this in the theorems that follow). Thus:

$$\text{Equation 7. } |\Pr[S^{C_{\alpha,\beta}, D_{\alpha,\beta}}(1^k) = 1] - \Pr[S^{Z_k, D_{\alpha,\beta}}(1^k) = 1]| = 1, \text{ for bounded } k$$

Equation 7 shows precisely, given bounded input size  $k$ , that the difference between oracle black box access and source code access does indeed vanish. We leverage this fact and introduce constructions next that meet the VBB security definition for a useful class of programs that we can obfuscate: those with small input size. This also does not contradict the VBB results in [172] at all because functions with enumerable input/output (exactly learnable via oracle queries) are candidates *for* meeting the VBB property.

### 5.7.2 Provably Secure Obfuscators for Bounded Input-Size Programs

In the information theoretic sense, we define perfectly secure obfuscation by information gained by a PPT simulator  $S^P$  that has oracle-only access to some original program  $P$ . If a PPT algorithm uses only the information gained from an oracle of  $P$  to construct a semantically equivalent circuit  $P'$  for  $P$ , then it is impossible for any circuit  $P'$  created in a such a manner to leak more information than what the oracle for  $P$  could give. In particular, an oracle for  $P$  simulates an algorithm that utilizes the truth table of  $P$ . The existence of such an oracle simulator for  $P$  assumes that the possible input range of  $P$  and its corresponding output can be fully enumerated, stored, and accessed.

We pause to clarify and amplify an oracle’s capability. Classically, an oracle answers questions with no notion, reference, or intuition on our part as to how it knows the answer; we universally accept that the oracle’s answers are correct. We utilize truth tables in our arguments because truth tables capture the oracle’s capability for answering function queries, since each answer, essentially, fills in a space in the function’s truth table.

Some functions are easily learnable (as Barak *et al.* point out) in that we can learn them from partial truth tables. Our results address functions whose truth tables we can completely *construct* in polynomially bounded time from oracle access, and point out that, even for functions whose complexity grows exponentially, truth table construction complexity simulates polynomial growth for small input sizes. This function class provides the opportunity to observe provably VBB protected circuit implementations. These circuit implementations possess [VBB] perfect security because, given the circuit  $C_{\alpha,\beta}$ ’s truth table (using an example from the impossibility proofs), we can canonically construct a circuit exclusively from that truth table. Since the truth table is generated by oracle access and the obfuscated  $C_{\alpha,\beta}$ ’ is generated canonically from that truth table also, the circuit can reveal nothing more about the original circuit  $C$  than does the oracle. This concept is illustrated and leveraged in Theorem 3.

A natural question to ask is: “How does protecting a circuit whose truth table can be computed provide security?” As we mention in our review of obfuscation security models, a well-demonstrated value exists for obfuscation models that operate on semantically non-equivalent versions of programs and circuits. In our ideal black box construction (under Theorem 1), we protect the truth table for the obfuscated circuit via black box semantic transformation, and thus do not reveal anything about the original circuit’s I/O. Moreover, the canonical circuit construction described in Theorem 3, when used as an obfuscation technique, reveals nothing about the original circuit *structure*, thus providing perfect white box protection.

**Theorem 3.** Perfectly secure white-box obfuscators exist for  $b$ -bounded input-size programs (under Definition 16).

**Proof:** Our proof is by construction. We give a three-step obfuscator  $O(P)$  that takes any executable program  $P$ , generates the truth table from oracle access to  $P$ , and applies a Boolean canonical reduction on the truth table to produce a circuit that is semantically equivalent to  $P$ . Assume  $n$  is the input size of  $P$  and let  $2^n \leq n^b$ , for some user specified  $b$ .

Then:  $O$  is a  $b$ -bounded input-size program obfuscator for the class of programs  $\{P_n\}_{n,b \in \mathbb{N}}$ ,  $2^n \leq n^b$ , for any  $P \in P_n$ , under the following construction:

**Step 1.** Using  $P$ , acquire or create  $S^P$  as an efficient oracle emulation of  $P$ .

**Step 2.** Generate the truth table for  $P$ ,  $T(P)$ , by running  $S^P$  on all  $2^n$  inputs of  $P$ . Given  $P: \{0,1\}^n \rightarrow \{0,1\}^m$ ,  $T(P)$  is the  $m \cdot 2^n$  size matrix of input/output pairs obtained in the following manner:  $\forall x, [x,y] = [x, S^P(x)]$ , where  $S^P$  is a PPT simulator with oracle access to  $P$ .

**Step 3.** Create circuit  $P'$  by applying the algorithm for canonical complete-sum of products [219, 220] to  $T(P)$ .  $P' = \sum_{i=1, \dots, n} \pi_i$  is in disjunctive normal form (DNF) where each product  $\pi_i$  is a conjunct of literals and each literal is either an input variable  $x_j$  or its negation  $x'_j$  ( $1 \leq j \leq n$ ). Minimize  $P'$  via minimal-sum of products algorithm such as Blake's reduction based on Shannon's recursive expansion.

1.  $P'$  is perfectly secure with respect to  $P$ . Since  $P' = O(P)$ ,  $T(P)$  is *fully derivable* given  $P$  assuming some polynomially bound  $b$  on input size  $n$ . Given bounded size, the following relationship holds between any PPT simulator  $S^P$  and obfuscator  $O$ . Both can derive  $T(P)$  and thus a canonical circuit for  $P$  in polynomially bounded time.

$$\left| \Pr[A(O(P)) = 1] - \Pr[S^P(1^n) = 1] \right| \leq \alpha(n), \text{ for bounded } n$$

Because the adversary may query the obfuscated program in polynomially bounded time and derived the full truth table,  $T(P)$ , then  $\Pr[A(O(P)) = 1] = 1$ . Because the simulator, given black box access to  $P$ , may use polynomially bounded time black box queries and derive the full truth table,  $\Pr[S^P(1^n) = 1] = 1$ . Thus, the two can distinguish properties of  $P$  with equally likely probability and thus with negligible difference.

2. For  $\forall x, P(x) = P'(x)$ . By construction,  $P'$  precisely implements  $T(P)$ .
3. There is a polynomial  $l(\cdot)$  such that for every  $n, b \in \mathbb{N}$  where  $2^n \leq n^b$ , and for every  $P$  in  $\mathcal{P}$ ,  $|O(P)| \leq l(|P|)$ . In the worst case, a *complete* sum-of-products expansion is composed of  $m$  outputs consisting of up to  $2^n$  minterms composed of up to  $n-1$  products (AND) and up to  $2^n-1$  summations (OR). The maximum size,  $m2^n(n-1)(2^n-1)$ , is  $O(2^{2n})$  while the minimal possible size is  $\Omega(m)$ —representing where each output is constant 0. By bounding the input size of program  $P$  with  $b$ , the size for the complete sum of products expansion circuit becomes  $O(n^b)$ . We would not (in practice), use the complete sum of products expansion because much more efficient representations are possible. From the security aspect alone, however, any *more-efficient* derivation of the complete sum of products circuit retains the perfectly secure obfuscation (hiding) property.
4. The minimal SOP expression of  $P'$  is polynomially equivalent in input-size to the original  $P$  related to some polynomial bound  $b$ , because  $n = |x_P|$  and  $|P'| \leq n^b$ .

We point out that obfuscators constructed under Theorem 3 produce perfectly white-box protected circuits (in the information theoretic sense) from bounded input-size programs, but assume nothing about the hardness or difficulty of learning the original program  $P$ . If the input/output of  $P$  (and thus any semantically equivalent version of  $P$  such as  $P'$ ), reveals the intent or function of  $P$ , then no degree of white-box hiding can prevent the adversary from learning the function of  $P$  from the input/output relationships of  $P'$  (we state this precisely in Lemma 2). The truth-table derived construction of Theorem 3 perfectly hides only the algorithmic construction of  $P$ —and nothing more.

### 5.7.3 Perfect Obfuscation in a Private Key Setting

In the VBB constructions,  $P$  is assumed to be a function whose input/output behavior is hard to learn to begin with. However, constructions under Theorem 3 point out two useful practical realizations when used in context to hard-to-learn, one-way, pseudorandom functions: truth-table-based circuit derivations provide a method to hide embedded encryption keys programmatically and perfectly secure obfuscated private-key encryption schemes are possible where the (unpadded) input size (of the plaintext) is bounded.

Several block-cipher-based, private-key encryption schemes exist with pseudorandom properties. The hardness of key recovery and the one-way properties of ciphers such as DES are well established and pseudorandom properties of the DES family is discussed by Bellare *et al.* in [221] and Goldreich in [188]. Our interest in the DES family of functions, including variants such as 3-DES, is the comparatively small block size of the plaintext (64 bits). Though the virtual key size of 3-DES is larger than 56 bits, we focus on DES nonetheless with its standard 56-bit key space.

**Definition 17. (Private-key Block Encryption Program Obfuscator)** The tuple of PPT algorithms  $(KG, E, D, O)$  enforces perfectly secure obfuscation in the private-key setting with security parameter  $k$  and block-size  $m$  for the class of programs  $\{E_{k,m}\}$  where  $E \in E_{k,m}$  if:

1. **Private Key Encryption:**  $(KG, E, D)$  defines a pseudorandom private-key block encryption scheme with block-size  $m$  and security parameter  $k$ :
  - $KG$ : a probabilistic algorithm which picks  $K$  (on input  $1^k$ , produces key  $K$ ); assume  $KG$  never produces “weak” keys
  - $E$ :  $\{0, 1\}^k \times \{0, 1\}^m \rightarrow \{0, 1\}^m$ , on input  $K \in \{0, 1\}^k$  and plaintext message  $M \in \{0, 1\}^m$ , produces ciphertext  $C \in \{0, 1\}^m$
  - $D$ :  $\{0, 1\}^k \times \{0, 1\}^m \rightarrow \{0, 1\}^m$ , for all  $K \xleftarrow{R} KG(1^k)$  and  $M \in \{0, 1\}^m$ ,  $D_K(E_K(M)) = M$
2. **Semantic Equivalence:** Given  $K \xleftarrow{R} KG(1^k)$  and program  $E \in E_{k,m}$ ,  $\forall x$ ,  $E(K, x) = E'(x)$ , where  $E' = O(K, E) = E_K(\cdot)$
3. **Efficiency:** There is a polynomial  $l(\cdot)$  for every  $E$  in  $E_{k,m}$   $|O(K, E)| \leq l(|E|)$
4. **Perfectly Secure Obfuscation:** For any PPT  $A$ , there is a PPT simulator  $S$  and a negligible function  $\alpha$  such that for every for every  $E \in E_{k,m}$  and for every  $K \xleftarrow{R} KG(1^k)$

$$\left| \Pr[E, A(O(K, E)) = 1] - \Pr[E, S^{E_K}(1^m) = 1] \right| \leq \alpha(n)$$

We assume any distinguisher does not have access to the private key  $K$  but has knowledge of the encryption program  $E$ .

In Definition 17, we specify the requirements for an obfuscator of block-based private-key encryption schemes (such as DES), that provides a semantically secure hiding of an encryption key. In essence, the obfuscator  $O(K, E)$ , under this definition, takes a private-key  $K$  and block encryption algorithm  $E(K, \cdot)$  and returns  $E_K(\cdot)$  such that no key-recovery attack can reveal the key  $K$  based on analysis of the source code/gate structure of  $E_K$ . Theorem 4 now gives the formulation for obfuscating a key-embedded block cipher under the construction of Theorem 3.

**Theorem 4.** Perfectly secure obfuscators exist for  $b$ -bounded input-size private-key block encryption programs.

**Proof:** Our proof is by construction. We give a three step obfuscator  $O$  that takes  $K \xleftarrow{R} KG(1^k)$  and block-cipher program  $E$  with block-size  $m$  and key-size  $k$ , generates the truth table from oracle access to  $E(K, \cdot)$ , and applies a Boolean canonical reduction on the truth table to produce a circuit  $E'$  that is semantically equivalent to  $E(K, \cdot)$ .

At this point, we distinguish between the block-size of cipher  $E$  which is  $m$  and our desired (bounded) input-size  $n$ . We establish that  $n \leq m$  and  $2^n \leq n^b$  for some user specified  $b$ . Where  $n = m$ , no padding of the input is necessary for  $E(K, M)$ . Where  $n = m$  is too large for a user chosen bound  $b$  (meaning there are not enough computational resources available to achieve truth table elaboration or the reduced sum-of-products circuit

derivation), an input size reduction is necessarily in order to meet the efficiency requirements for a polynomial bounded circuit size on  $E'$  or polynomial time speed for  $O$ . Where  $n < m$ , we must choose whether to pad with  $m - n$  zeros or to pad with a (randomly) chosen  $m - n$  bit string. We assume padding with 0 for simplicity at this point but point out that our plaintext message space is now  $\{0,1\}^n$  as opposed to  $\{0,1\}^m$ . The security ramifications where the adversary knows that a (possibly) reduced (virtual) block size is being used is a separate but related discussion to whether the adversary can recover the key  $K$  when given the source code (gate structure) of  $E'$ .

Let circuit  $E' = O(T_{EK}) = O(K, E)$  be an obfuscation of the encryption program  $E$  with embedded key  $K$  (i.e.  $E'$  retains the functionality of  $E(K, \cdot)$ ) where  $T_{EK}$  is the truth table of  $E(K, \cdot)$ . Assume  $m$  is the input size of  $E$  and  $n$  is the virtual (unpadded) input size of the plaintext where  $n \leq m$  and let  $2^n \leq n^b$ , for some user specified  $b$ . Let  $T_{EK}$  be generated through the PPT simulator  $S^E$ .

Then:  $O$  is a  $b$ -bounded input-size private-key block encryption program obfuscator for the class of programs  $\{E_n\}_{n,k,b \in \mathbb{N}, n \leq m, 2^n \leq n^b}$ , for any  $E \in \mathbb{E}_{k,m}$

Given  $E \in \mathbb{E}_{k,m}$  and  $K \xleftarrow{R} KG(1^k)$

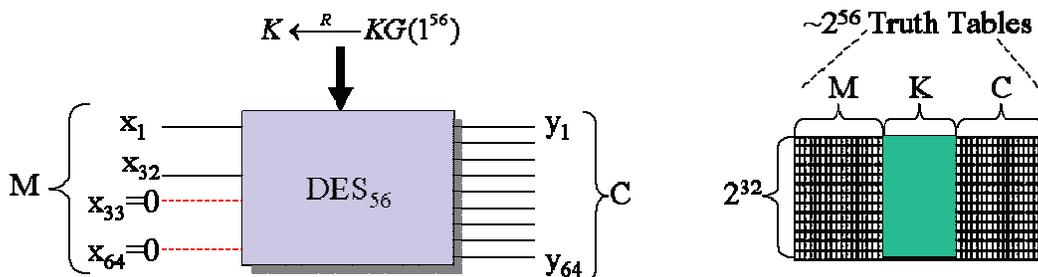
**Step 1.** Acquire an efficient implementation of  $E, S^E$ , to use as oracle emulation.

**Step 2.** Generate the truth table for  $E(K, \cdot), T_{EK}$  by running  $S^E$  on all  $2^n$  inputs of  $E$ , where  $n$  is related to the polynomial efficiency bound  $b$ . Where  $n < m$ , pad each input with  $m-n$  zeros.

**Step 3.** Create circuit  $E'$  by applying the algorithm for canonical complete-sum of products to  $T_{EK}$ .  $E' = \sum_{i=1, \dots, n} \pi_i$  is in disjunctive normal form (DNF) where each product  $\pi_i$  is a conjunct of literals and each literal is either an input variables  $x_j$  or its negation  $x_j'$  ( $1 \leq j \leq n$ ). Minimize  $E'$  via minimal-sum of products algorithm such as Blake's reduction based on Shannon's recursive expansion.

From Theorem 4,  $E'$  has the same characteristics based on its construction and meets requirements for semantic equivalence, efficiency, and perfectly secure obfuscation.

Consider, for example, that we could easily encrypt the output of our sensor from Figure 52, p. 66, using an embedded-key DES program because the sensor outputs 64 bits of data at a time (which matches the block input size of DES). We could then send the encrypted computational result  $DES_K(sensor(x))$  back to the processing facility, decrypt the output using the private key  $K$ , and then analyze the true sensor data. The only stipulation given under Theorem 4 is that we have computational resources related to the bound  $b$  such that  $2^{32} < 32^b$ . The primary limiting factor is the input size of the sensor since the size of circuit is related only polynomially to the number of outputs (which would be a factor of the encryption algorithm  $E$ ). If there are adequate computational resources to accomplish the truth table enumeration for the 32-bit input / 64-bit output matrix, then circuit  $E'$  can be constructed and a perfectly secure key-embedded circuit can be used in the sensor.

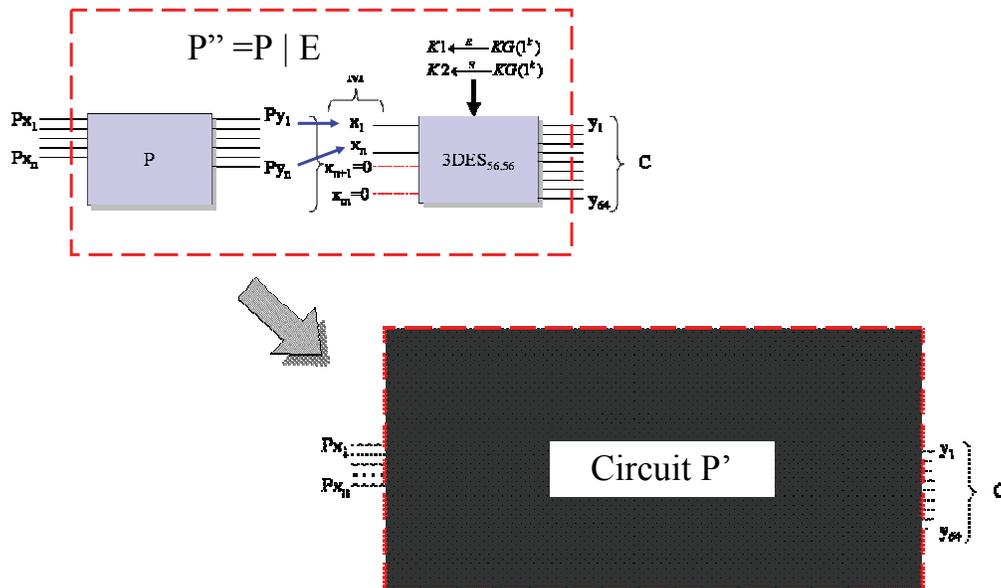


**Figure 71: Bounded-Size Input DES**

Assume that the output of the sensor described in Figure 52 were 32 bits instead of 64. We now must consider that only 32 bits (not the total 64 bit block size) of DES are under consideration. Figure 71 illustrates the issue of key recovery attacks and puts this in perspective of a DES program that takes messages that are 32 bits long. Here, the job of the adversary is to find the one truth table out of the approximately  $2^{56}$  possible truth tables (excluding those based on weak keys) that is based upon the specific  $K$  that is embedded in  $E'$ . For our specific sensor example, each truth table is a possibly  $2^{32}$  enumeration (versus a  $2^{64}$  enumeration) of entries corresponding to each message/ciphertext pair.

The obfuscators defined under Theorem 4 need only produce one of these truth tables in order to embed the key with perfect semantic protection in the circuit  $E'$ . The adversary (on the other hand), must enumerate up to  $2^{56}$  such truth tables in order to use the circuit  $E'$  to pinpoint the particular key  $K$  embedded within. Because of the construction process for circuit  $E'$ , which is based only on the input/output relationships of an embedded-key encryption operation, the adversary cannot discover the key  $K$  by examination of the actual gates of circuit  $E'$ . In fact, the gates of  $E'$  yield only semantic information concerning the input/output behavior of  $E_K$ , and nothing more. The adversary can do no more than observe input/output pairs which are obtained from execution of  $E'$  itself: this describes both the intuitive and theoretical notion of a virtual black box.

Given a possibly reduced message space, we relate the security of the circuit  $E'$  more to the key-space of DES than to the reduced message space  $2^n$  versus  $2^m$ . We can leverage this observation and replace the  $DES_{56}$  program with  $3DES_{56}$ ,  $AES_{128}$ ,  $AES_{512}$ ,  $RSA_{512}$ ,  $RSA_{1024}$ , or even an  $RSA_{2048}$  variant. In each replacement just mentioned, the efficiency of the obfuscator under Theorem 4 given a bounded input size (32-bits in our example) increases only in relationship to the additional running time incurred by the oracle for each prospective encryption algorithm to generate one truth table. The circuit size of  $E'$  does not vary based on the encryption algorithm chosen other than a linear variation based on additional output bits (64-bits versus 128, 512, 1024, etc.). We can use public key encryption algorithms under this same construction with both public and private keys held private—especially in the computational model of a sensor net because the execution environment of the program (the sensor) does not require decryption of the data it is processing.



**Figure 72: Fully Generalized Bounded Input-Size Program Obfuscation**

#### 5.7.4 Protecting Bounded Input-Size Programs with Easily Learnable Input

Consider now a sensor that takes in 32 bits of data and produces 32 bits of input: an adversary may observe much less than  $2^{32}$  input/output pairs of the sensor in order to adequately determine the programmatic intent of the sensor and therefore find an (effective) way to subvert it. Theorem 5 provides a basis to consider any bounded input-size program  $P$  that has (easily) learnable input/output patterns versus one-way relationships (like  $DES_{56}$ ) that we assume to have provably hard-to-learn I/O. Figure 72 gives a notional / specific view of this construction using a program  $P$  and a  $3DES$  encryption algorithm. In this construction, we create circuit  $P'$  as a concatenation of the output of program  $P$  with a data encryption cipher  $E$  (which is a  $3DES$  cipher that uses 2 keys in E-D-E relationship). As illustrated,  $P$  is a function  $P: \{0,1\}^{|\mathcal{X}_P|} \rightarrow \{0,1\}^{|\mathcal{Y}_P|}$  and  $E$  is a function  $3DES_{K_1,K_2}: \{0,1\}^{64} \rightarrow \{0,1\}^{64}$  with two embedded keys. We assume the output size of  $P$ ,  $|\mathcal{Y}_P|$ , is less than or equal to the input size of  $E$  (which for  $3DES$  is 64 bits). The circuit  $P'$  is a concatenation of  $P$  and  $E$  that then becomes a virtual black box, such that for all input  $x$ ,  $P'(x) = 3DES_{K_1,K_2}(P(x))$ .

In Theorem 5, we extend the results of Theorem 1 (which is a provable black box construction) and incorporate a provable white box construction based on VBB. The constructions of Theorem 5 provably meet the definition of full intent protection under our Definition 6. To distinguish our approaches, the circuit randomization methodology we define in Section 5.6 is not VBB-based, but rather random program model based. We note that Ostravsky and Skeith define similar public key encryption-program-padding obfuscators in [171] with follow on work by that implements such constructions in obfuscated mixnet programs. We provide now the definition and theoretical construction that would take any bounded-input size program  $P$  with easily learnable I/O and concatenate the output of that program with an embedded-key strongly pseudorandom encryption algorithm (producing  $P'$ ). Then we white box protect that program with canonical circuit reduction to produce  $P'$ . We only specify the symmetric/private-key block cipher variant and follow the construction for obfuscated mixnets given in [195].

*We let  $P \mid E_K$  refer to the concatenation of program  $P$  with the program  $E$  such that  $(P \mid E_K)(x) = E_K(P(x))$ , for all  $x$ . Let  $P$  be defined as function  $P: \{0,1\}^n \rightarrow \{0,1\}^{|\mathcal{Y}_P|}$  and  $E: \{0,1\}^k \times \{0,1\}^m \rightarrow \{0,1\}^m$ . Let  $P \mid E_K$  for encryption algorithm  $E$  with embedded key  $K$  be defined as  $P \mid E_K: \{0,1\}^n \rightarrow \{0,1\}^m$ .*

**Definition 18. (General White/Black box Obfuscator for Bounded Input-size Programs)** For PPT algorithms  $KG, E, D, O$ , obfuscator  $O$  provides perfectly secure obfuscation for the class of  $b$ -bounded programs  $\{P_n\}_{n,k,b \in \mathbb{N}, n \leq m, 2^n \leq n^b}$  where  $P \in \mathcal{P}_n$  if:

1. **Private Key Encryption:**  $(KG, E, D)$  defines a pseudorandom private-key block encryption scheme with block-size  $m$  and security parameter  $k$  under Definition 2.
2. **Semantic Equivalence:** Given  $K \xleftarrow{R} KG(1^k)$  and program  $P \in \mathcal{P}_n, \forall x, P(x) = D_K(P'(x))$  where  $P' = O(K, P, E)$ . Furthermore,  $\forall x, P'(x) = E_K(P(x))$ .
3. **Generality:**  $(|\mathcal{X}_P| = n) \leq m$ , for all  $E \in \mathcal{E}_{k,m}$  under Definition 17
4. **Efficiency:** There is a polynomial  $l(\cdot)$  for every  $P$  in  $\mathcal{P}_n, |O(K, P, E)| \leq l(|P|)$
5. **Perfectly Secure Obfuscation:** For any PPT  $A$ , there is a PPT simulator  $S$  and a negligible function  $\alpha$  such that for every  $n, b \in \mathbb{N}$  where  $2^n \leq n^b$ , and for every  $P \in \mathcal{P}_n$  and for every  $K \xleftarrow{R} KG(1^k)$

$$\left| \Pr[E, A(O(K, P, E)) = 1] - \Pr[E, S^{E_K}(1^n) = 1] \right| \leq \alpha(n)$$

**Theorem 5.** Perfectly secure obfuscators exist for  $b$ -bounded input-size programs with easily learned I/O relationships.

**Proof:** Our proof is by construction. We give a three-step obfuscator  $O()$  that takes as input  $K \xleftarrow{R} KG(1^k)$ , a block-cipher encryption program  $E$  with block-size  $m$  and key-size/security parameter  $k$ , and a  $b$ -bounded program  $P$  with input size  $n$  and output size  $|y_P| \leq m$ , and where  $2^n \leq n^b$ , for some user defined  $b$ . Let circuit  $P' = O(K, P, E)$  be an obfuscation of any general program  $P$  with these constraints such that  $\forall x, P(x) = D_K(P'(x))$  and,  $\forall x, P'(x) = E_K(P(x))$ .

Construct  $P' = O(K, P, E)$  in the following manner:

**Step 1.** Given  $K \xleftarrow{R} KG(1^k)$ , let  $P'' = P \mid E_K$ . Acquire an efficient implementation of  $ORACLE_{P''}$  to use as oracle emulation.

**Step 2.** Generate the truth table  $T(P'')$  by executing  $ORACLE_{P''}(x) = E(K, P(x))$  for all  $2^n$  possible inputs  $x$  of  $P$ . Where  $|y_P| < m$ , pad the output of  $P(x)$  with  $m - |y_P|$  zeros.

**Step 3.** Create circuit  $P'$  by applying the algorithm for canonical complete-sum of products to  $T_{P''}$ , as defined in Theorem 3. Minimize  $P'$  via a standard 2-level Boolean circuit reduction technique.

Then:

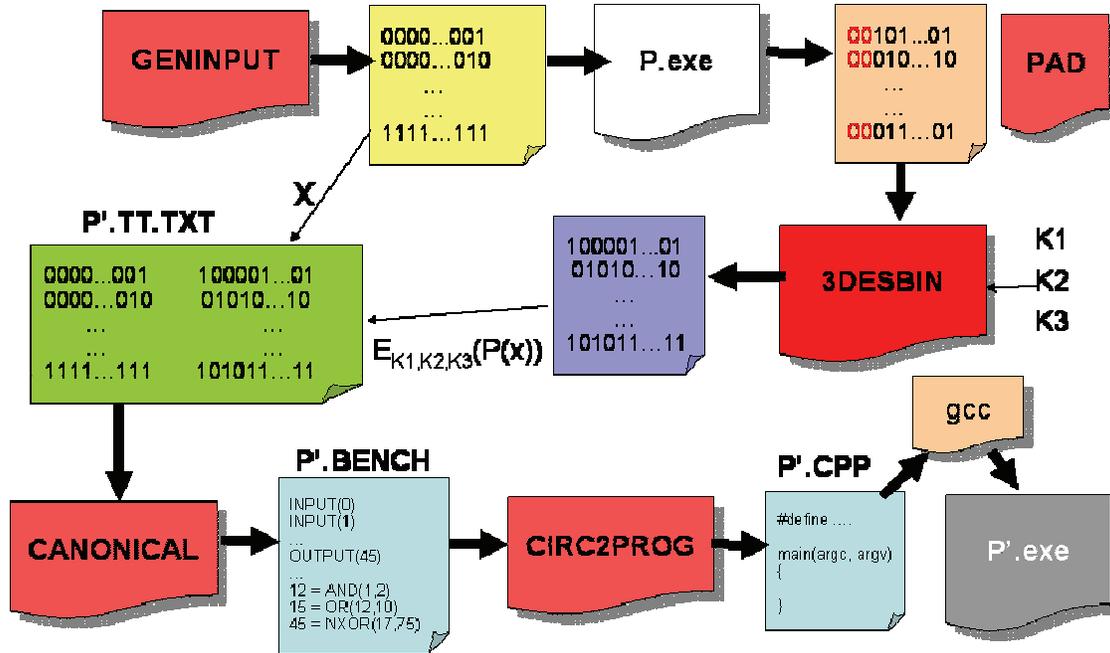
1.  $E$  is hard to learn and therefore  $P''$  and  $P'$  are hard to learn from black box observation *alone*. However, recovery of any *intended* output of  $P$  (which is easy to learn) is possible because  $\forall x, P(x) = D_K(P'(x)) = D_K(E_K(P(x)))$ . Thus, the semantic equivalence between  $P'$  and  $P$  is established.
2.  $P'$  is a perfectly secure obfuscation with respect to  $P$  and the embedded-key encryption algorithm  $E_K$  because  $P'$  is produced only from oracle access to  $P'' = P \mid E_K$ .
3.  $|P'|$  is poly- $(n)$  given bound  $b$ .
4.  $O$  is a general, efficient obfuscator for *any* program  $P$  that runs in poly- $(n)$  time, given a bound  $b$  related to the input size of  $P$ .  $P'$  is roughly equivalent in efficiency to  $P$ . The minimal SOP expression of  $P'$  is polynomially equivalent in size to  $P$  related to some bound  $b$ , because  $|P'| \leq n^b$ . Note that the size characteristics of  $P'$  are related to the input size of  $P$  and not the possible input size of  $E$ .

## 5.8 Implementation Work

We describe briefly the circuit construction and manipulation architecture that supports our research results. Figure 73 provides a high-level overview of the software pieces in our architecture that implement perfect white box protection methodology outlined in Theorem 5. We have built several software pieces (GENINPUT, PAD, CANONICAL, CIRC2PROG, BENCH, etc.) that work together as an end-to-end program encryption architecture. We describe their interactions next.

We first provide a capability to generate binary input, either padded or unpadded, for some inputs size  $n$  (GENINPUT). Given a generic program  $P$  ( $p.exe$ ) with bounded input size, we enumerate all inputs for  $P$  using GENINPUT and then execute  $P$  ( $p.exe$ ) on all inputs. We take the (binary) output of  $P$  and provide a padding mechanism (PAD) to configure the output of  $P$  to match the candidate encryption algorithm  $E$  (3DESBIN in Figure 73). Using an encryption algorithm, we generate a pseudorandom key choice and keep the key private. In our illustration, the 3DES algorithm uses three keys, so we provide three embedded private-keys to the application 3DESBIN. Using the (padded) output of  $P$  ( $p.exe$ ) generated by elaborating all possible inputs to  $P$ , we now execute all outputs of  $P$  on the encryption algorithm (3DESBIN). Using the input of  $P$  ( $p.exe$ ) and the output of  $E$  (3DESBIN), we now have a full truth table relationship for  $P'' = E(P(x), K)$  for all  $x$ .

Using the truth table relationships for  $P$  ( $P.TT.TXT$ ), we use a program **CANONICAL** to generate a **BENCH** circuit specification based on the complete sum-of-products form. Future work will address reductions on this specification. This specification now represents a fully black box and white box protected version of  $P$ , with respect to recoverability of encrypted output related to  $E$ . We then use the circuit specification ( $P'.BENCH$ ) as input to our program **CIRC2PROG**. This program takes a generic **BENCH** specification and produces a C++ specification that implements the same I/O functionality. This C++ specification ( $P'.CPP$ ) can then be compiled using the native O/S compiler to produce an executable program ( $p'.exe$ ).  $p'.exe$  represents a provably secure, efficient given bounded input-size, white box and black box protected version of program  $p.exe$ .



**Figure 73: Architecture for General Program Intent Protection ( $P.exe \rightarrow P'.exe$ )**

We discuss our implementation work further in Appendix D and note that this architecture provides an end-to-end native binary transformation for any generic *executable* program  $p.exe$  to a perfectly intent protected *executable* version  $p'.exe$ .

## 5.9 Chapter Summary

We present in this chapter a number of novel techniques for end-to-end program encryption to support provably secure program intent protection. We give some of the first results in the field that tie cryptographic primitives such as data encryption and randomization directly into obfuscation. We also provide one of the first proposed obfuscation security models with a measurable cryptographic basis (random programs).

Our program encryption results provide several positive indications that we can intent protect program securely. We pose three different program intent-protection methodologies in this chapter and summarize their results in Table 20. Section 5.4 introduces a perfectly secure black box obfuscation approach that is general and efficient for all programs (depicted as **Semantic A | B** in Table 20). Of course, adversaries will exploit the knowledge of our method (concatenating an encryption cipher to the protected program) will and utilize that during white box analysis attacks.

Section 5.6 introduces a methodology (depicted as **Randomized A | B** in Table 20) based on the random program security model (introduced in Section 5.5) to combat such white box

analysis. The approach uses a non-VBB method of circuit randomization to make the resulting protected circuit (program) hard to distinguish from a random circuit. If an adversary can tell nothing more from a protected circuit than the information gained from analysis of a random circuit, then we achieve white box protection. The method is both general and efficient, but for small sizes, the probability increases that an adversary can discern seams between two concatenated circuits.

Section 5.7 introduces a second white box methodology related to VBB (depicted as **Canonical A | B** in Table 20). In this approach, we absolutely hide all semantic information of the concatenated circuit (A | B) for general programs. However, the methodology only works for programs with bounded input-size. This method holds great promise for a relevant class of programs with typically small input-size: they can enjoy perfectly secure intent protection. This method also provides basis for provably hiding small data constants within a program (including embedded keys). We demonstrate that for encryption algorithm with bounded-input size, we can securely embed a private key using this methodology.

**Table 20: Program Encryption Results Overview**

	$X_A$	$Y_A = X_B$	$Y_B$	Security	Practical
<b>Canonical Form A   B</b>	Reveals	There is a very high improbability that any subcircuit of C contains any representation of variation of the circuit A: $\{0,1\}^{X_A} \rightarrow \{0,1\}^N$	Reveals	Strong	Difficult to create (exponential)
<b>Randomized A   B</b>	Reveals	The likelihood of $Y_A$ or $X_B$ being recognized is based upon an intractability argument for circuit analysis. There is some probability that the end of A or the beginning of B can be calculated based on a randomized equivalent version of A   B, but the probability of detection decreases only as the circuit size of C increases.	Reveals	Size dependent	Polynomial or linear time to create
<b>Semantic A   B</b>	Reveals	It is very probable that (with the knowledge of how C is constructed via A   B), subcircuits or some variation of the subcircuits A: $\{0,1\}^{X_A} \rightarrow \{0,1\}^N$ and B: $\{0,1\}^N \rightarrow \{0,1\}^M$ can be distinguished with A   B	Reveals	Weak	Very easy (linear) time to create

We provide conclusions concerning our research work next.

## CHAPTER 6

### CONCLUSIONS

Mobile agent security presents us with many hard problems to solve. Most researchers attribute security as the top reason why mobile agents failed to achieve commercial or wide scale implementation. The malicious host problem is not unique to mobile agents though. The problem finds parallel manifestation in other secure function evaluation schemes. We motivated our initial work with mobile agent security by trying to solve periphery problems that occurred in specific implementations. Our initial labors led us to consider multi-agent architectures for solving certain nagging problems related to colluding malicious hosts. The question of protecting the agent's privacy, even when architectural solutions can enforce other specific security requirements like integrity, always come back to haunt you. Based on the most promising approaches for providing that level of security, we naturally focused our work on secure multi-party protocols and schemes for secure function evaluation.

As our research progressed, we considered the current work on obfuscation and their associated impossibility results. This painted a bleak landscape for applying obfuscation techniques to mobile agents with any expectation of provable security. Where we found our greatest impact was considering obfuscation and obfuscation security under a different model. Particularly, we understood that program intent protection was the primary security goal that mobile agents required. If malicious parties cannot alter the code or game the input to produce their (desired) result or output, then we have won the code protection game in malicious environments and any other security requirements we enforce are bonus.

Our main contribution involves several definable security properties that we produce concerning code security in remote environments. In spite of impossibility results, we created a security model that describes code protection properties (particularly in mobile environments) with provably security. The fruit of our efforts culminate in the research results we present here and the several proofs we give for provable code security in Chapter 5.

Specifically, we prove that black box protection is general, provably secure, and accomplished (relatively) efficiently. Black box code protection derives from the underlying semantic security of data encryption algorithms themselves. As far as we know, our semantic encryption transformation methodology was the first black box obfuscation approach with provable security properties and strong cryptographic basis. It is simple in its design because it involves composing the output of one (protected) program functionality with a (secure) data encryption cipher.

As our research progressed further, the next natural question that arose was, "If you compose two programs together, how can you hide the seam between the two?" The answer to this question relies on the ability to provide (provably) secure white box protection. A provably secure white box protection mechanism is equivalent to providing a provably secure virtual black box code version. Since theoretical VBB is a flawed model for describing obfuscation security strength, our research led us to consider cryptographic primitives such as randomization as a basis for security. With randomization and the assumed existence of pseudorandom number generators, we make an appeal for the existence of random programs, random circuits, and pseudorandom program generators.

In describing obfuscators that incorporate randomization into their program protection model, we further considered other natural cryptographic constructions including permutation/substitution ciphers. We leverage such algorithms used by strong symmetric ciphers based on diffusion and confusion of data and key material and produce a methodology of randomizing, sub-circuit substitution and replacement, circuit encryption obfuscators. Using the random program model, we demonstrate that an indistinguishability argument is the basis for semantically strong white box protection. We also specify how we can employ circuit randomization techniques to produce the white box property.

The question of whether we can provably hide the seam between one program and another, however, relates directly to the size of a program itself. Randomness properties emerge only as string sizes increase. Likewise, we wanted to prove that no seam detection between two programs is possible at all, even when program sizes are small or one program (the encryption program for example) has distinct properties that no degree of randomization can remove. In seeking to hide the seam between two programs, our investigation produced a perfectly secure white box protection method that is applicable to a relevant number of program classes. Unlike our first white box approach, we prove under VBB assumptions that the information contained in white box obfuscated circuit leaks no information about the original circuit, other than its input / output relationships.

Our perfect white box protection scheme is general, but unfortunately not efficient for all programs as our black box technique and randomizing white box technique are. However, when we incorporate our black box methodology, we can prove perfect semantic security for an obfuscated circuit or program. This result culminates from a long history of intermediary findings, but is one that can allow the development of future, secure mobile agent applications. For programs with bounded input-size, white box protection is both efficiently feasible and provably secure.

The provably secure white box results also provide a companion result significant to the security community. This approach proves that we can hide an embedded key within a program. Such a technique foundationally addressed how to convert a private key system into a public key system. The only stipulation, again, is that we must assume reasonable, bounded input sizes for the data cipher. Given such assumptions, though, this technique is one of the first demonstrated approaches for securing embedded-key ciphers and applications.

## APPENDIX A

### COMPREHENSIVE SURVEY OF MOBILE AGENT SECURITY

We provide in this appendix a comprehensive review of mobile agent security. We review first defensive mechanisms that protect the host (Appendix A.2) and agent (Appendix A.3). We review agent data-protection mechanisms in Appendix A.4 and discuss the integration of secure multi-party computation techniques with mobile agents in Appendix A.5. We discuss multi-agent approaches and their applicability to enhancing mobile agent security in Appendix A.6. We cover background material and related trust infrastructure research in Appendix A.7. We give an overview of software protection techniques related to mobile code privacy in Chapter 3. We also provide technical reports that catalogue mobile agent security techniques in [222] and describe integration of trust integration in [152].

#### A.1 Evaluating Agent Security Mechanisms

Section A.4 details numerous security mechanisms relevant to protecting partial results. As we look towards mechanisms that meet various requirements for securing both mobile agents and hosts, we consider the evaluation criteria useful for reviewing such frameworks. Mechanisms for security in *any* field introduce overhead and we should weigh heavily such considerations for mobile agent security. The highest levels of security often bring with them the highest overhead in terms of cost, lost flexibility, and performance degradation.

A mobile agent system may not require every form of protection offered. Some protection schemes are mutually exclusive as well—for example, free-roaming itineraries preclude solutions that require knowledge of all hosts to be visited. Some defense mechanisms are only notional and have no current usable implementation and some still have serious issues that limit their full realization. As such, metrics need to be considered in a taxonomy for requirements. Evaluation criteria can help determine the effectiveness of one criterion over another and address issues of efficiency, integration, and cost. There are three considerations when examining agent-based security mechanisms in the mobile environment, discussed next.

First, ***what is the performance cost of any added security mechanism?*** This cost can include increased size of the migrating agent, increased network bandwidth, increased number of messages, increased number of agent migrations or host visits, increased host-processing time, increased computational complexity, and increased overall job time. For example, Gunupudi and Tate [223] evaluate four different protection mechanisms that provide data integrity and encapsulation. They characterize computational time and data growth size for each scheme and provide simulation-based evidence that a new scheme (modified set authentication codes) provides greater efficiency in certain dimensions. The hash chaining mechanism (discussed in Section A.4.8), for example, is expensive in data growth when compared to other approaches.

Cryptographic primitives come with various overhead. Symmetric key cryptography offers the greatest efficiency in terms of computational processing for mobile agent security, but it incurs overhead for key distribution and maintenance. Asymmetric key cryptography requires greater computational overhead but is easier for key distribution itself. On the other hand, public keys require certificate verification and this normally incurs the overhead of a public key infrastructure (PKI) with some form of certificate authority (CA) for scalability.

Sobrado [224] compares the security overhead of keying mechanisms in two different agent protection schemes. His work analyzes one-time symmetric keys versus asymmetric public/private keys and the associated cost of computing encryption and signatures over parts of the agent data. Sobrado makes a case for the flexibility of the public key approach in this case, citing the fact that the originating host in the symmetric case can only verify code and data but honest hosts could provide detection for the agent when asymmetric crypto is used.

In terms of evaluating the actual security strength for a given mobile agent framework, Fischmeister *et al.* [225] provide security analysis of three separate mobile agent frameworks

(Aglets, Jumping Beans, and Grasshopper). Security weaknesses exist in all three middleware systems including authorization attacks, code repository attacks, interface attacks, runtime system calls, and trusted code base attacks. Likewise, Altmann *et al.* [66] compare performance and security tradeoffs of various mobile agent frameworks while Milagres *et al.* [226] give an example of security analysis for an existing multi-agent architecture. Bellavista *et al.* [64] review security mechanisms for the Condordia, Voyager, Aglets, D'Agents, Ajanta, MARISMA-A, SOMA, Grasshopper, and NOMADS mobile agent system. Roth [109,112] performs protocol analysis of several data protection mechanisms to show their security vulnerabilities by proof and then proposed remedies for each protocol.

Agent architectures often use multiple classes of agents to enforce security, including our own concept for data integrity via multiple cooperating agents [142, 227]. Performance issues involve additional message overhead between cooperating agents and increased migrations for any mobile agent classes involved. We find multi-agent system evaluations in current research and detail them in Section A.6. To decide whether mobile agents are more efficient than static multi-agents, we can evaluate different architectures appropriately. For example, O'Malley *et al.* [228] conduct a simulation-based appraisal to determine whether multiple static agents are better than multiple mobile agents. The performance results in their simulation indicate mobile agents offered slight but reasonable advantages over static configurations of agents performing similar tasks. What Kotz and Gray [59] give as assumptions (*mobile agents are advantageous for performance reasons*), Gray *et al.* [10] also support later via analysis and simulation.

**Table 21: Security Evaluation Criteria**

<ul style="list-style-type: none"> <li>◆ Location of security mechanism (host, agent, trusted third party, agent owner)</li> <li>◆ Form of security mechanism (centralized, distributed)</li> <li>◆ Individual host execution time</li> <li>◆ Overall job execution time</li> <li>◆ Flexibility of cryptographic approach</li> <li>◆ Vulnerabilities of security approach</li> <li>◆ Size of agent data state growth</li> <li>◆ Communication/network bandwidth</li> <li>◆ Number of messages and migrations</li> <li>◆ Complexity of solution</li> <li>◆ Requirements coverage of security mechanism</li> <li>◆ Expressiveness of the security policy</li> <li>◆ Ease of integration with existing security</li> <li>◆ Customizability</li> <li>◆ Agent user, framework implementer, host operator transparency</li> <li>◆ Physical cost</li> <li>◆ Learning and adaptability based on historical data</li> <li>◆ Auditing capability</li> <li>◆ Platform independence</li> </ul>
--

The second consideration when evaluating agent security mechanisms is: ***What is the increased physical cost?*** Classically, the use of trusted hardware solutions (detailed in Section A.3.19) represents the highest level of protection achievable for mobile agent applications. Unfortunately, the cost of deploying such solutions in a large ubiquitous network environment like the Internet is not feasible and cost prohibitive. Other application domains, like the military or specialized corporate settings, may be able to support such costs. Physical cost may also be measured in terms of licensing and maintenance of agent frameworks themselves, especially when security solutions may involve tasks such as proof construction, low level code manipulation (assembly level), or formal analysis.

Lastly, the final consideration is: ***What is the increased reliance on proprietary solutions?*** In many cases, mobile agent security mechanisms are typically monolithic and without

consideration for integration into real world infrastructures. Trusted hardware, trusted third party software services, non-standard cryptography approaches, and the introduction of fixed non-interoperable software architecture are all roadblocks to greater acceptance of mobile agents in a wide domain. However, such solutions may be the only way implementers can achieve certain levels of security when deploying a mobile agent application. We present a summary of criteria in Table 21 for reference and use categories found in other tutorials such as [229].

Though not exhaustive, criteria in Table 21 form a reasonable collection of considerations for application creators and system developers. We *implement* agent mobility (depicted as part of our taxonomy in Figure 12) by agent middleware and agent middleware *uses or implements* one or more security mechanisms. Middleware implementations address a subset of security threats and security requirements by using some set of security mechanisms. We now provide a review of these mechanisms—host-based protection in Section 2.3 followed by agent-based protection in Section 2.4.

## A.2 General Host Protection

The host platform defense against malicious mobile code is a combination of trade-offs. Host platforms find it difficult to discerning program intentions or rely on a trust relationship from unknown code. Mechanisms used to prevent malicious agent behavior can often restrict mobile code with good intentions while failing to discern and restrict hostile code [137]. Malicious code defense mechanisms fall into seven categories (listed in Table 22). Host middleware typically enforce specific security requirements (see Table 3, p. 9) by use of one or more of these mechanisms. Host protection ultimately seeks to limit the overall power of the execution environment while reducing the overall vulnerability of a host to a malicious mobile agent. Sometimes these are competing goals where we middleware must sacrifice one for the other.

**Table 22: Host Protection Mechanisms**

Section	Mechanism
A.2.1	Sandboxing (SBFI)
A.2.2	Safe Interpreters
A.2.3	Code Signatures
A.2.4	State Appraisal
A.2.5	Proof Carrying Code
A.2.6	Path Histories
A.2.7	Policy Management/Authentication

### A.2.1 Sandboxing (SBFI)

Sandboxing, as its name implies, provides a separate but protected place for unsafe code to execute in as it enters the domain of a remote host. A sandbox may confine code through type checking, properties of the language, and allocating code to protection domains [230]. Middleware can use fixed policies to limit the power given to an application within an execution environment through the sandbox. Wahbe *et al.* [231] provided early work concerning a software-based approach to isolate faults in lieu of hardware-based methods. Programs operate in isolated virtual address spaces without chance of influencing other programs except through specific cross-domain requests. By using software based fault isolation (SBFI), a developer can encapsulate a module's object code to prevent any references for addresses outside the fault domain.

The Java sandbox is a common SBFI implementation because it restricts allowed operations of remote mobile code (such as an applet) and erects a barrier between the code and host resources [232, 233]. As Figure 74 illustrates, an execution environment such as the Java Virtual Machine (JVM) can allow normal programs to be trusted with full (or normal) host access to system resources. Hosts consider sandboxed programs, like remote code received in the JVM, untrusted and give such programs only specific permissions for a subset of host resources. In JDK 1.2, all code is considered *untrusted* and subject to fine-grained access-control mechanism

[234]. This method implements a policy framework approach to sandboxing. Java is a favorite implementation environment for mobile frameworks [67] because of built-in capability for SBF and because each sandbox has its own set of privileges [70]. Middleware can also execute programs written in unsafe languages like C within a sandboxed environment to minimize their detrimental effect on the host environment [22]. Sandboxing can limit the effectiveness and power of a given application and possibly limit the usefulness of applications that utilize this approach. Sandboxes provide operating system specific protection from mobile code, but the design of mobile agent languages and the use of safe interpretive languages bolster an execution environment even more. We discuss these languages next.

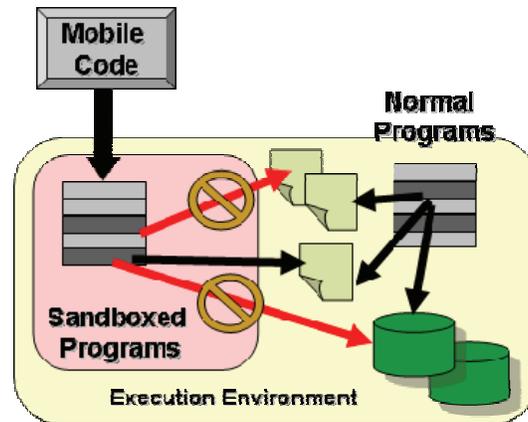


Figure 74: Sandboxing

### A.2.2 Safe Code Interpretation

Code *interpretation* offers better security features for remote code execution than *compiled* environments because middleware interpreters can examine instructions for their intended harmful effects before execution. We deem interpreters “safe” when fine-grained access control decides which statements to execute. Of course, interpreters are slower in general and developers consider languages like Java ill chosen for certain high performance applications on these grounds alone. Other scripting languages (like Tcl) can offer security primitives with greater functionality than can be found in typical systems languages [235].

We can think of safe interpreters as sandboxing with quarantine [70] in the sense that we execute commands in compartmentalized areas and alias commands to those that are safe by some policy definition. Figure 75 illustrates the Safe Tcl model [236] which utilizes this approach in the form of two different interpreters: one for trusted code and one for untrusted. The host gives the master interpreter normal levels of authority with system resources while it isolates untrusted applets to the safe interpreter. The system hides unsafe commands and cannot invoke them from within the padded cell. Instead, the commands have appropriate aliases assigned to them, which call real commands in the master interpreter. Protected commands, such as those that do file access or network communication, can have policy constraints associated with them for protective purposes. This protection approach mimics the kernel-mode of various operating systems where the system allows commands greater access to system resources that are not normally available in user-mode.

Safe code interpretation and sandboxing go well together and both are implemented within the Java specification and found in other scripted environments such as Tcl<sup>17</sup>. In Java, we compile source code into bytecode that allows the sandbox to perform certain runtime checks for security purposes. The JVM for example provides the following security features:

- ◆ namespace separation through the applet class-loader
- ◆ bytecode checking to make sure commands conform to the language specification

<sup>17</sup> Available: <http://www.tcl.tk>, October 2005.

- ♦ execution of system methods through a security manager, type-safe casting of references
- ♦ garbage collection to avoid explicit deallocation of memory
- ♦ automatic array bounds checking

Sandboxing and safe interpretation provide operating system and language level protection of the host, but they do not help establish trustworthiness of a given mobile agent program. For this purpose, authentication via signatures is necessary.

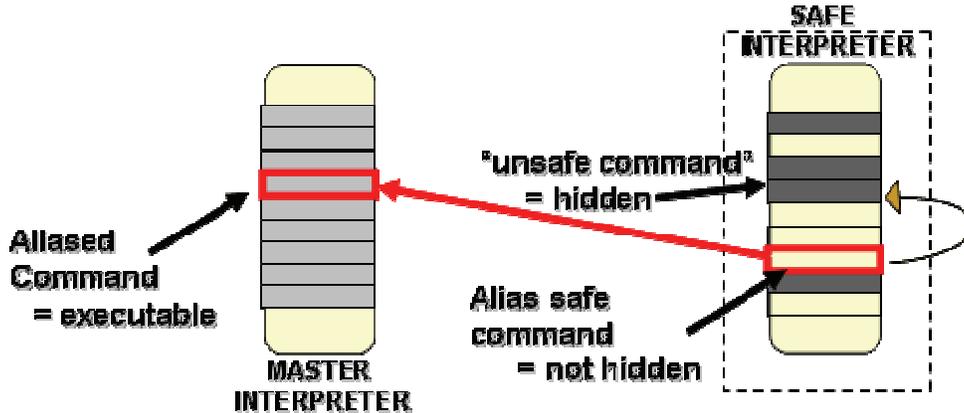


Figure 75: Safe-TcL Padded Cell Concept

### A.2.3 Code Signatures

Typically, the host cannot discern whether code is malicious. Though other mechanisms may inch towards that goal, hosts most likely rely on trust primarily from authentication of an agent's identity. By using digital signatures, the host can verify the identity of the agent or at a minimum the *signer* of the agent, which could be the author, transmitter, or owner of the code [237]. The Microsoft ActiveX framework originally introduced code signatures and signed components remain a standard part of the Java environment in the form of signed applets [238, 22]. As a drawback, verifying authenticity of the static code via the signature says nothing about the security or even *non-malicious* fault properties of the code except that one party trusts the *source* from which the code came from.

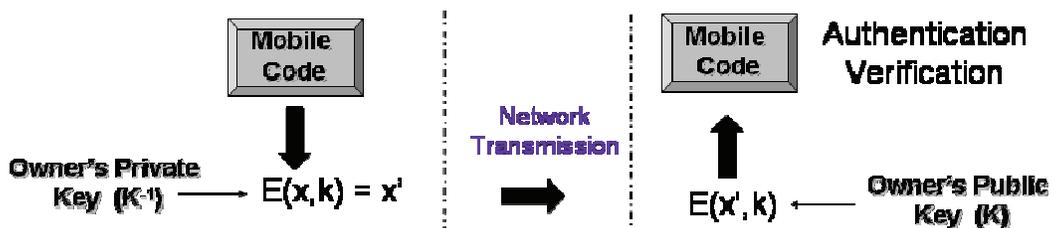


Figure 76: Simple Authentication

Developers can generate and use code signatures in several different ways. Public key cryptography signatures use a public/private key pair associated with a particular principle. As Figure 76 depicts, the code owner digitally signs their code by encrypting it with their private decryption key. On arrival, a prospective host verifies the signature by applying the public key of the sender. Hosts can also use signatures to verify the integrity of the code when used in conjunction with cryptographic strength one-way functions. Figure 77 illustrates how we can generate the hash of the mobile code and use it as a message digest sent along with the mobile code. After reception of the mobile agent, the host runs the same hash function. If the result equals the message digest that came with the agent, then host verifies integrity.

The executing host must somehow verify the public key of the signer: the developer can add a certificate to the contents of the mobile code package to accomplish this. A public key infrastructure (PKI) must be in place to scale the approach for large numbers of agent servers. In the browser model of today, an initial set of public keys are distributed with the browser itself and new keys must be self-authenticated without a PKI in place. Developers and application owners can use a slight variation with the message digest approach to prevent replays of prior code transmissions and to improve certainty of ownership. In this case, the owner encrypts the program and message digests together. The remote server decrypts the message and hashes the program. If the generated and received digests match, the code is authentic.

Note that in the mobile agent paradigm, the “network transmission” in Figure 76 and Figure 77 can represent a multi-hop traversal of the agent. A verified signature on the code does not guarantee an executing host can trust the mobile code—even non-malicious incorrect code produces harmful results when given full access to local host resources. The trust model with signed code is all or nothing in the sense that the executing host allows the code to run with some set of privileges once it authenticates the code. The middleware can establish policy statements for how to interpret valid code signatures: at a minimum, the system can imply some trust level between an agent originator and the remote host executor.

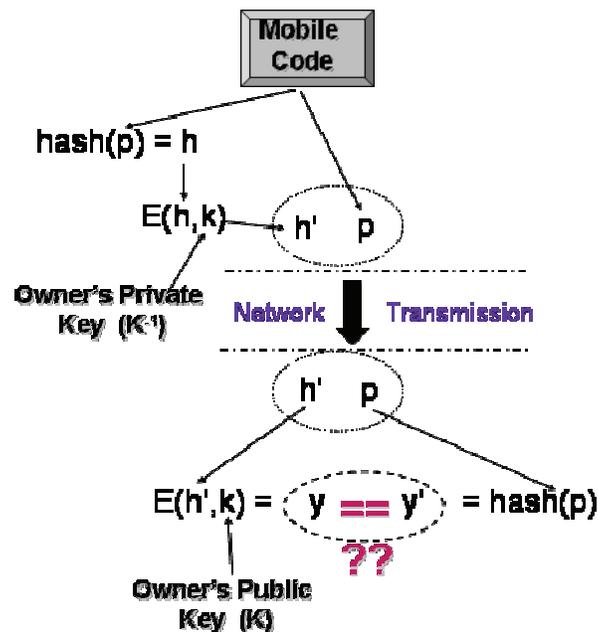


Figure 77: Integrity and Authentication

#### A.2.4 State Appraisal

The state of an agent dynamically changes as it traverses a network due to interactions with each host. One form of attack is to change values embedded within the state of the agent code or to alter code for malicious purpose. Farmer *et al.* [239] proposed a defensive mechanism for hosts in the mobile environment to verify state has not been altered during transit of the agent. Developers link appraisal functions to code based on invariant values. The hope is that the executing host can detect *illegal* alterations to other variant state values. The strength of this technique relies on the assumption that attacks will dangerously alter the state of the agent in *detectable* ways. Unfortunately, 100% detection is not possible, though high probability detection is possible if high overhead is acceptable. Figure 78 depicts the exchanges in the state appraisal model, which we describe next.

The architecture for the state appraisal mechanism is two fold. First, the host system *authenticates* the agent to determine the responsible principle for the code. Next, the host

performs *authorization* by first running the agent's state appraisal functions (seen as  $f_1()$  and  $f_2()$  in Figure 78) and then formulating a set of requested agent privileges based on evaluation of current agent state. After evaluation of the appraisal, a server can decide which permissions to grant. We tie invariant information to the agent code itself and utilize even simple techniques such as the sum of two variant values that must equal the same sum. State appraisal serves both to protect agents and hosts from malicious activity since alterations can be performed by malicious hosts to turn friendly agents into malicious agents that are forward to other hosts.

State space size can be large and designing appraisal functions that cover a majority of the possible attacks to the state is unrealistic. It is likewise difficult to provide invariant functions that cover less obvious alterations in the data state. Ultimately, a normal result may be indistinguishable from a maliciously produced result, making state appraisal techniques difficult to implement. When combined with other techniques, state appraisal can provide a simple method for covering the most *likely* or *important* alterations in a mobile code segment. Having authorization driven by an appraisal mechanism can greatly reduce vulnerability and operator fear in host frameworks that execute mobile code.

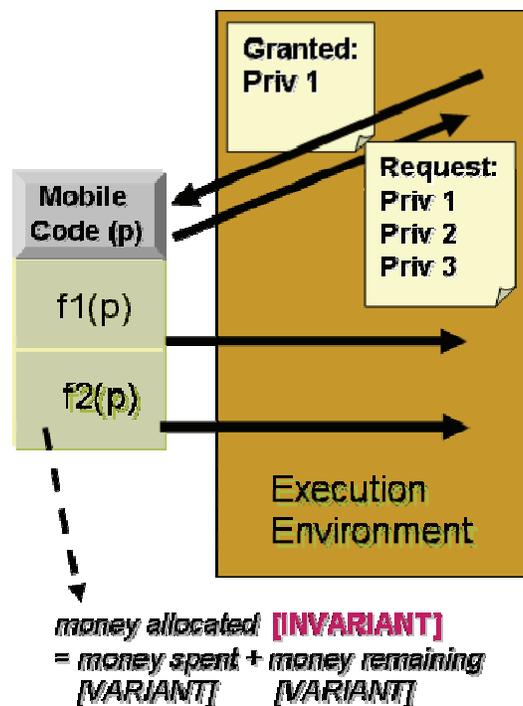


Figure 78: State Appraisal Technique

### A.2.5 Proof Carrying Code

Another form of code analysis technique that can be helpful to host defense is a formalized approach that defines safety characteristics of a program. Lee and Necula [240, 241] develop proof carrying code (PCC) with the idea that a remote host can verify the safety properties of the code about to be executed. In this process, much work is done up front to construct a proof that matches a security policy stipulated by a given agent server. Ultimately, an executing host uses the approach to prevent malicious agent execution. Appel [242] and Feigenbaum *et al.* [243] also contribute positions for using PCC for code protection.

Figure 79 depicts the overview of the PCC process as described by Necula [240]. Code executors first provide safety rules that programs must conform to. Because of the heterogeneous environment of a mobile agent transit, these rules may vary from host to host—not all hosts may be known in advance (limiting use of freeroaming agents). *Code producers* (agent originators) certify their code based on security predicates and *code consumers* (agent

servers) validate that proof before allowing the program to execute. The executing host would detect malicious alterations to the code and prevent the code from passing verification—thus preventing execution. PCC requires a formulatable safety predicate from the original mobile program that embodies the semantic meaning of the program. The code developer or application owner generates the predicate by following axiomatic rewriting rules and uses it to establish that a given proof indeed corresponds to a given program. We describe safety rules in first order predicate logic (based on Edinburgh logical framework) and tie them to compiler specifications of the underlying architecture. The rules are a formal description of data-representation invariants kept constant by a given program and the calling conventions a foreign function meets. The safety proof essentially guarantees that the code meets invariants and calling conventions.

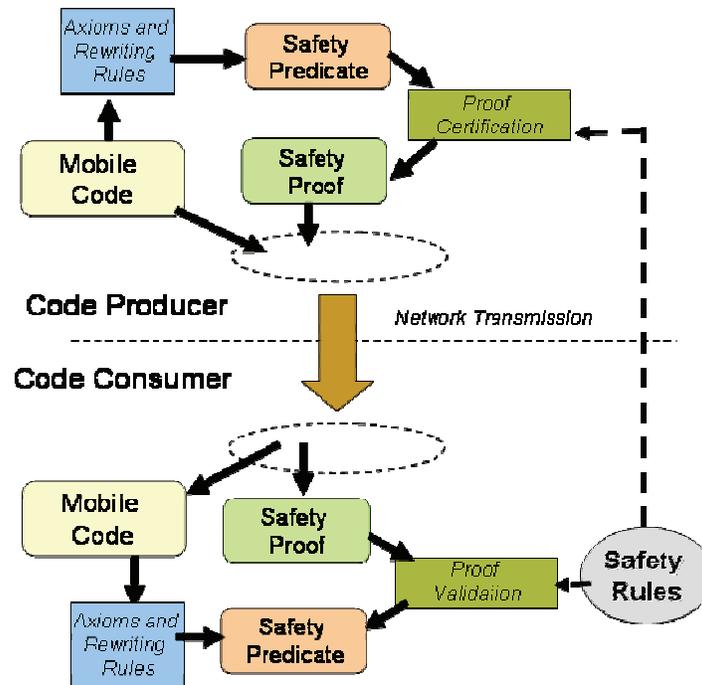


Figure 79: PCC Framework

We can see PCC as an extension to a signature, but instead of integrity or authentication, we verify security end-to-end. PCC is also similar to state appraisal: the code producer must create additional items that establish program correctness. The advantage of provable code is that it reduces expensive run-time checks necessary in an interpretive environment. The disadvantage of PCC is making proof development simple—a task not easily accomplished. In order for the mechanism to be successful, the technique requires a standard way of expressing the security policy in a formal manner and requires a limitation on proof size. Lastly, PCC ties safety rules to underlying hardware and thus places a heavy proprietary burden within the agent execution environment.

### A.2.6 Path Histories

Another method of host protection exploits the history of the agent's migration to evaluate the relative trustworthiness of the agent result. Ordille [75] suggests agent trust level can correspond to the minimum trust established by previously visited hosts (embodied in the agent itinerary) and the agent trust level itself. To apply path history evaluation, hosts add signed entries to the agent that contain the identity of the server and the next host in the planned itinerary. Figure 80 depicts that each server signs a new entry in a non-repudiatable log that links the path to itself and the next visited host. The log itself may be corrupted and we can use other measures to detect such alterations [79]. Wilhelm *et al.* [76] suggest the incorporation of trusted hardware to guard the agent itinerary while Westhof *et al.* [244] provide software based mechanisms to protect the

itinerary from colluding malicious hosts working in partnership. We can use other *host* defense mechanisms that use chaining relationships for path history protection and we discuss these in Section A.3.16 and A.3.18.

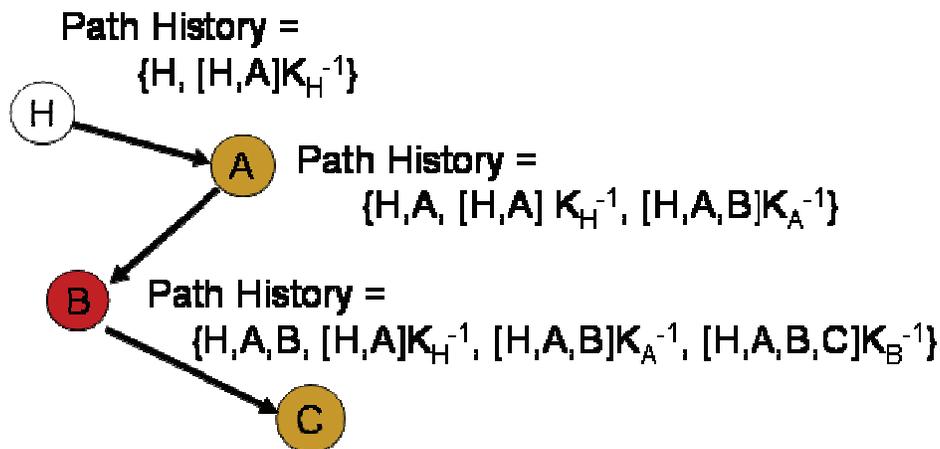


Figure 80: Path Histories

With path histories come some limitations. Namely, the signed path history grows with large itineraries and it is not clear how an executing host would evaluate previous platforms and their trustworthiness. Anonymity and agent privacy are also hard to maintain in this scheme because servers have access to agent history and must be identifiable themselves to other servers. Some applications require host anonymity, such as bidding or auction schemes, where participant identities need secrecy. Like other solutions that involve signature methods, path histories require a PKI to support authentication and non-repudiation.

### A.2.7 Policy Management

Before agents can be allocated resources on a local host, their identity must be authenticated and their authorization level determined. Policy management is a protection scheme, similar to sandboxing, which assumes local policy enforcement is available on the host platform and that agents use embedded policy attributes. Policy-based security management has become a growing research area for mobile agent security as a spill over from work done in network security management [Wright *et al.* 2002].

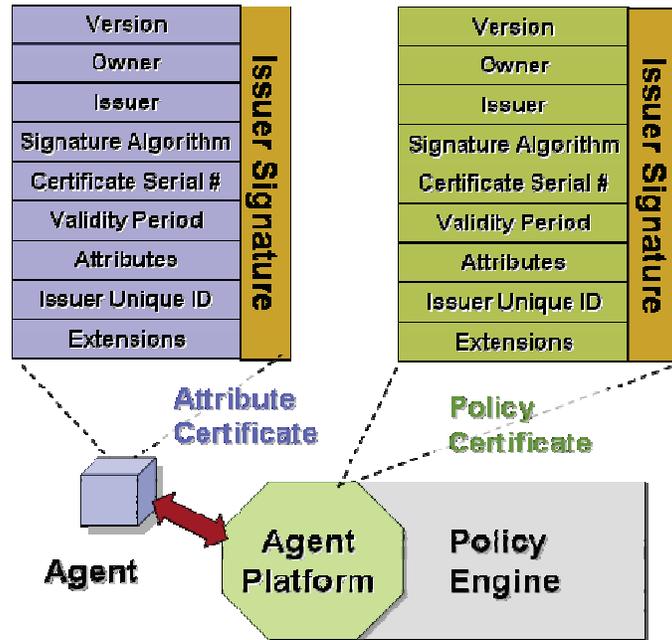
Policy frameworks ultimately protect both agent and host because they concern themselves with expression and development of dynamic trust assessment in mobile contexts. Policies ultimately allow the reigning in of application privileges as well as a limit on the authority of the host itself. Bellavista *et al.* [64] mention several benefits of policy frameworks when used to implement security: reusability, extensibility, verifiability, efficiency, context sensitivity. Efforts have been underway for several years to integrate policy level management into mobile agent environments [79,245,246,247,248,249]. Jansen [2001] formulates a privilege management scheme (depicted in Figure 81) that shifts focus away from countermeasures designed into the internal data structures of an agent [250].

Four weaknesses to embedding authentication or authorization methods in the internal state of the agent include:

- ◆ the number of policy setting principles and trust levels is hard to manage
- ◆ policy expression is limited and not easy to extend when done internally
- ◆ protection means become more limited
- ◆ interoperability is greatly decreased

To overcome these shortfalls, we can embody the prescribed security policies for both the agent and the host externally in separate certificates. The agent governs its use of resources by an *attribute* certificate while the executing host uses a *policy* certificate to govern rules for visiting agent. *Policy* certificates and *attribute* certificates are nearly synonymous except policy

certificates represent more than just a host platform; in some cases, the application system can create and maintain policy certificates offsite from the host. Jansen’s policy framework allows an agent to carry one or more attribute certificates (assigned by the owner or another authority) to hosts in their itinerary, all of which determine the relevancy of a given certificate after verifying an issuer’s identification. The executing host grants privileges to an agent based on whether attribute certificates of the agent comply with policy certificates of the host.



**Figure 81: Agent Policy Management**

Authentication and authorization is in many cases delegated to an agent by various signature schemes. Policy management insures agents do not violate owner intent or do not serve ulterior motives after corruption by malicious hosts. Authentication in distributed systems is well established as a research area [251, 252, 253] and it shares common issues with mobile agents. Several different types of signature schemes for mobile agents have been posed including proxy certificates [254], forward signature schemes [255, 256], strong non-designated proxy signatures [257], multi-signatures [258], undetachable signatures [29, 30, 259], and one-time proxy signatures [260, 261].

Policies should help agents represent their security levels while also giving visibility to the underlying host resources that are accessible to them. Knoll *et al.* [79] use policies to establish trust level of agents based on information contained in their history, namely by assigning trust levels to IP addresses. Once the executing host verifies the integrity of an agent’s path and determines trust, it can choose appropriate security policies. Roth and Jalali-Sohi [262,263] postulate using policy management in a tree-based structure to facilitate cryptographic primitives such as encryption and signature functions. Their quest for a generalized agent security model included an integrating framework with security context, security policies, certificates, and access keys associated with both static and dynamic parts of an agent. Jansen [250] mentions chained authorizations as a corollary to normal certificate delegation—but hosts chain privilege across the multiple hosts in the agent path. Since an agent migrates from host to host, it is difficult to know in many cases the set of visited nodes a priori. In application contexts where agents cross domains and associations, policy negotiation and dynamic trust establishment must take place among principles. Scott *et al.* [248] appeal to policy management as a means to achieve “sentient computing”—a term that describes transparent pervasive network environments that allow users freedom to focus on tasks rather than systems. Their approach relies on modeling

agent interactions with ambient calculus-style primitives so that hosts can incorporate dynamic changes to the agent's environment back into their policies.

**Table 23: Agent Protection Mechanisms**

Section	Mechanism
A.3.1	Contractual Agreements/ Reputation
A.3.2	Detection Objects
A.3.3	Oblivious Hashing
A.3.4	Protective Assertions
A.3.5	Execution tracing
A.3.6	Holographic Proofs
A.3.7	State Transition Verification
A.3.8	Reference States
A.3.9	Environmental Key Generation
A.3.10	Secure Routing
A.3.11	Multi-Hop Trust Model
A.3.12	Returning Home
A.3.13	Phoning Home
A.3.14	Trusted Nodes/Third Parties
A.3.15	Server Replication / Fault Tolerance
A.3.16	Agent Replication / Mutual Itinerary Recording
A.3.17	Route/Itinerary Protection
A.3.18	Sliding Encryption and Decryption
A.3.19	Trusted/Tamperproof Hardware
A.3.20	Function Hiding w/ Encrypted Functions
A.3.21	Function Hiding w/ Coding Theory
A.3.22	Undetachable Signatures
A.3.23	Policy Management
A.4	Data Protection
A.5	Secure Multi-Party Computation]
A.6	Multi-agent Mobile Architectures
A.6	Group Host Protection
5.3.2	Time-Limited Black box/Code Obfuscation

Bellavista *et al.* [64] consider two of the greatest needs in policy management research to be an expressive language for specifying policy and an efficient mechanism to implement policies with less performance overhead. Jansen [250] mentions both Abstract Syntax Notation (ASN) and eXtended Markup Language (XML) as candidates for a policy language, with XML being preferred. We can also use mobile calculi for policy expression—Scott *et al.* [248] for example use Ambients—and mobile agent frameworks themselves (Fargo, Aglets, etc.) may come with their own policy languages. Bradshaw *et al.* [246] point out trustworthy agent systems depend on the ability of hosts to control and adjust agent behavior and the ability of agents to guarantee hosts will follow specified policies.

Framework policies are another proprietary method for security that require infrastructure in both agents and frameworks to implement. Despite this drawback, we might consider frameworks as the only hope of an extensible *generalized* means for combined agent and host defense. We now turn our attention specifically to the role of agent defense mechanisms.

### A.3 General Agent Protection

Defending agents against malicious host attacks is the second major category for considering security mechanisms. Table 23 gives overview of nearly thirty different protection mechanisms found in literature and research.

### **A.3.1 Contractual Agreements/Reputation**

A non-technical means of dealing with malicious host attacks is formation of contractual agreements among host operators [31,129]. Host platforms can make agreements to operate their agent environment in accordance with policies that do not violate the agent's state or data in terms of either privacy or integrity. Enforcement of such agreements remains a societal issue of law with no verification mechanism in place to detect dishonest behavior. Reputation is also a means posed by Rasmusson and Janson [264] to socially identify dishonest servers and prevent agents from migrating to them.

### **A.3.2 Detection Objects**

Being able to verify the correct execution of the mobile agent on a remote host is a coveted goal in mobile agent security. Several methods aim at this by static code analysis or by runtime code analysis; we introduce several more techniques in the following subsections with a similar goal. In this case, the runtime state of an agent is at interest: particularly, has a host modified the stack, variables, execution thread, or instruction counter of a running program to produce unintended results or unintended control flow. Meadows [265] proposed a technique known as detection objects which can discover such irregularities in code execution. By baiting mobile code with dummy data items or functions, an agent owner can detect whether a malicious host has likely altered the execution state of the agent in some way. The assumption albeit is that if no detection objects were modified then the agent was not corrupted. The mechanism assumes that developers can design enough tests to cover the various execution possibilities of the code and that a smart adversary cannot discover which objects are being used for verification.

As a disadvantage, detection objects are not comprehensive in code scope and their use is application-specific (much like proof carrying code and state appraisal techniques). Furthermore, determining the properties of a given set of objects is ad-hoc and the developers may have to change the objects themselves on a routine basis to prevent discovery. Positively, they offer a simple method with small overhead for additional processing and code size that supports rudimentary detection. When used with other supporting techniques, the approach shows great promise.

### **A.3.3 Oblivious Hashing**

Execution tracing is the ability to log or record the runtime behavior of a program given some static code base. Chen *et al.* [266] posed oblivious hashing as a form of software fingerprinting that can be used to perform remote code authentication and provide a level of execution tracing. Oblivious hashing produces a hash of the program's trace and works much like detection objects by adding additional code to the original program. In the hashing approach, the code developer adds additional computations so on-going program execution produces hash values.

Just like detection objects need seamless mixing into the original mobile agent, hashing code requires indistinguishability in order for the technique to remain viable. In the abstract model for this operation, we use a set of instructions and their corresponding memory side effects to produce hashable trace values. Since the trace reflects actual execution, the hash value represents a signature on the behavior of the function. Chen *et al.* implement this trace via code injection that consists of one or more hashing instructions. Figure 82 illustrates the placement of hash instructions in the midst of normal instructions. These commands take the results of previous commands and perform operations on them so that results are stored in separately identifiable memory locations. An actual implementation of their approach was done at the syntax tree level (as opposed to the assembly level) to make hash instructions less obvious.

Local software tamper resistance and remote code authentication remain as possible applications for oblivious hashing. This method parallels other work by Vigna [267] on execution tracing and has immediate usefulness as an agent protection mechanism. The mechanism is adaptable to validate dynamically the behavior of a mobile agent on a remote host by using a form of challenge/response with randomly chosen inputs selected by the sending party or the receiving party. Even though it poses the same disadvantages as detection objects (being

program specific and language dependent), future research could show its usefulness as a two-way verification mechanism against both malicious agents and malicious host attacks.

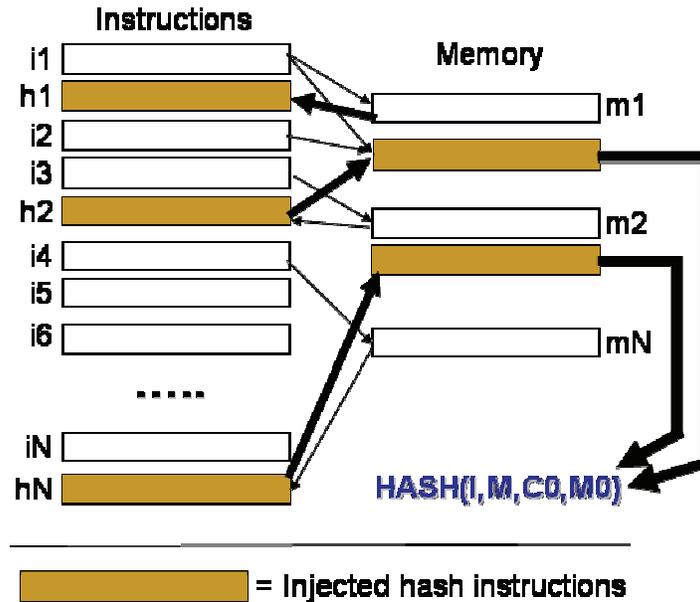


Figure 82: Model for Oblivious Hashing

#### A.3.4 Protective Assertions

As another approach to runtime state evaluation, Kassab and Voas [268] present a methodology similar to state appraisal and detection objects. In the protective assertion approach, we expand traditional notions of run-time assertion checking to encompass tracking of intermediate agent states given to an application owner in the form of snapshots. The owner makes evaluations on whether the snapshots are consistent with expected execution of the mobile program.

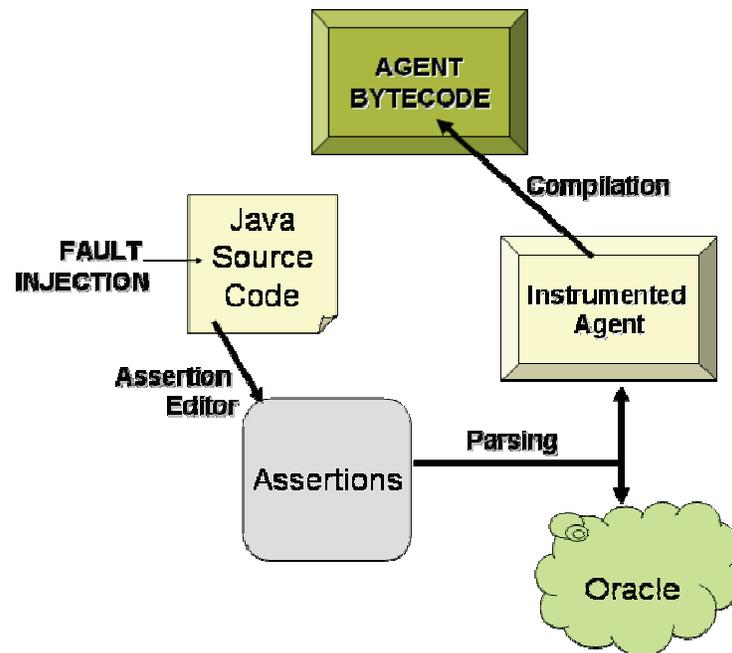
Figure 83 depicts the basic framework of how insertions would be loaded into a mobile agent. In order to produce correct protective assertions, developers must make worst-case predictions on agent code compromise. Developers use these assumptions as part of the fault injection process to identify potential weak agent areas and provide insight into which computations need hardening. We can define assertions by different categories as well, based on pre-conditions, post-conditions, environmental conditions, or invariants to name a few. Once code the developer parses or compiles code into a monitored form, the agent is ready for migration. The originating host evaluates assertions based on the run-time state of the agent during execution or after the agent completes execution.

The assertion method is not comprehensive but does provide a level of detection capability for tampered agents. There is an overhead associated with returning state snapshots to the host for verification and the returned data itself has its own tamperproofing requirement. However, the malicious host would have to anticipate the state check generated by the oracle (seen in Figure 83) in order for the alteration to go undetected. If an agent host bypasses assertions, the absence of state snapshots can pinpoint malicious activity.

#### A.3.5 Execution Tracing

Vigna [267] proposed a cryptographic tracing mechanism that allows servers to defend (or indict) themselves concerning questions of mischief by producing historical evidence of agent execution. It mimics path histories in some ways but instead of keeping itinerary records for subsequent hosts, it examines prior agent execution history. Cryptographic traces extend the ideas of protective assertions and oblivious hashes as another form of fingerprinting. The goal of tracing is to detect illegal changes to the data, code or execution thread of a mobile agent. An

agent owner can check after termination whether the execution log conforms to a correct execution tree. An executing host can send the hashed summary of the log to a trusted third party or back to the application owner itself in order to prove non-repudiability.



**Figure 83: Protective Assertion Framework**

In order to implement execution tracing, applications require several ingredients: hosts must store potentially large numbers of agent history, a PKI must be in place, a method to verify traces after the agent returns home must exist, and a method for time-synchronization among all participants must exist. An executing host provides a hashed trace log if the originating server believes mischief took place. There is also an underlying assumption that the agent owner can gather all of the inputs of each server in the agent's itinerary (unless they are derivable from the agent state). The owner then uses these inputs to run the agent independently to create a new trace. If the hash of the owner's trace is equivalent to the hash provided by the suspect server, the owner verifies execution integrity. Because tracing every instruction in a program can be a burdensome overhead, Vigna distinguishes between white and black code: black code is "tainted" by some interaction with the host environment and is therefore the most important execution to log. Figure 84 illustrates that white statements depend only on the internal state of the agent and such statements remain uninfluenced by inputs of the host external environment. Figure 85 shows an example code fragment and a trace of the execution based on this classification, utilized also by Hohl [63].

Trace logs consist of updates to black statements to minimize their size. As an alternative, the system can use compression by only logging instructions that influence control flow. In practice, execution tracing in a multi-hop environment requires a protocol to transfer or sign log traces for future verification. Vigna suggests the use of a public/private asymmetric key pairs among participants of the mobile agent application for authentication of the logs. Collection in a multi-hop mode could rely on the agent itself and this variation allows a future host in the itinerary to verify the execution of the agent from origination up to that point.

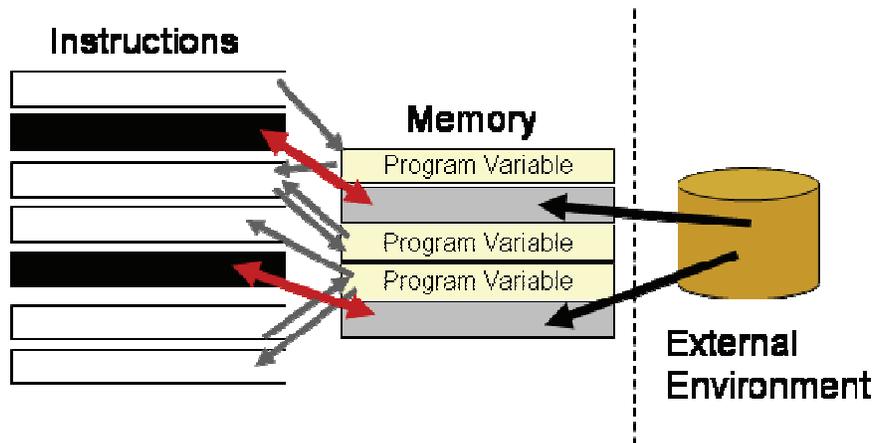


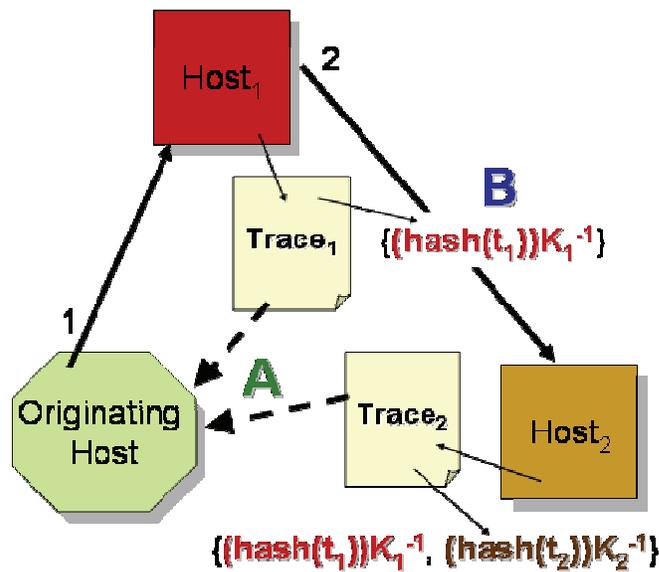
Figure 84: White and Black Code

	Instructions	Trace
7	$x = x + 1$	
8	<code>read(y)</code>	$y = 10$
9	$z = y + x$	
10	$x = x ^ 2$	
11	<code>h = hash(z)</code>	
12	<code>b = getPrice()</code>	$b = 2500.50$
13	$t = b * 0.10$	

Figure 85: Code Fragment and Trace

Figure 86 depicts the exchange of an agent using tracing during its network traversal. The originating host can request traces on demand (method A, Figure 86) or have an appended set of signed hash traces contained in the body (data state) of the agent (method B, Figure 86). If a future honest host in the itinerary wants to verify execution of the agent, other hosts need to be willing to divulge their inputs in order to do so (a scenario not likely in certain application contexts). The drawbacks of proofs include the overhead storage of execution logs and the transmittal of proofs to an owner or trusted third party. Even though some compression methods can make logs smaller by limiting focus to selected ranges of statements (where pricing is determined or a transaction is sealed for example), the mechanism is still only triggered on suspicion and verification is conditioned on server agreement for disclosure.

Tan and Moreau [33] introduce an extension to tracing that somewhat mitigates these disadvantages and solidifies the ad-hoc nature of the trace verification process. They introduce a trusted third party, which serves the role of verification authority for traces generated by an agent server. This solution reduces the individual overhead of trace log storage and induces the detection of denial of service attacks when combined with a time-stamping service (to support time-out determination) among the hosts. Figure 87 depicts the protocol exchange involved in this extended environment. In extended tracing, one or more verification servers ( $V_A$  and  $V_B$  in Figure 87) provide cooperative agent migration among hosts in the itinerary. The verification server receives both the agent body (code and state) and the execution trace from the host after it accomplishes its previous execution. The executing host signs all protocol interactions such as request for the agent, receipt of the agent, and receipt of the agent trace ( $m4$ ,  $m6$ ,  $m8$  in Figure 87) to ensure non-repudiability. EET performs tracing as part of the agent's migration through the network and thus provide immediate tamper detection.



*Method A: Upon Request*

*Method B: Carried By Agent*

Figure 86: Execution Tracing Model

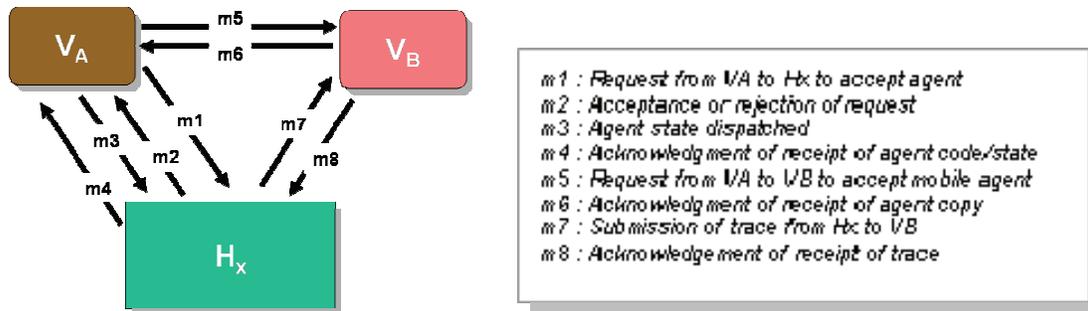


Figure 87: Extended Execution Tracing

When used in conjunction with time synchronization, this approach also prevents denial of service where an executing host unduly detains an agent or never releases an agent to its next hop. The tracing process reflects a limited detection capability (which is application dependent and assumes an adversary cannot create a valid trace to an altered program run). Traces only indicate if a given result is a *possible* execution of the program and not necessarily the *actual* execution of the program. Trusted third parties like the verification server decrease the openness of this solution however in certain mobile agent environments.

### A.3.6 Holographic Proofs

Yee [31] offers several mechanisms for agent protection, one of which is a cryptographically based integrity checking method known as holographic proofs. In the Vigna [267] model, the application owner needs to compare hashes or full traces of a program execution in order to verify agent computation integrity—but transmission of large traces remains inefficient. Holographic proofs improve efficiency of execution tracing for proof verification by reducing

overhead of these comparisons. Figure 88 highlights the sense of the holographic proof exchange and shows that a predicate  $p(x,y)$  is equal to 0 when it represents the verified execution of a program  $x$  (the mobile agent) that has generated an execution trace  $y$  (the trace from running on a remote host). The size of  $y$  (the trace) is large but it can be encoded as holographic proof  $y'$  having the property that only a few bits need to be examined to verify the execution. However, the size of  $y'$  is also large (in many cases larger than  $y$ ) and thus prohibitive to send as well—so another approach must be chosen.

If the remote server can discern which bits of the entire trace belong to the holographic proof, it can subsequently invalidate the results of the proof itself. Figure 88 illustrates how the originating host can query the bits of the holographic proof  $y'$  without allowing the remote host to know which bits are being evaluated by means of private information retrieval (PIR) techniques, expounded further by Biehl *et al.* [269] and Gertner *et al.* [270], while Loureiro *et al.* [230] mention it as a solution to the size problem. The overall result is that large proofs of integrity remain on the execution platform (remote host) while execution tracing itself becomes more efficient.

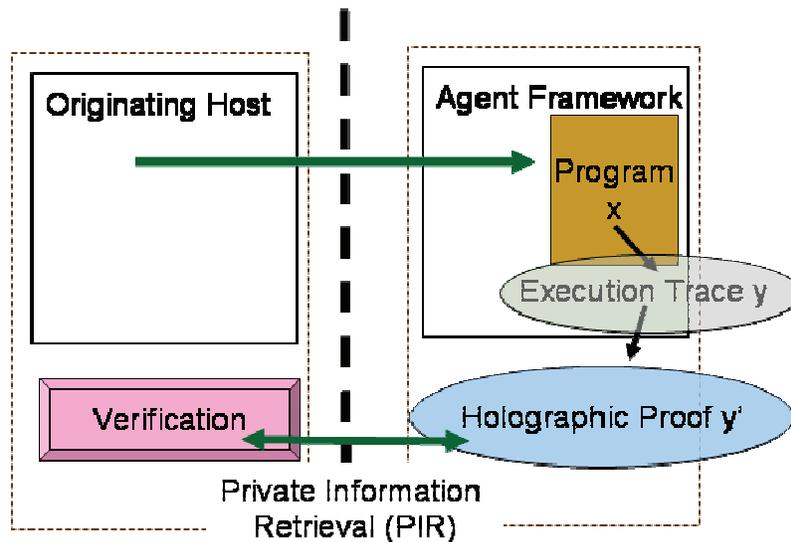


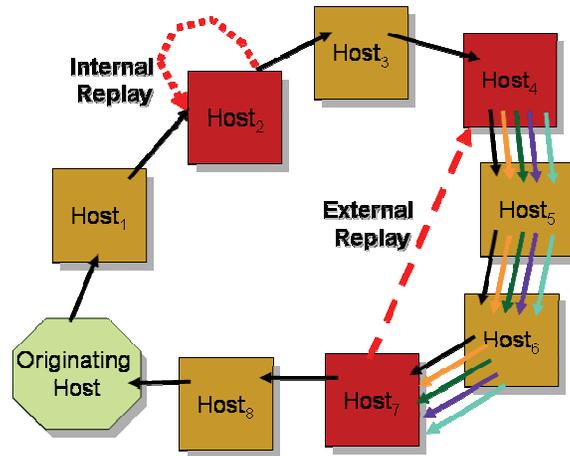
Figure 88: Holographic Proof Checking

### A.3.7 State Transition Verification

Yee [74] did further work on agent security with a proposed scheme to detect replay attacks against migrating agents. Replays are essentially when an individual server executes an agent repeatedly in order to understand and undermine the semantics of the program. If a server can game the input of the agent to produce a desired output, it has essentially cheated. We refer to this form of attack as internal replay. We can also define external replays as a cycle in the itinerary of an agent induced by cooperating malicious nodes that send agents back to each other over and over again. Figure 89 illustrates both types of attacks and depicts an agent migration path through a network that includes three malicious hosts, two of which are cooperating.

Figure 89 illustrates the agent interaction model used by Yee to define properties of monotonic operations. An agent computation can be seen conceptually as a set of functions  $q_i$  used to query resources  $R_i$  belonging to a server  $S_i$ . The previous dynamic data state of the agent ( $y_{i-1,0}$ ) provides an initial starting point for the server. New data states result when the agent issues a query via its static code to the server, creating a sequence of intermediate data states ( $y_{y,i}$ ) that are a function of the previous state and the output of a given query ( $x_{i,j}$ ). The executing host packages the final data state with the agent and sends it forward to the next host. For internal replays, the ability to monitor from the outside the sequence of individual state transitions is required. Yee [74] describes *monotonicity* as the enforcement of one-way state transitions of the executable state of a program in time. If developers can analyze code to indicate which program

statements create such transitions, then application principles can monitor state transitions to identify and detect replay attacks. Yee develops the thought behind such one-way transitions and asserts that agents cannot trust the state they are carrying and would need to rely on verification services by an outside party.



**Figure 89: Replay Attacks**

Internal replay attacks are the hardest to prevent even when state transitions can be adequately determined. The reasoning is that unless obfuscation or program encryption is used, an adversary can eventually determine what part of the code communicates with an outside verifier of the internal state transition. A malicious server may not need to understand all of the agent internals either. It only needs to be able to predict some desirability factor of making a given change to an input to be successful. External agent replay attacks are easier to detect because intermediate hosts can help check state transition values (carried with the agent or queried from a trusted third party). A similar problem is the detection of agent clones, which is addressed further by Baek *et al.* [271], Lam and Wei [272], and Roth [109].

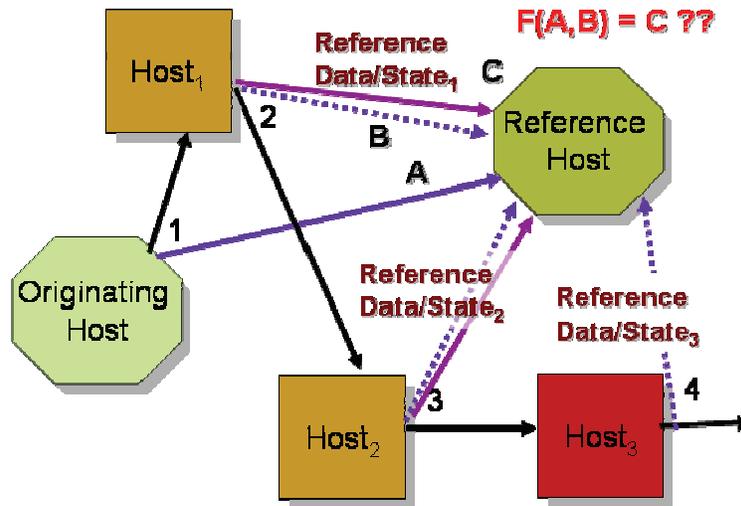
As a drawback, the mechanism attempts to model explicitly the extremely large statespace of a program for correct determination of one-way state transitions. Therefore, developers look at only certain state transitions, just as holographic proofs only look at certain black/white statements [31]. State transition also depends on trusted hardware or a trusted third party to assist in monitoring transition information associated with a mobile program.

### A.3.8 Reference States

Another way to perform state analysis of a mobile agent does not involve tracing but rather full comparison of an agent to a known good execution. Hohl [63] develops such an approach to detect state modifications based on comparisons to a known good model. In this method, the application owner generates a baseline agent state on a trusted reference platform and compares the execution state to those created by executing hosts in the real environment. Figure 90 shows the interaction of an agent execution based on a reference state monitor and we explain the protocol next. Assuming that a developer or application owner digitally signs the constant values within an agent to detect modification, the application owner can analyze the non-changing values of interest in during agent execution. An untrusted host and a trusted reference version provide the comparison opportunity, as long as all the host input is available for the reference program. The method uses a replicated server for parallel agent processing during its itinerary traversal. A reference state by definition is the variable part of a mobile agent executed by a host with similar input (thus displaying reference behavior). Hohl [63] analyzed the core operations of three protection mechanisms (state appraisal, execution tracing, and server replication) to formulate a generalized model for reference behavior.

The application owner checks reference states either when the agent returns home or after each intermediate host execution in the itinerary. As seen in Figure 90, before migration to Host<sub>1</sub> (1) an agent sends its initial state (A) to the reference host. Assuming the *moment of checking* is

per-server execution, Host<sub>1</sub> sends both its final agent state (C) and its reference data (B) to the verification host. Since the next state of an agent is a function of the previous state (A) and any reference input (B), the reference host can now compare the computed next state  $F(A,B)$  with the received next state of the host (C). The reference data (which hosts must provide to mimic the execution) varies according to verification time during the itinerary.



**Figure 90: Reference State Mechanism**

In terms of limitations, reference states cannot detect actions that do not result in a modified agent state, such as a confidentiality attack where a malicious host exposes private information. It can also not detect when the input and output on the remote server has been modified or is not supplied properly. Variations exist to strengthen the approach which include being able to choose an arbitrary checking algorithm (like execution tracing, oblivious hashing, etc.), using less than all possible reference data, and having a subsequent host be empowered to perform the reference check itself. This scheme of course induces a proprietary overhead for a trusted third party to be a reference monitor and a lack of host data privacy, but the method itself does not require modifications to the code or creation of proofs.

### A.3.9 Environmental Key Generation

When an agent must carry private information or sensitive information that can only be used to conduct certain transactions (like signing a transaction), it is sometimes best to keep the agent naïve about the information it carries until the appropriate time arises. To prevent an adversary from determining when the agent will expose its key for such purposes, Riordan and Schneier [273] posed a solution that keeps the agent “clueless” about what it is looking for. A clueless agent performs an appropriate decryption operation only when the agent executes on a host meeting specific environmental conditions; such conditions arise when the executing host executes the agent static code. The agent keeps code in an encrypted form until needed, then dynamically decrypts it after the executing host meets the appropriate environmental condition. Figure 91 depicts the general operation of the environmental key generation process.

This scheme utilizes a methodology similar to how UNIX-like operating systems store the hash value of a password and compare that to the hash of passwords entered by a user at login. Riordan and Schneier [273] give several methods for environmental information that agents may respond to. A simple example involves a scenario where the code unlocks a key based on the cryptographic hash of a piece of information the agent is interested in. Figure 91 illustrates, for example, the string representation for a stock item of interest from a buy/sell transaction. The hash of this key is  $h(h(I))$ , with  $I$  being the item of interest; the agent code contains this hash. A malicious host wishing to subvert the transactional capability of the agent in some way cannot determine from  $K$  or  $X$  what exactly the agent is looking for. If it could, it may offer a false bid or

query result in the hopes that the agent divulges a private signature key or reveals private information. The agent can thus be decrypted only when the executing host meets the right environmental conditions. A malicious host still has the opportunity to subvert the agent by changing the execution of the agent to its advantage when it does meet a matching condition. Another drawback is that certain agent frameworks may not allow dynamically generated code execution, thus limiting the approach.

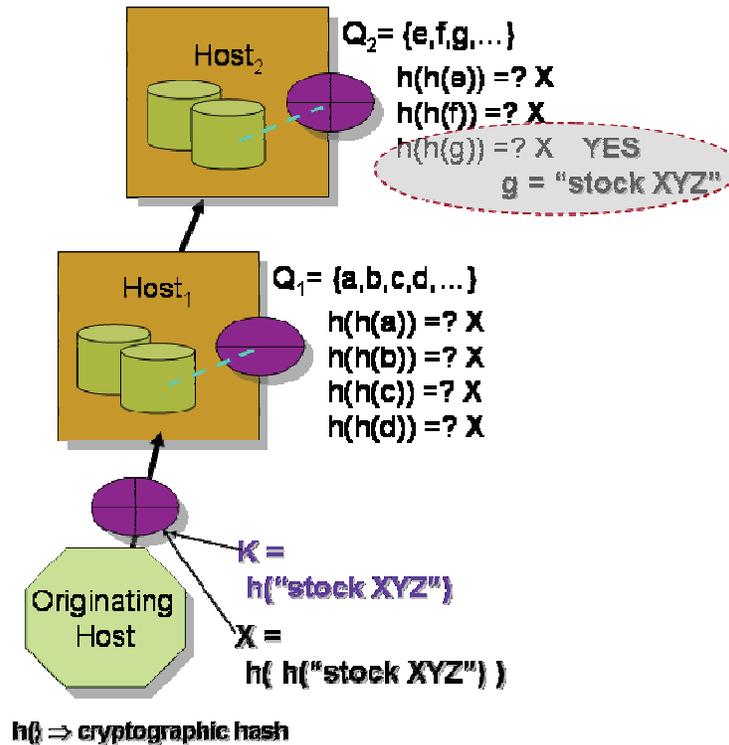


Figure 91: Environmental Key using Hash

### A.3.10 Secure Routing

Secure routing assumes trusted hosts will never perform malicious activity and will send agents only to hosts that have credentials with a certificate authority (CA). Farmer *et al.* [23], Swarup [1997], and Knoll *et al.* [79] elaborate the mechanism usage which involves restricting the itinerary of an agent via routing policies. Figure 92 illustrates how an agent may only visit hosts that have an association such as IP address registration with a CA. Servers can also create policies where they only accept agents with established security associations—much like executing host can verify path histories to determine probabilities of malicious alteration.

Secure routing involves host infrastructures operated by a single party [106] or secure networks induced by the presence of trusted hardware, discussed shortly. We do not control servers in competitive environments or unknown security relationships in such a manner and agents do not consider them trusted as a result. Even in a military scenario where friendly hosts could be considered “trusted”, a compromised or captured node may or may not exhibit malicious behavior while still being considered trusted. Because of these issues, we consider secure routing best suited for protection in applications where mobile agents have a priori knowledge of the executing host environment.

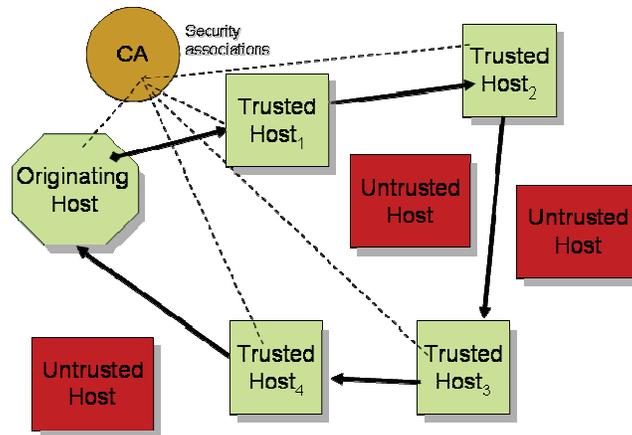


Figure 92: Routing Based on Associations

### A.3.11 Multi-Hop Trust Model

Thorn [70] describes agent security policies based on the worst-case access privileges of all prior visited hosts in the itinerary. Assuming that all hosts are at worst case malicious, this model designs agent access control to be a decreasing set of privileges based on the itinerary. At each hop, the agent merges the prior access control list (ACL) with current ACL so that security privileges can only remain the same or decrease. Figure 93 shows a notional view of this with a decreasing privilege list for the agent based on specific actions disallowed at a given host. Here, an agent encounters a host with a limited policy that disallows remote procedure calls (RPC). As the agent migrates, the subsequent executing hosts disallow access to a particular file as it carries with it the previous restriction against RPC.

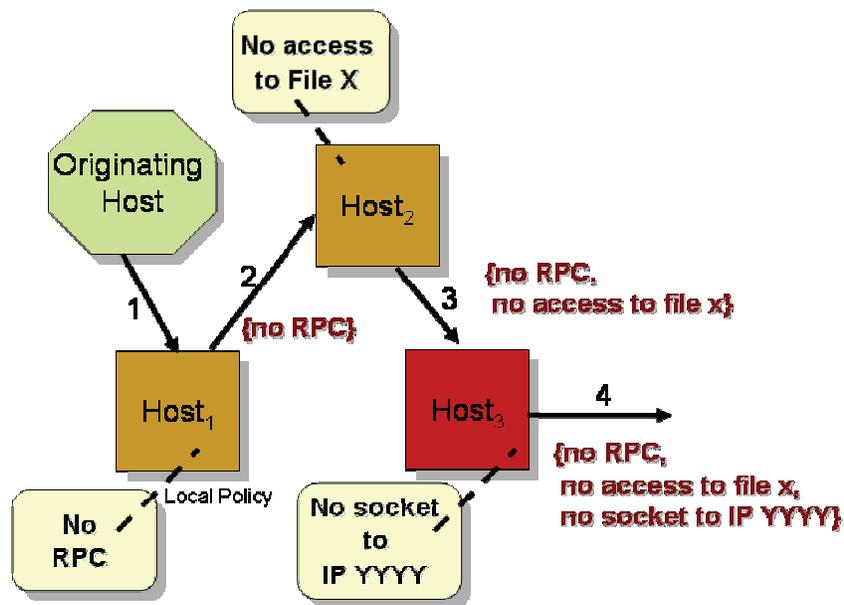
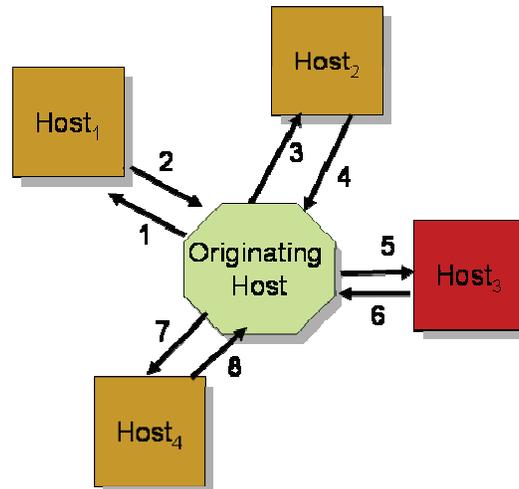


Figure 93: Multi-Hop Trust Model

### A.3.12 Returning Home

Some mobile agent architectures require the mobile agent to return (migrate) to its home platform after a single hop, which is described as the Jumping Beans model by Jansen [250]. Figure 94 illustrates the star shaped itinerary that centers on the originating host being available

during the lifetime of the agent. Single-hop and two-hop boomerang migrations in this configuration allow for the strongest level of trust association and validation of the agent code and state. The application owner verifies each subsequent hop first hand and prevents or detects malicious activity incrementally. This approach involves migration of code and state versus sending of static messages—thus requiring code within the agent to account for return trips home and processing of information embedded within its state. Executing hosts can pass static messages along to services or static agents without requiring additional change in logic. As a drawback, there is increased communication overhead and a requirement for the host platform to remain available during agent migration.



**Figure 94: Agent Returning Home**

### A.3.13 Phoning Home

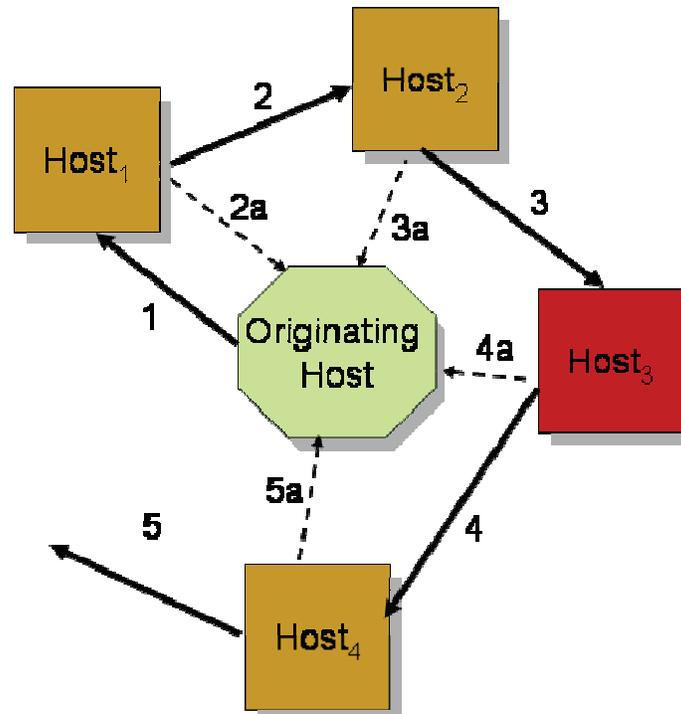
Grimley and Monroe [274] describe a countermeasure involving status updates and data sent home by the migrating agent. In this mechanism, an agent sends a message to its originating host either on arrival or migration from a remote server indicating that it is alive. When done in conjunction with a time-based measurement, this scheme can be used to help avoid denial of service attacks or to detect them being executed by a particular host. Several variations are possible in this approach. Agents can send static messages (via KQML, KIF, or another form of ACL) that include the host data result. If the application owner does not need to analyze data until after the agent returns home, the agent does not have to carry the results any further. Figure 95 illustrates that for every migration of the agent (2, 3, 4 ...) there is also a corresponding static message (2a, 3a, 4a...) in conjunction with it. If the information is required as part of the computational state of the agent, then at minimum the originating host has an unaltered copy of each host's data result.

We see other variations in several defense mechanisms (execution tracing, protective assertions, holographic proofs, and reference states). The methodology does not totally avoid host tampering but phoning home can eliminate the exposure of intermediate results to other platforms and provide detection capabilities for the agent owner. In lieu of sending back collected data, the agent can also maintain results in the data state but must use some type of protection mechanisms to ensure the integrity and confidentiality of the accumulated data.

### A.3.14 Trusted Nodes/Third Parties

A variation to phoning or returning home is to make use of trusted third parties. Wilhelm *et al.* [138] define trust in this respect as belief that another party will abide by a published security policy. Both parties (agent originators and agent executors) can create a trusted third party relationship by introducing tamperproof hardware. Certain models place trust in specific entities implicitly, like a base station in a mobile ad-hoc network, and always assume trusted operations.

As an alternative to requiring the originating host to be online during the course of an agent's execution lifecycle (which severely limits disconnected operations), Figure 96 illustrates how we can introduce one or more trusted third parties (TTP) to facilitate agent verification between host migrations.



**Figure 95: Agent Phoning Home**

Using TTPs reduce the bottleneck or single point of failure from reliance upon the originating host and allows implementation flexibility for other security mechanisms. Extended execution tracing, for example, extends normal tracing with a TTP to prevent denial of service attacks and perform automatic execution log verification. Agents can also use a TTP as a location where secure transactions can take place, such as signature using a private key embedded in the agent. The TTP can take advantage of stronger security associations with an agent owner to support sensitive operations that might lead to leakage of confidential information on a malicious host. We can use TTPs to migrate agents securely, store sensitive agent information (private keys, private originating host information, or private analysis results) and perform secure operations. Proxies can also reduce malicious agent attacks by reassuring host operators that agents have passed certain verification tests or proofs of integrity before dispatching them.

Ng and Cheung [275, 276] use a trusted party to individually negotiate the security requirements and trust level between an agent and a host. Figure 97 depicts the interaction of an agent with a TTP to establish trust relationships with all hosts that are in the itinerary. Trust level exchange occurs in the sense that an agent owner and an agent server both delegate trust verification services to the TTP—using it to establish authentication and trust levels upon agent dispatch. In the first step of a multi-hop traversal, the TTP negotiates authentication information with a particular host (Figure 97-2a). Next (Figure 97-2b), both the receiving execution host and originating host communicate trust levels embedded in the agent.

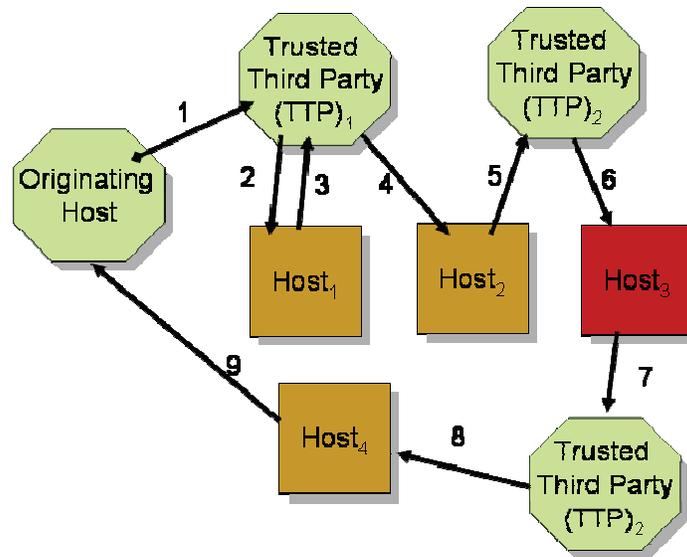


Figure 96: Agent Protection Using TTP

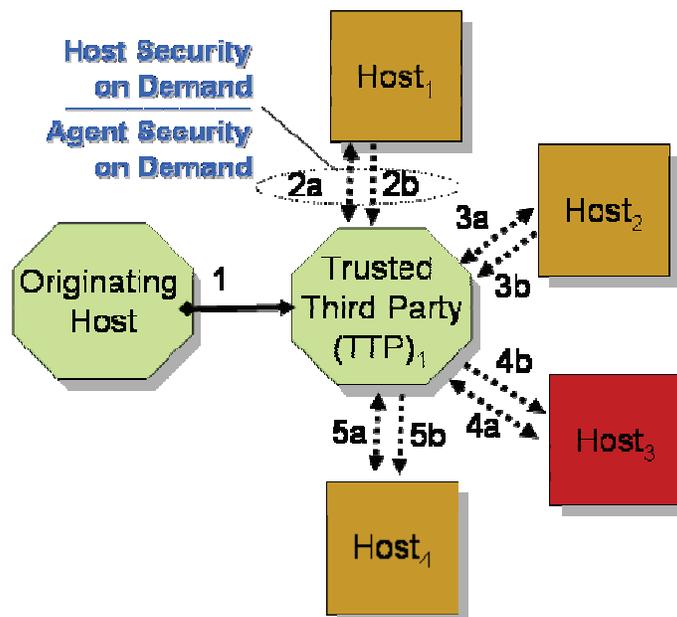
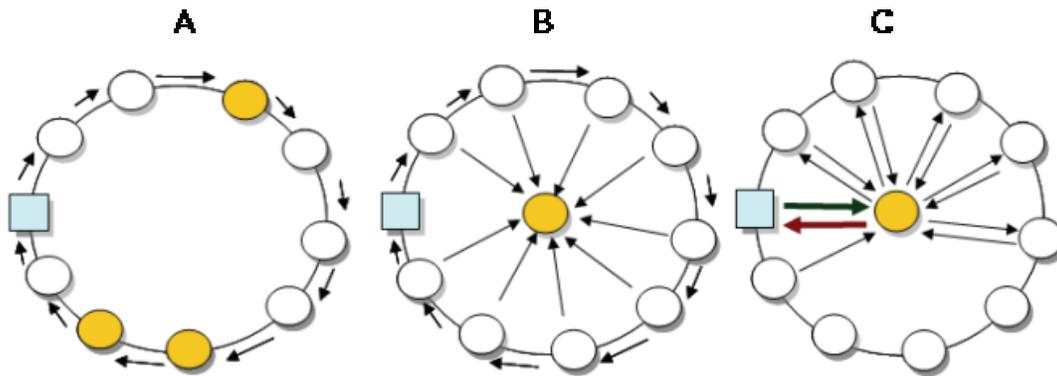


Figure 97: Trust-Level Exchanging Protocol

Once the TTP accomplishes these steps for every host in the agent's itinerary, the TTP can perform various security countermeasures on behalf of both host and agent, including guarantee of anonymity. For the host, the TTP may generate a proof of the mobile agent or certify the agent using a trust token of some kind. For the agent, the TTP can apply an obfuscation algorithm, introduce noisy code, or encrypt parts of the program. Trust level exchanging illustrates both host and agent security on demand negotiated by a party that both sides trust. The TTP also performs agent migration as in extended execution tracing. The approach provides flexibility by allowing hosts and agents to specify security countermeasures on demand, a coveted facet of policy negotiation. The drawback is the bottleneck and single point of failure for the TTP, though we can mitigate risk by shared or multiple TTPs. Protocol exchanges further incur additional communication overhead for every migration. If the host and TTP do not trust each other, the scheme is not practical at all.

The presence of tamperproof hardware in most cases elevates trust to acceptable levels on intermediaries or execution hosts in the mobile environment). Zachary and Brooks [2003] point out that trusted intermediaries can perform functions to reduce risk of agent tampering such as storing secret information in between host executions, maintaining non-repudiatable logs of host activity, securely transmitting agent code to hosts, and providing authenticated control of host access to mobile code. These schemes use some version of trusted hardware in the role of a middleman—but they do not provide true privacy of computation unless prospective executing hosts use trusted hardware.



**Figure 98: Trusted Host Configurations**

Figure 98 summarizes common intermediate hosts configurations: relying on trusted hosts in the path to perform verification on behalf of an agent (Figure 98-A), sending partial results or hash values of host computations to a trusted repository for later verification (Figure 98-B), or relying on a trusted computing base to verify each step of an agent's life cycle (Figure 98-C). We refer to a final category of trusted third parties to as *honest* nodes, which are not necessarily *trusted* but are simply *not malicious*. Trusted third parties typically support specific security purposes whereas honest nodes execute security-conscience agent functions. We consider honest nodes friendly in the sense they help to verify agent security during transit just by virtue of executing the agent correctly. An honest host for example can use publicly verifiable detection chains to discover tampering of intermediate results gathered and collected by the agent. The public property allows the application owner to use honest nodes in the path without having to rely on specifically designed trusted third parties, greatly reducing the proprietary nature of candidate security solutions. Execution tracing, reference states, and route protection mechanisms all rely on the friendliness of honest nodes to execute algorithms that detect malicious behavior.

### A.3.15 Server Replication/Fault Tolerance

Minsky *et al.* [277] developed early thought in fault-tolerant agent based computing. One remedy to malicious host corruption of the computation is to duplicate the servers an agent might visit and send multiple copies of the same agent to them. In terms of a simple agent computation that traverses a network visiting several hosts in an itinerary, a pipelined view of the system would see each host as a stage in the process and each migration as a step in the pipeline (labeled as stages in Figure 99). The application owner achieves the final answer when the agent processes on the last host, or last stage of the pipeline known as the actuator. Minsky *et al.* note that hosts communicate any malicious activity introduced into the pipeline at an early stage subsequently to all other pipeline stages as a result. Correct computations are therefore dependent on propagating results of error-free runs in the pipeline.

From a fault-tolerant viewpoint, if we could duplicate a given server's individual processing, we could attain a more resilient agent computation. This assumes each stage is deterministic; however, hosts perform computations in subsequent stages with unknown created values. If this assumption holds, we can replicate each step of the process (each agent server) and take a vote at each stage of the computation by that particular group of servers. The majority vote among servers for that stage will decide which outcome of the computation will proceed to the next

stage. Figure 100 depicts two stages of an agent's journey across three replicated servers. The servers for stage 1 all have similar functionality; the originating host duplicates the original agent and dispatches it to each one individually. After servers in the first group have completed processing, the servers take a vote to see which state they will transmit to stage 2. This carries forward until the end of the agent's lifetime.

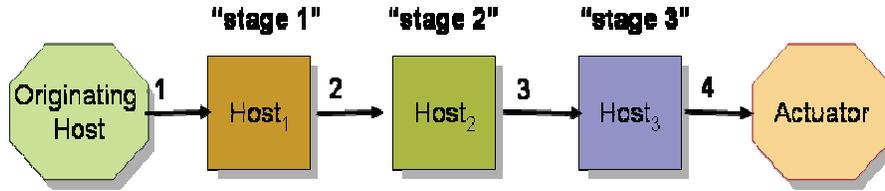


Figure 99: Simple Agent Pipeline

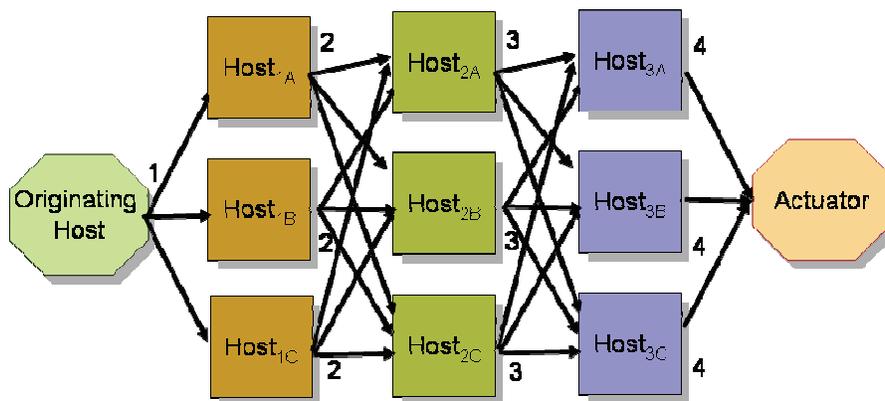


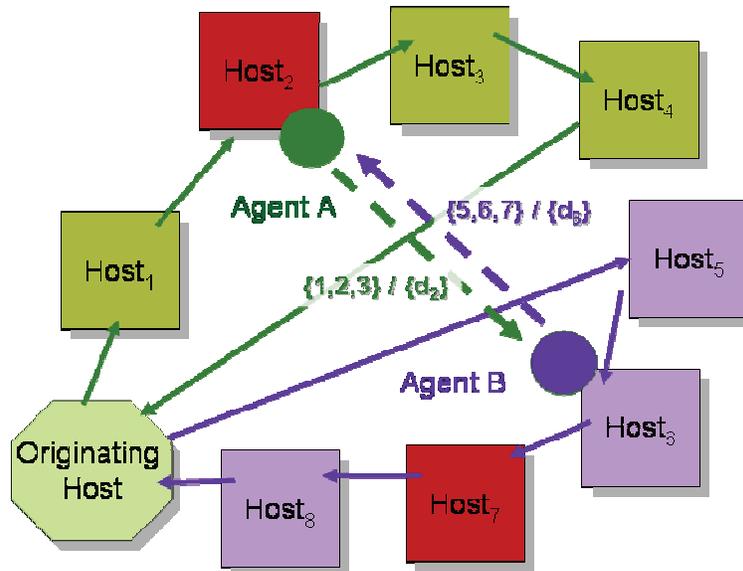
Figure 100: Replicated Agent Pipeline

Minsky and colleagues conclude the simple notion depicted in Figure 100 does not tolerate one malicious node at each stage of the pipeline. We need to apply cryptographic techniques in the replication scheme by equipping agents with a secret carried throughout their itinerary. The methodology considers the use of the  $n$ -of- $k$  secret threshold scheme proposed by Shamir [278] to avoid misuse of the secret or destruction of the secret from preventing proper completion. Resplitting of the secret share is actually required to make the scheme secure. Two proposed protocols may mitigate the exponential message sizes involved with distributed secret sharing and destruction of secret shares by one or more colluding hosts. Voting comes with a cost and replicated servers and agents for each stage of an agent's itinerary may not be realistic. Other work by Pears *et al.* [279] illustrates how a malicious party could simply replicate their stage (server) several times. As such, we do not expect replication to work well in competitive environments but we may adapt it to work with other security mechanisms in a controlled host deployment context.

### A.3.16 Agent Replication/Mutual Itinerary Recording

Replicating servers and agents has some parallel applications to fault tolerant mechanisms. Roth [280] for example proposed a protocol that partitions hosts into disjoint sets and tries to decrease the likelihood of collusion across those sets. Two agents in this mechanism share the authorization method for a transaction between them and verify the activity of the other agent. This protocol (known as mutual itinerary recording) is a variation of path histories and relies on itinerary information exchanged between two cooperating agents. However, this approach can never prevent two colluding malicious hosts from being in the same set of visited hosts and requires a secure communication channel between the two agents. Figure 101 depicts the cooperating mutual agent approach and highlights the fact that agent communication of partial results and itinerary takes place as each agent traverses its itinerary. The itinerary information

contains the previous hosts, current host, and the next host in the agent's path, which forms a non-repudiatable chained log. For example, agent A in Figure 101 would communicate to agent B the previous hosts (1), the current host (2), and the next host to be visited (3). The figure also depicts agent A communicating to agent B its data result ( $d_2$ ) obtained at host 2.



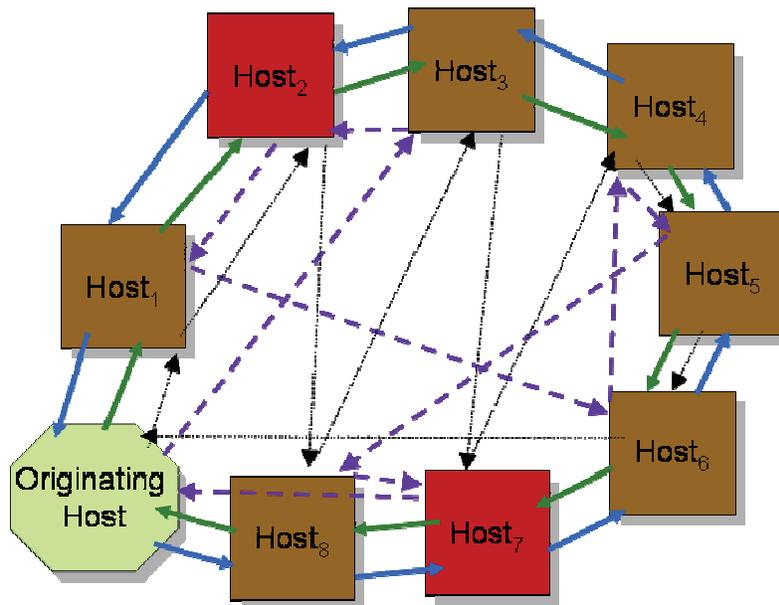
**Figure 101: Cooperating Agents**

The drawback to cooperating agents is that if one agent dies, the other agent may not be able to finish its task properly. The size of the agent state grows linearly with the number of servers visited by each agent, though other mechanisms also suffer from this overhead where the agent carries intermediate results. An adversary can foil the scheme if the server partitions happen to include cooperating malicious nodes working together or multiple colluding agents on the same partition. Yee [31] proposed a two-agent scheme that used replicated servers and noted that replication alone does not mitigate the individual “brainwashing” on an agent. However, replicating agents and using simple cryptographic communication defense can solve certain simple problems, such as limited applications that have *at most one* malicious host in the itinerary. Figure 102 depicts the use of multiple agents with different itinerary orderings—all visiting the same set of servers.

We can program a set of multiple agents that are identical in task to also transit the *same itinerary*, but in *different orderings*. This scheme may help indicate when malicious activity has occurred in the routes of any particular agent if the application owner compares results. This is another form of replication where we use voting and analysis of returning agents to pinpoint the presence of a malicious host. Agents can visit subsets of servers in multiple different set arrangements. Though the mechanism is not preventative, it supports a relatively simple method of detection and fault tolerance against multiple colluding hosts.

### A.3.17 Route/Itinerary Protection

Protection of the itinerary, which determines the agent's migration route in a network of servers, is of particular interest for security mechanisms. The use of a trusted environment on each host server provides a family of solutions in this regard. Wilhelm and Staamann [76], for example, describe the use of the CryPO protocol for keeping the itinerary of a mobile agent secure and private. They postulate that no possibility exists to enforce policy rules without relying on a tamperproof hardware environment (though several results prove this assertion false in limited contexts). In essence, because the executing host only decrypts and executes code in a sealed environment, the mechanism for manipulating the itinerary is out of reach to a malicious party.



**Figure 102: Agents w/ Variable Itineraries**

When we cannot rely upon trusted third parties or tamperproof environments in a mobile agent environment, we must employ alternative mechanisms to protect agent data (including the itinerary) from modification and disclosure. Several schemes in the literature provide for itinerary protection and each scheme has varying degrees of secrecy and integrity. Westhoff *et al.* [244] introduce a method to keep the itinerary of a free-roaming agent unaltered and confidential. Their protocol uses a partial encryption technique based loosely on the onion routing scheme of Reed *et al.* [281] that supports anonymous exchanges. The mechanism composes the itinerary of an agent so that any host in the itinerary can at most know the previous and the next host. The protocol requires public key operations to encrypt the addresses used by each host with the signature of the originating agent owner. In this instance, only the originating host knows the full route. Figure 103 depicts the notional exchange for such a procedure that uses anonymous routing.

As part of the protocol, the agent carries with it signed and encrypted messages that contain the previous, current, and next host in the itinerary. In Figure 103, host H1, H2, and H3 each verify the signature on their particular piece of itinerary information found embedded in the agent and then use their decryption key to determine which host to send the agent to next. The next honest host is able to determine malicious host alterations of the itinerary because their particular triplet would not match the sender and their own identity. In terms of limitations, the scheme does assume a fixed agent itinerary and uses public key cryptography, which is less efficient and computationally more costly than shared key methods. Westhoff *et al.* [244] provide an extended mechanism where an executing host can dynamically add an itinerary fragment, provided it supplies signed routing information like the original host does.

Knoll *et al.* [79] devise route protection via a method depicted in Figure 104. Their method for providing itinerary protection for the NOMADS agent system resembles path histories but ensures the history and future destinations of an agent form a verifiable chain of IP addresses. In essence, this protocol verifies that the previous host added its identity appropriately, but suffers from the inability to detect truncation when colluding malicious hosts work in tandem. This lightweight protection method does not provide cryptographic security, but rather allows each host

to append path information directly. This mechanism supports free-roaming agent scenarios by allowing non-static itineraries.

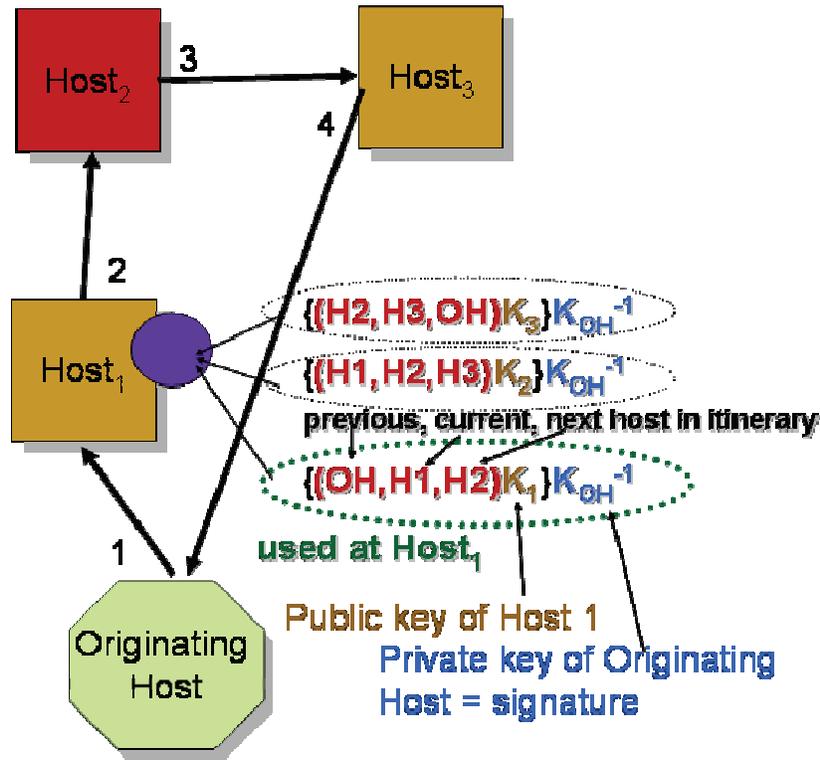


Figure 103: Anonymous Itinerary

As Figure 104 indicates, each time an agent migrates the current host appends a copy of its path information (a collection of IP addresses and hostnames) to the agent's data state. On arrival at the next platform, the receiving host can verify the IP address from the communications channel itself and verify that the sending host appended its correct information to the path information set. The use of a TTP improves reliability and integrity of the chain itself. By keeping the path unencrypted in the original form of the protocol, honest nodes can perform a path history evaluation of the agent and apply a corresponding security policy.

Wang and Pang [282] develop an algorithm for parallel agent deployment that provides protection for the agent path in context to Internet e-commerce applications. In this setting, the agent reveals minimal path information to an executing host. As such, an application owner dispatches a large number of agents to accomplish the same task, two at a time, in a binary tree format so that migration information available to a host is never more than the right child agent. The protocol prevents, for example, agent dispatch to a wrong host by linking the decryption of the route to the dispatch tree. An honest host would eventually detect a route alteration and send the agent back to its originating host. The binary dispatch model proposed points to an area of research focused on finding optimal agent deployment strategies. In this case, the dispatch model also provides security characteristics that protect the agent path.

Vijil and Iyer [283] address the issue of co-operating malicious hosts and pose an algorithmic approach to detecting their activity. They extend the append-only container developed by Karnik *et al.* [284] in the Ajanta mobile agent system and develop a container that uses a cryptographic checksum applied every time a new entry is provided by a remote host. Each host signs its own data element (which is an itinerary addendum) and then encrypts the signed item, identify of the remote host, and current checksum with the public key of the originating owner. The protocol can

detect collusions in both static and dynamic itineraries while being able to solve the cut-paste attack problems described by Roth [108,112]. As an added benefit, the protocol also pinpoints malicious activity and indicates which host is responsible.

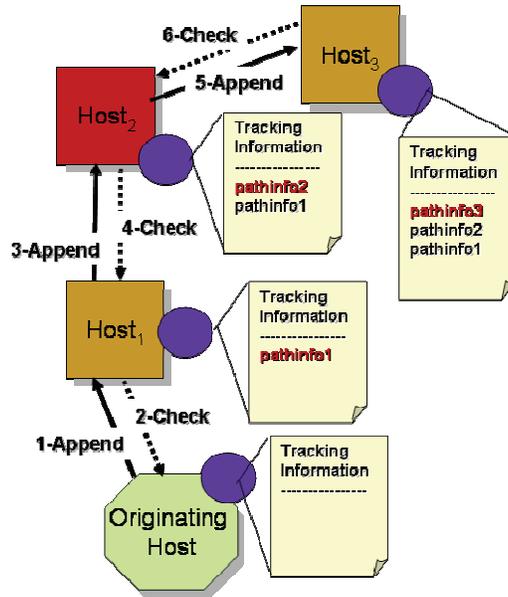


Figure 104: Chained IP Protocol

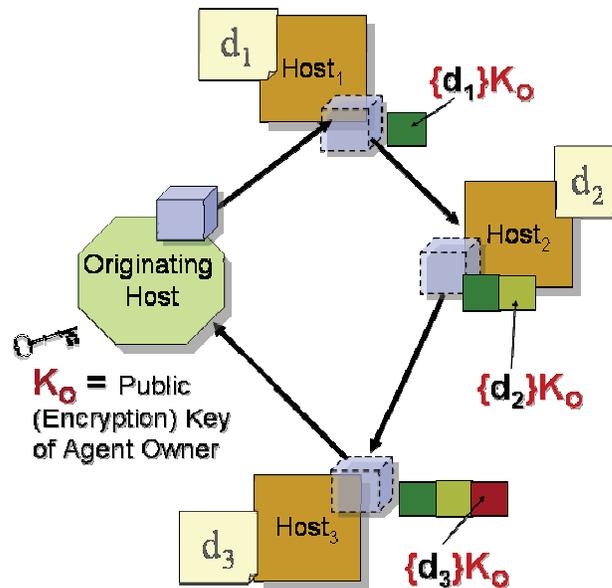


Figure 105: Public Key Data Encryption

Various topically related works exist in the literature concerning agent itinerary protection. Borrell *et al.* [77] offer a partial solution to protecting the itinerary by forcing a level of non-repudiation among host entries. In terms of efficient and secure route protection strategies, Domingo-Ferrer [285] introduces a hash-based technique used in two separate protocols. The

first protocol uses hash collisions and looks to minimize computational costs of the remote agent platform. The second protocol uses Merkle trees and looks to minimize cost of route protection by the agent owner. Satoh [82] presents a general approach for selecting optimal and secure mobile agents in a multi-agent setting based on their prospective itineraries. His contribution includes a formalized method to define the itinerary using a specification language, a step noticeably missing in many solutions.

We view route protection as a subset of the more involved agent defense requirement to protect the free-roaming data state of an agent. Though some agent frameworks or standards may treat the route separately from the computational state of the agent, the problems of integrity and confidentiality are the same for itinerary and state. In particular, how can *any* data computed at a remote host be kept from *unauthorized* disclosure or be kept from modification.

### A.3.18 Sliding Encryption

As an agent migrates around the network, it may carry with it an increasing collection of data results from each host that it visits. Section A.4 discusses requirements and terminology associated with agent data state protection and motivates certain integrity and privacy mechanisms. Desired properties for such mechanisms include confidentiality, forward privacy, non-repudiation, and forward integrity. We illustrate one of the simplest methods to enforce data privacy (keeping future hosts from seeing results of previous hosts) in Figure 105; it uses the public key of the agent owner to encrypt results at each host.

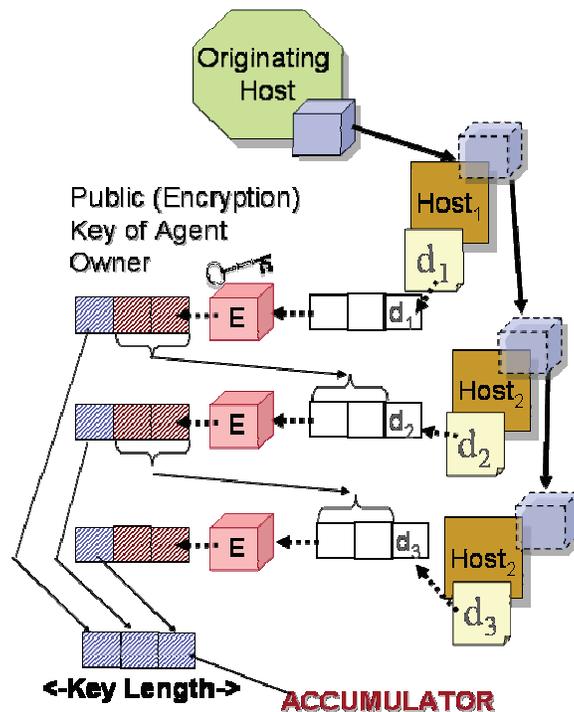


Figure 106: Sliding Encryption

This method is very inefficient when the original information is small (< 10K) in comparison to the block size of the encrypted ciphertext (~1K). The mechanism also does not prevent data alteration or removal. To solve this problem, Young and Yung [286] introduce a method that encrypts only small amounts of the most sensitive data using the public key of the agent owner using a chain relationship known as sliding encryption. Sliding encryption provides confidentiality of the data by using a storage relationship based on a randomly chosen block size that is some smaller factor of the public key size. As Figure 106 illustrates, the executing host encrypts its data result (which is small) and then chains it into an “accumulator” that is equivalent in length to

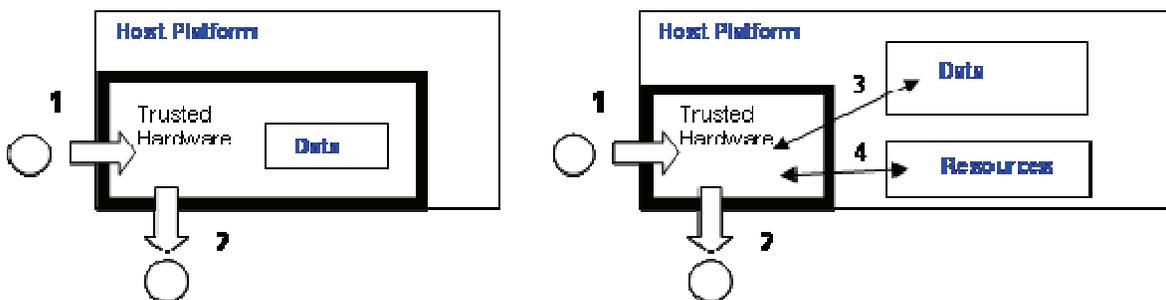
the key size. The agent only carries part of the previous encryption to the next host, which then uses it with its own input to create another small accumulator block. When the agent returns home, sliding decryption takes place beginning with the accumulator and proceeds by recovering each host result in a chained fashion, in reverse of the encryption process.

Though more efficient than using public key partial result encryption alone, sliding encryption also prevents partial results from being used in any subsequent agent computation—limiting its use in certain applications where the next host relies on the computational results of previous hosts. The mechanism comes with a computational cost involved with performing a public key operation and the associated overhead of how to distribute public key certificates efficiently. We mention sliding encryption in particular because of its identification as a popular defense mechanism in most every piece of literature, though it falls in the same category as a large number of mechanisms that provide agent data protection, discussed next.

### A.3.19 Trusted/Tamper-resistant hardware

Farmer *et al.* [23] suggest that we could solve most malicious host threats by simply disallowing agent migration outside of a trusted host environment. Since this notion is inconsistent with many real world application scenarios for mobile agents, Yee [31] suggests the use of trusted platforms or tamperproof devices that run unaltered Java interpreters in secure co-processors. Likewise, Wilhelm [76,138] defines tamperproof in the context of a full execution environment—complete with protected RAM, ROM, CPU, and volatile storage. In a fully protected environment, the host operating system must provide an interface to the host protected area—in essence creating a physical “black box” that cannot be interfered with or observed by malicious parties on the outside. By encrypting, decrypting, and executing an application (particularly a mobile agent) inside some protected computing environment, we can make strong guarantees regarding the safety and security of the execution assuming the environment is truly tamperproof.

As Figure 107 illustrates, an ideal environment is one in which all data and all resources (memory, processor, non-volatile storage, etc.) are completely embedded in the trusted environment and therefore completely shielded from any observation of the host—leaving only the arriving and exiting state of the agent open to observation (1 and 2 in Figure 107). If the host encrypts the agent upon arrival and departure, trust rests squarely on the physical security of the tamperproof devices themselves. In more probable scenarios, the trusted hardware may have partial interaction with data or resources outside the black box that come from the host platform (3 and 4 in Figure 107). In this case, the data provided to the agent computation is observable along with any use of resources such as memory or storage is observable as well. In either case, TPH reduce the security to the physical characteristics of the devices or servers themselves.



**Figure 107: Trusted Hardware – Full/Partial**

Trusted or tamperproof devices, whether provided as full or partial execution environments, can be used in a multitude of ways to enhance agent security. Karjoth [287] proposed the use of trusted devices as platforms for secure execution of certain user routines in support of e-commerce, Wilhelm and Staamann [138] suggest their use for agent itinerary protection, and Loureiro and Molva [110] suggest a combination of trusted hardware used with other security mechanisms to produce host-side privacy of computation. Since smart-cards have limited

processing capacity on a remote host, we must make some tradeoff to execute only small pieces of efficient code. Loureiro and Molva combine smart cards with their approach to privacy of execution based on encrypted circuits and secure function evaluation. There are other schemes [27,43,107,288] that use a trusted *service* or host in combination with untrusted hosts to perform services for an agent during its computation cycle. These mechanisms rely on a proxy architecture where secure servers act on behalf of agents to guarantee the integrity of their computational results.

Borselius *et al.* [259] propose two mechanisms that utilize trusted hosts in e-commerce applications: equipping more than one agent with shares of a commitment function to complete a transaction and the use of a single trusted host to allow multiple agents to report transaction information back before the owner makes a purchase decision. The latter approach would provide a safe “home base” separate from the originating host that an agent can interact with to verify results before commitments are made on behalf of a user. Many researchers like Chess [24] and Wilhelm and Staamann [76] feel that TTP offers the only feasible remote host agent protection. Sander and Tschudin [25, 159] were one of the first to challenge this assumption and pose software-only approaches. The next several sections deal specifically with software protection of remote agents without reliance on TPH. These approaches use mathematical or cryptographic methods for software hiding, thus reducing any remote host operation to blind disruption.

### **A.3.20 Function Hiding with Encrypted Functions**

Sander and Tschudin [25, 159] proposed one of the earliest software-only approaches to agent security. In particular, they did not want a solution that required interactions beyond those already assumed in a *strongly* mobile code paradigm. Programs, data, and messages need not be executed or passed in cleartext either. Many researchers consider their work seminal in describing software-based agent protection—Sander and Tschudin thus pose three important questions that deal with malicious host protection:

- (1) *Can tampering by a malicious host be prevented?*
- (2) *Can a program be concealed from a malicious host?*
- (3) *Can cryptographic operations such as a signature function be performed without revealing the private key?*

Computing with encrypted functions (CEF) tries to give an affirmative answer to all three of these questions. A companion problem to CEF is computing with encrypted data (CED), posed by Abadi and others [289, 290], whose solution requires a number of rounds of communication between parties. In CED, Alice encrypts her input  $x$  in such a way that Bob can compute  $f(x)$  without knowing what the cleartext  $x$  was. Likewise, Alice cannot learn anything specific about Bob’s private function  $f(\cdot)$  by examining the resulting  $f(x)$ . Abadi and Feigenbaum propose a mechanism to accomplish this by embedding Alice’s input  $x$  into an encrypted Boolean circuit and by performing several rounds of computation before giving Alice the final computation of  $f(x)$ . As Figure 108 illustrates, CEF is the opposite approach and expresses the mobile code paradigm where an originator (Alice) wants to execute a function (with privacy of computation) on a remote host (Bob) who will provide some private input  $x$ . In this case, Alice again will be able to decrypt the resulting  $f(x)$  without learning  $x$  and Bob will not learn anything about  $f(\cdot)$  itself.

The crux of the mobile agent paradigm is autonomy—the idea that code sent to perform a task for a user should not have to interact with the user until it is finished. Original solutions to CED were computationally infeasible because the protocol required a number of rounds based on the circuit depth. CEF eliminated this restriction while presenting a mobile code paradigm that does not rely on trusted hardware. Sander and Tschudin’s [291] CEF approach, illustrated in Figure 109, shows how Alice can conceptually encrypt a function,  $f(\cdot)$ , in some program  $P(E(f(\cdot)))$  and send Bob that program to be executed. The encrypted result can be understood by Alice with Bob not being able to discover the semantics of the original function  $f(\cdot)$ , while also being able to hide his input  $x$  from Alice. In order to achieve an encryptable program property, the user must find a transformational program  $E$  that has certain cryptographic properties, detailed by Loureiro



accomplish function hiding with other techniques beside homomorphisms in rings and groups. We discuss next a novel approach using coding theory.

### A.3.21 Function Hiding with Coding Theory

Loureiro and Molva describe another instance of computing with encrypted functions [296,297]. They rely on the security of McEliece cryptosystem and the incorporation of error correcting codes to hide the function from a potential host. Several cryptosystems exist whose provable security rests on the difficulty of decoding or finding a minimum weight codeword in a larger linear code. Though researchers view the general problem as NP-complete, some error correcting codes remain more susceptible to attack than others. By using Goppa codes generated from Goppa polynomials (adopted by McEliece in his cryptosystem), Loureiro and Molva meet three criteria to create an intractable decoding sequence: 1) a large enough code space to avoid duplication; 2) an efficient decoding algorithm for this class of codes; and 3) the generator function for the code does not leak information. As Figure 110 illustrates, Alice computes a function with parameters based on the McEliece paradigm; the protocol utilizes matrices and generators of error correcting codes.

The function computed by Bob on his input  $x$  becomes a matrix operation whereas the decryption performed by Alice is based on the existence of invertible matrices and the properties of the Goppa decode operation. Because this approach relies on the reduction of a function to a Boolean circuit as part of the computation, it suffers the same exponential increase in the complexity of the circuit as those proposed by Sander and Tschudin [25]. It also suffers from the lack of a general method to encode mobile agent program code. The fact that a code developer has to be intimately familiar with both error-correcting codes and the McEliece cryptosystem makes this solution difficult to apply in the general case.

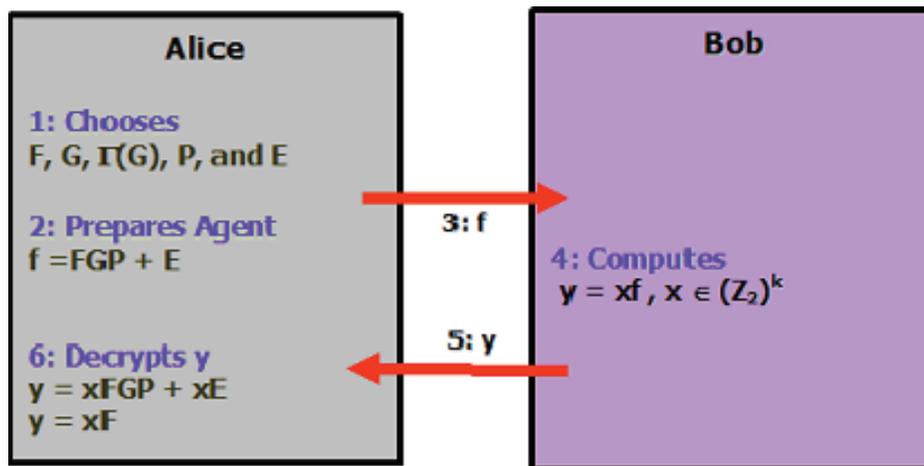


Figure 110: A CEF Based On Coding Theory

The approaches for CEF posed by Loureiro and Molva [296] and Sander and Tschudin [111] point out an important limitation of mobile agent execution first noted by Hohl in [34]: if applications require privacy of *computation*, the models for such applications are restricted because applications owners may send cleartext data only to trusted hosts. By computing with an encrypted function  $E(f(x))$  on some input  $x$ , the result  $y$  is left in an encrypted (albeit readable) form. If the agent needs to use the cleartext results of computations from other servers or even the current server, the encrypted results must be decrypted while still at the host. A problem arises: if the originator gives the decryption algorithm to the host so that cleartext results can be included in future computations of the agent, the privacy of the agent's data is at risk. Loureiro and Molva [110] incorporate trusted hardware (smart cards) to address this issue. The trusted hardware requires a smaller amount of processing to allow decryption and use of the cleartext server computation result. This computation requires less resources than the computation of the

encrypted function on input  $x$  performed by the host platform. Any executing host with computations that use the server's cleartext result ( $y$ ) can thus perform operations in an unobserved manner—preserving privacy of data as well as privacy of computation.

Zhou and Sun [298] claim to provably secure an agent against all forms of computational and data attack except denial of service using an interesting combination of the CEF protocol based on Loureiro and Molva [296] and an adaptation of reference states proposed by Vigna [267]. Zhou and Sun convert mobile agents into a series of one or more Boolean circuits and then represent each circuit as a matrix. By incorporating coding theory/CEF approaches with reference states, Zhou and Sun believe their approach prevents or detects all attacks except for DoS. As with the original protocol [296], the Zhou/Sun protocol requires an in-depth knowledge of the McEliece algorithm, coding theory, and Boolean circuit decomposition of a program in order to be practical for implementation or viable for protocol analysis.

Lastly, we envision other combinations of both prevention (privacy of computation) and detection (privacy of data) that produce similar provably secure properties. The basis for computing with encrypted functions assumes a malicious host cannot discern the original function of an agent. If the results obtained from executing the agent remain encrypted, we can guarantee privacy of computation in a mobile agent system. Thus, discovery of new methods for CEF and new homomorphic encryption schemes continue to be an open area of research. A general-purpose program encryption mechanism that is provably secure, general, *and* efficient—an encryptable Turing machine for example—remains unfound. Only limited applications for provably secure CEF using rational functions, polynomial functions, and small Boolean-circuit-reducible programs are possible as a result. The easiest software-only solutions encrypt only necessary parts of the agent computation and leave the majority observable as in [298,299]. We consider now a less provably secure method for providing privacy of computation based on masking the nature of a computation by confusion.

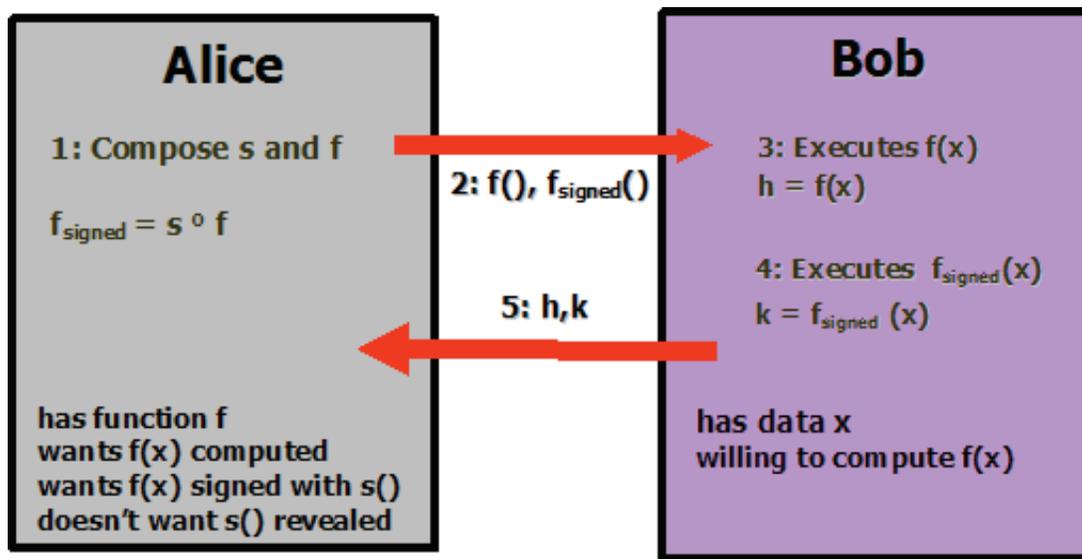


Figure 111: Undetachable Signature Scheme

### A.3.22 Undetachable signatures

Despite the fact a *general* homomorphic encryption solution to protecting the privacy of an agent program has not been found, one of the more important contributions by Sander and Tschudin [291] is the concept of an undetachable signature scheme. This particular application of CEF defines how an agent can carry a secret, such as the signature function of an originating host, and then use that secret in public. The obvious problem with releasing a signature function for use at remote servers is that a malicious server can reuse the function to sign transactions

which the original user had no intention of making—buying a new Porsche or booking a European vacation. Sander and Tschudin proposed a scheme that would compose a signature function with a task function,  $f()$ . As illustrated in Figure 111, a composed signature function can be sent from an originating host (Alice) to a host platform (Bob) allowing a task function  $f$  to be executed with Bob's private input  $x$  and then signed with Alice's signature function. The results of Alice's program with Bob's input and Alice's signature of the result are thus said to be *und detachable* from each other—in other words, the signature function  $f_{signed}()$  is only valid when used with function  $f()$ . Unfortunately, Sander and Tschudin did not propose any practical application of this und detachable scheme.

Kotzanikolaou *et al.* [30] provide a practical implementation of the und detachable signature scheme and give a real implementation of CEF with an RSA-based homomorphism. This scheme, depicted in Figure 112, is able to bind the signature of a prospective server's bid ( $bidS$ ) to a user's requirements ( $reqC$ ) in such a way that security is reducible to the strength of RSA itself. The RSA-based signature function (which raises a message to the  $d$  power, where  $ed = 1 \text{ mod } \phi(n)$ ) is computed on the hash of the bidding function  $f()$ . When the server computes  $f(x)$  and thereby places a bid, the transaction is signed based on the input  $x$  with the composed signature function. The originating customer, Alice, can therefore guarantee her signature is valid only for signing transactions computed from the function  $f()$ —namely because a malicious server would have to modify the constraints (found in  $reqC$ ) then produce a matching und detachable signature pair,  $(h',k')$ . This is only possible if one can break the RSA signature scheme.

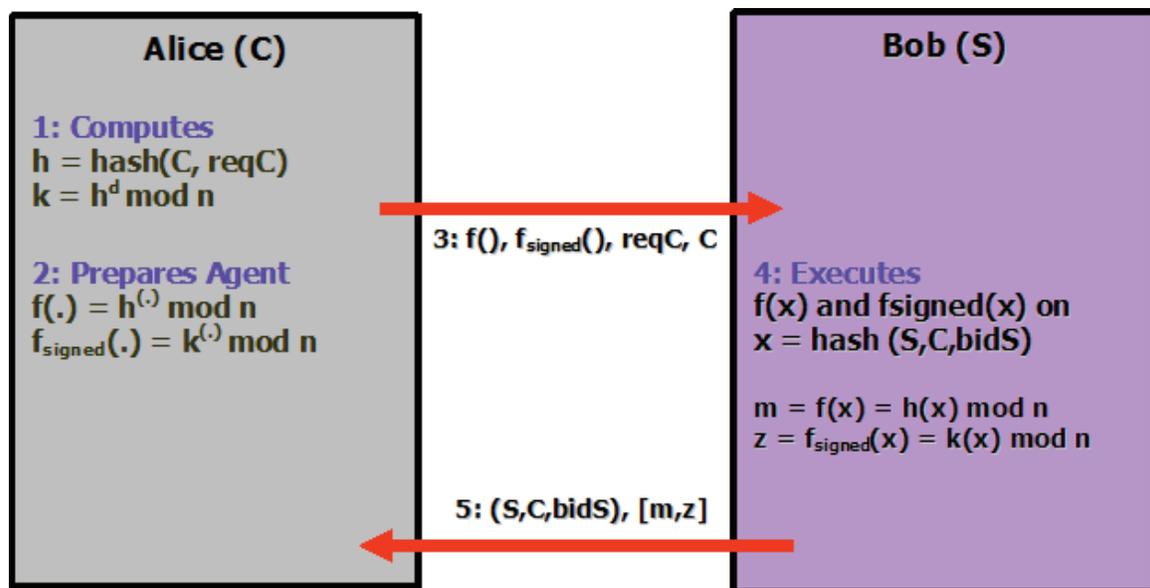


Figure 112: RSA-Based Und detachable Signature Scheme

Borselius *et al.* [29, 259] use threshold cryptography which was first proposed by Desmedt [300] in order to secure agent transactions. In a multi-party agreement scheme, we give a group of  $n$  entities shares of key in a way that  $k$  members of the group can create a valid signature together [301]. By applying threshold group schemes to und detachable signatures, we can achieve similar agreement properties in a mobile agent environment. A user may desire to use more than one agent and embed each with a share of the signature key, limiting the power of any one agent to sign agreements from coerced malicious hosts. In an e-commerce example, even though constraints embedded within an und detachable signature can limit the value of an item or the type of item purchased by an agent, a malicious host can still commit the agent to a less than ideal transaction.

By deploying a certain number of agents for accomplishing a task,  $k$  of  $n$  agents can be used to distribute the signature function of the agent to ensure agents are treated fairly (the threshold aspect) while also constraining the type of transactions any one agent can sign (the undetachable aspect). We can thus verify the correctness of a signature by a group of cooperating agents (or trusted third parties) and we constrain agent power to at least  $k$  of  $n$  uncompromised parties. Borselius *et al.* [29] also present an alternative to the RSA-based detachable signature of Kotzanikolaou *et al.* [30] using conventional signatures and public keys. This scheme relies more on secure delegation techniques that allow an agent to carry certification authority for only certain transactions. The non-RSA approach proves to be more efficient because only one signature (versus two) is required for the agent and the user need only generate a valid key pair and certified public key—saving one exponentiation. Group agreement schemes such as undetachable threshold signatures are very similar to another family of protection schemes that rely on the power of multiple parties for security, discussed next.

### A.3.23 Policy Management Architectures

Policy management is a method for both malicious host and malicious agent protection, mentioned here for completeness. Varadharajan and Foster [302] and Shi *et al.* [303] point to the need for an architectural view of agent security management. Policies play a key role in defining such architectures and the incorporation of security mechanisms will confront system designers for years to come. Because the success of the mobile agent paradigm is security dependent, we must consider architectural viewpoints as well. Schoeman and Cloete [65] mention little reuse currently of agent architectures or reintegration of lessons learned from various research efforts. Policy management plays a key role in future standards and thoughts for common mobile agent frameworks.

## A.4 Agent Data Protection

This section provides an overview of related research material to results present in Chapter 3.

Privacy problems in mobile agents are invariably the source of many fears that prevent widespread deployment of agent systems currently. A large body of research work reveals a variety of proposed defense mechanisms that fall under the category of data state protection. We list a large assortment of protection mechanisms reviewed in this section in Table 24.

**Table 24: Data Protection Mechanisms**

Section	Data Protection Mechanism
A.3.18	Sliding Encryption
A.4.1	Digital Signature Protocol
A.4.2	One-Time Symmetric Keys
A.4.3	Bitmapped XOR Protocol
A.4.4	Targeted State
A.4.5	Append Only Container
A.4.6	Multi-Hops Integrity
A.4.7	Partial Result Authentication Codes
A.4.8	Hash Chaining
A.4.9	Set Hash Codes
A.4.10	OKGS
A.4.11	Configurable Protection
A.4.12	Modified Set Authentication Codes
A.4.13	Chained IP Protocol
A.4.14	EIGamal Encryption
A.4.15	Protocol Evaluation

Cartrysse [304] together with van der Lubbe [294,295] discuss the privacy and secrecy issues of mobile agent systems and mention that agent privacy falls under the larger umbrella of privacy-enhancing technology (PET). Accordingly, they cite several traditional methods to achieve privacy cryptographically such as blind and partial-blind signatures and pseudonym systems—all of which operate on the assumption that agents execute on fully trusted hosts. In the mobile agent case, however, integrity of partial results requires other non-traditional means. Figure 113 illustrates a conceptual view of data protection offered by Cartrysse [304] where an agent consists of static code, static data, transmitted data, and dynamic data. Mechanisms (internally based rather than policy-based) reside within the agent to ensure both privacy and trust maintenance. For each type of data, there is some information regarded public (not requiring confidentiality) and some information regarded private, which requires protection. All types of data—public or private—require integrity verification (protection from alteration). If the static code (agent function) needs protection, several mechanisms may accomplish this.

Under PET concepts, static *data* consists of information present in the agent before dispatch; dynamic data on the other hand exists as part of the agent's interaction with the host environment. Transmitted data includes communications with other agents, entities, or the host itself. Private static data can be information that will be eventually processed by the agent, but that requires protection until that time (to-be-processed). Some private static data may remain completely read-only and further concealed for certain remote hosts or intended for use only by the agent itself. For transmitted data, both private and public transmission require some level of time-stamping and integrity, while private read-only data still requires confidentiality. Dynamic data has the same descriptive categories as static data, where read-only dynamic data remains private for certain group members or only for the agent itself. The easiest data configuration consists of publicly accessible functions (static code) whose entire inputs (parameters given by each remote host in an itinerary) and outputs (intermediate host results) are publicly viewable. Every agent application that has some level of desire data privacy.

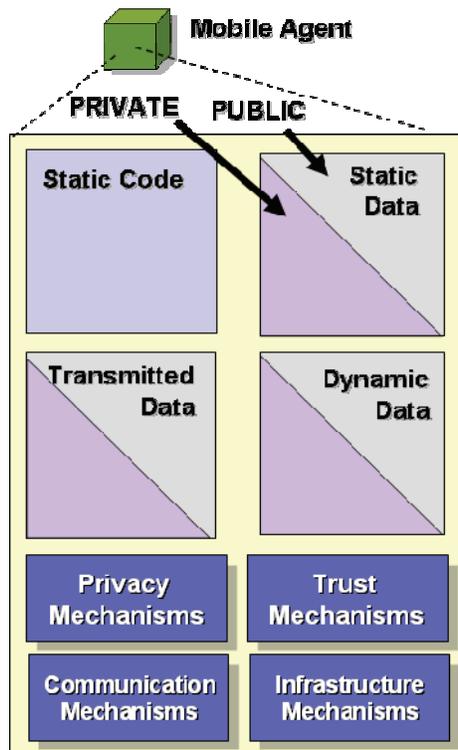


Figure 113: Public/Private Data

A particular class of malicious host does not want to subvert or actively attack an agent through code alteration. Instead, the host only wants to steal information or algorithms from a passing agent. Cartryse and van der Lubbe [2002a] present solutions for three problems in this vein: private communication protection, task information privacy (the threshold amount for purchase of an airline ticket or commodity for example), and secure private key transport (for signature generation). Their approach to task *information* privacy is of interest for data protection and discussed further in Section A.4.14.

We now review mechanisms that provide partial data result protection (private dynamic data) in various contexts. We can readily see data *collection* in the typical e-commerce bidding application. Notably, several authors point to the competitive bidding scenario and the role of the agent to gather bids from each prospective vendor [31, 144]. The possibly malicious intent of any one of the servers to tip the scales in their favor is the motivation behind several protection mechanisms.

#### **A.4.1 Digital Signature Protocol**

A method of protection similar to sliding encryption or public key encryption of the partial result can offer greater integrity protection. In a digital signature approach, each host signs its data result after encryption takes place (normally with the public key of the agent owner). Using this method, an intermediate honest server in the path can verify that offers from previous hosts remain unaltered, thus contributing to overall security. As with public key encryption mentioned above, the size of the agent will continue to grow as it collects partial results along its itinerary. Simple schemes also offer no protection against various integrity attacks where malicious hosts delete or insert results illegally.

#### **A.4.2 One-Time Symmetric Keys**

Several papers have proposed intermediate data protection mechanisms based on symmetric keys, one-way operations such as hashing, or reversible operations such as XOR. Sobrado specifies a simple scheme in [305] that does not require public key infrastructures, does not require agents to carry keys, allows revisits of an agent to an already visited host, and does not require the host to remain online during the entire transit of the agent. Figure 114 illustrates the protocol and the essence of the protection mechanism. As an agent visits each host in the itinerary, the remote host must generate a random one-time key and then use this key to perform message protection.

The “pad” in this case refers to both the specialized method for generating a signature proposed by Sobrado [305]. The signature is comprised of a codeword and message field integral to the encryption process. The approach uses a random number for the code word and uses rotated data bytes in an XOR operation with the one-time key generated by the host. Each host generates and signs/encrypts its own data result, which the agent carries to the next host in the itinerary (Figure 114-2c and Figure 114-3c). Each host in the route (host 1 and 2 for example in Figure 114) keeps this one-time key until the agent owner asks for it a later time (Figure 114-4a/b and Figure 114-5a/b). The agent owner recovers the symmetric key from each host visited by an agent later. If each host guarantees the one-time use and subsequent deletion of each key, this scheme achieves data privacy and authentication without requiring the agent to carry keying material. This method parallels the asymmetric key approach except the protocol must provide the public key of the originating owner to each remote host, either carried by the agent or provided by a certificate authority (PKI) out of band. An agent can also revisit a host because the originator does not request the symmetric key until after the agent returns home.

A host can locate its own information based on the digest/code-word of the register fields embedded in the agent and replace it with a newly generated ciphertext, create from another different one-time key. Though interaction is not required immediately, it is required at some point in order for this scheme to work (the owner has to communicate with each host to transfer the symmetric key). The one-time scheme protects against counterfeiting of data but does not address integrity attacks where malicious hosts delete partial results altogether or make insertions illegally. It counters agent “brainwashing” by reverting to the use of a trusted third party referred to as a *route server*. Agents can communicate path information (or possibly other

information) to one or more route servers accessible in a network (Figure 114-2d/3d). Like many other schemes, it does not offer full protection apart from using trusted intermediaries or tamperproof hardware and assumes software-only protection is not possible.

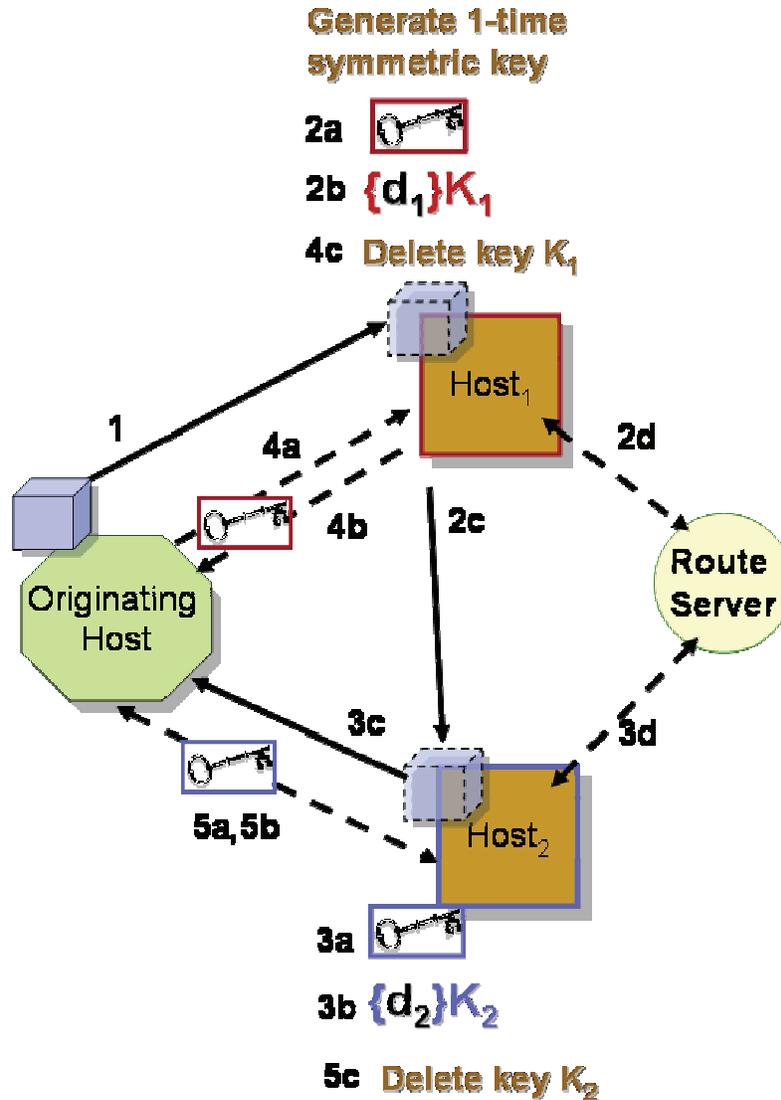


Figure 114: One-Time Protection

#### A.4.3 Bitmapped XOR Protection

As the sliding encryption model illustrates, not all remote data needs to be protected—only that which we classify as private in some way. A “fast and easy” approach is developed by Diaz and Gutierrez [306,307] under this assumption. We consider it fast because it does not rely on cryptographic techniques like digital signatures and we consider it easy because it relies on basic bit operations. In the mechanism, the agent owner builds a data table—one copy remains with the owner and the other embedded in the agent. Figure 115 shows the notional interaction of the protocol—namely the application owner provides one row for each intended host that the agent plans to visit—thus limiting this approach to bounded itineraries. Each row in the matrix has

several fields: a host identifier, the data gathered from a particular host, a random number code word created at the remote host, and a cyclic redundancy check (CRC).

In the operation of the protocol, the owner fills the table with random numbers to create a bitmapped basis for encryption. The originating host uses the original copy of the matrix to recover data upon return of the agent. As the agent visits each host, the host creates a duplicate of its original data row for its own archival and then inserts its identification and a random code word into the table. Data blocks are loaded into each block through by means of applying the XOR operation to the random number already in the agent—overwriting the existing number with encrypted data. Because the application owner cannot detect some alterations where CRC protection is used, the executing hosts apply rotations to both data and code words in a recoverable manner. Servers use the next set of free rows to store their data. Unfortunately, the mechanism does not provide a way to prevent deletion of results—only detection that a server has acted dishonestly. The agent owner can re-insert the original code word into the matrix for decryption operations in reverse of the host algorithm.

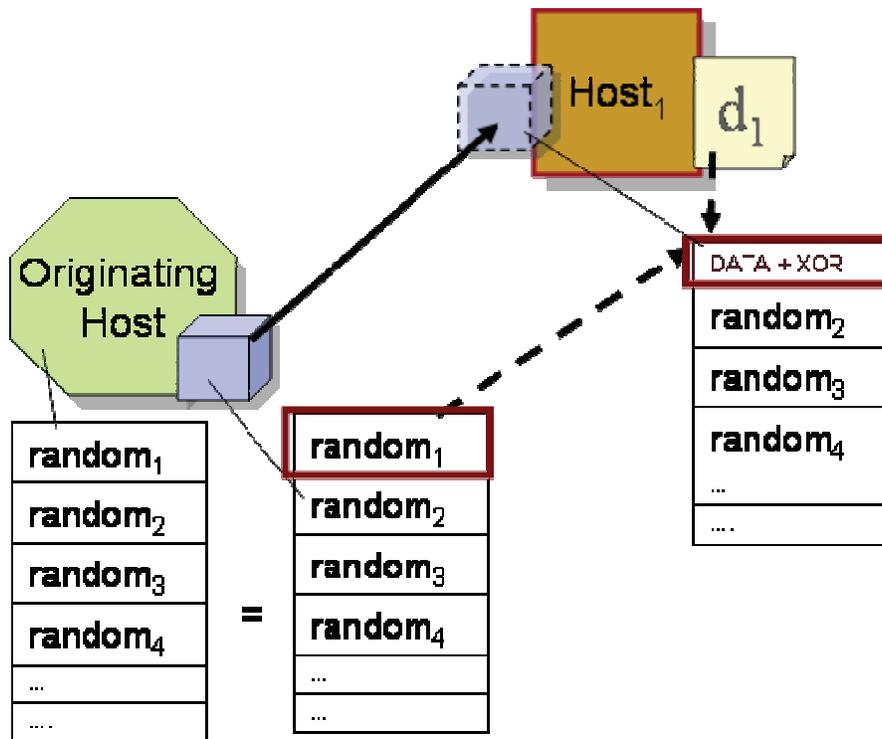


Figure 115: Bitmap/XOR Data Protection

Bitmap XOR operations are particularly useful for small data items such as dynamically generated itinerary information, as long as we place some reasonable bound on the number of hosts. Malicious hosts may attack the method by copying the data table in hopes that an agent will revisit (or that a colluding malicious partner resends an agent). If this occurs, a malicious host could perform the same decryption as the originating host. Diaz and Gutierrez suggest a remote host can provide its own random bit row for XOR encryption to eliminate this vulnerability and allow hosts to update their own data. The executing host, however, must still communicate the random number data back to the originating host.

#### A.4.4 Targeted State

Karnik and Tripathi [284] deal with the problem of providing private data *from* the agent *to* a remote host by using targeted states. Unlike protecting data gathered by the agent, this model

assumes agents carry data that is private except only on certain hosts, thus requiring protection from observation by other malicious hosts or agents. A remote host receives from a visiting agent a signed collection of states, which it verifies, and then performs inspection to see whether it can decrypt any target state using its own private key. If so, the host decrypts the state and makes any plaintext available to the agent. Figure 116 depicts the interaction of this protocol.

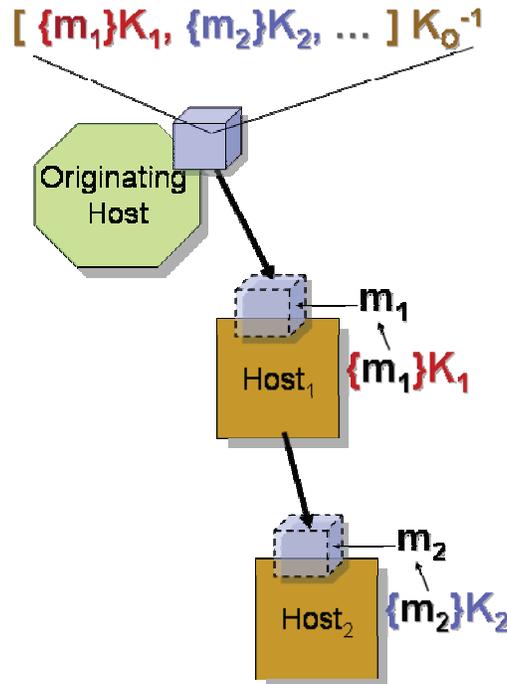


Figure 116: Targeted State Protocol

#### A.4.5 Append-Only Containers

Karnik and Tripathi [284] also introduce the append-only container that considers the integrity of the overall set of data items an agent carries, as opposed to just the confidentiality or integrity of a single datum. As the name implies, an executing host can only append new items to such a container and the originating host can detect modifications to existing items in the container. In order to initiate the protocol, an originating host creates an initial container with a secret random value and computes an initial checksum encrypted using the owner's public key (seen as  $\{r\}K_0$  in Figure 117.). Intermediate executing hosts can use checksums of the item set to provide intermediate verification as an agent visits each host in the itinerary. The append-only container is thus defined as an ordered set of intermediate results plus a checksum, whose value will be determined by the each host in turn:  $[\{d_1\}K_1^{-1}, \{d_2\}K_2^{-1}, \dots, \{d_n\}K_n^{-1}, C_n]$ . The previous hosts' checksum becomes a part of that host's data item so that each host forms a chained relationship with each previous host. Each host also applies a digital signature to its data result (seen via encryption with  $K_1^{-1}, K_2^{-1}$  in Figure 117). As an agent visits a new host, the host signs its data and includes it in the append-only container (AOC in Figure 117). The host then computes a new checksum using the previous checksum and its own signed data result.

#### A.4.6 Multi-Hops Integrity

Corradi *et al.* [308] envision a similar protocol that provides chained protection of data results from intermediate hosts. In their approach, the host begins again with a random number (indicated by  $nonce_0$  in Figure 118) and three data items: a message authentication code ( $MAC_0$ ), a data set ( $Data_0$ ), and a multi-hops code set ( $MHC_0$ ). This protocol sets up a chaining

relationship that not only ties the agent to the previous host (and the set of results collected so far, but also incorporates the identity of the current host and the identity of the next visited host). As the agent visits the next host in the itinerary, the host generates a new random number (nonce) by hashing the value of the previous nonce (which is itself a hash value). Likewise, the current hosts' data result ( $d_n$ ) is combined with the previous nonce ( $nonce_{n-1}$ ) and the previous message authentication code ( $MAC_{n-1}$ ) in a hash that binds the current hosts identity to it (seen as  $id(Host_n)$  in Figure 118). The current host then signs the MAC by the current host and then adds it to a set of previous signed MACs in  $MHC_n$ . The host concatenates the data result itself to the previous data chain and binds it with the identifier of the current host. The agent migrates to the next host in the itinerary by sending  $MHC_n$ ,  $Data_n$ ,  $MAC_n$ , and the current  $nonce_n$  encrypted with the public key of the next host to be visited ( $\{nonce_n\}K_{n+1}$ ). On return to the originating host, the owner of the agent should be able to create a non-repudiable chain of the agent's activity based on the one-way hashing operations and the public keys of each host the agent has visited.

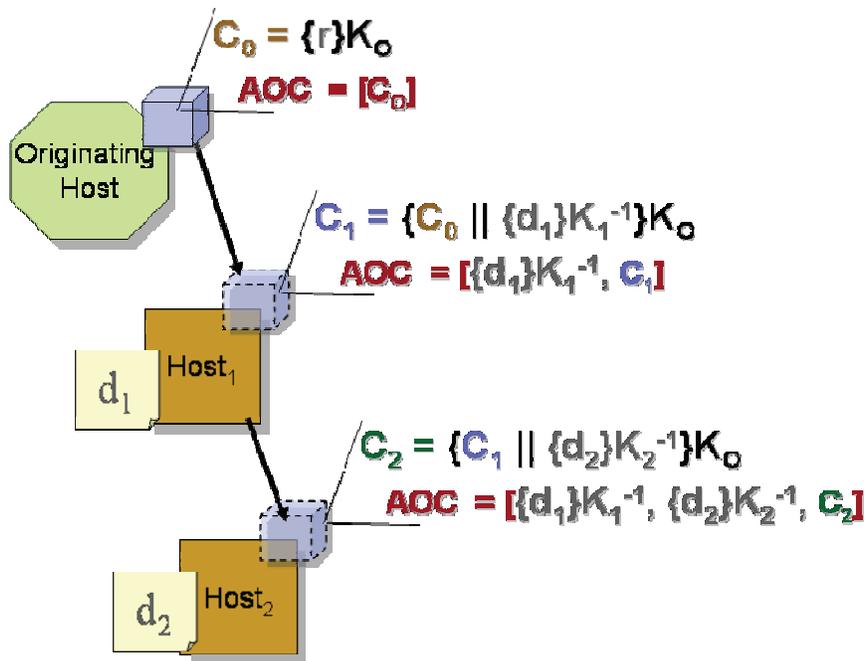


Figure 117: Append-Only Container

#### A.4.7 Partial Result Authentication Codes

The multi-hops protocol uses the notion of a message authentication code, which incorporates hashing to provide integrity verification of various values. Yee [31] offered original notions of such integrity checks called partial result authentication codes (PRAC). PRACs are key-based hashes that offer better efficiency compared to a digital signature and provide a weak form of *data forward integrity*. MACs, when used in the mobile agent context, support more than just authentication of origin; they also support verification of an intermediate result computed by any previously visited host. MAC verification assumes parties share a common key by which both sides verify integrity of data. Yee proposes three variations of PRACs for use in agent contexts.

**Simple MAC-Based PRAC.** In the simple case, Yee's protocol requires an originator to generate and keep a sequence of symmetric keys carried by the agent that are used per-server in generating an encapsulation of the agent's activity. The agent also sends a *summarized* version of the local host execution back to its owner before migration to the next host or carries it for transmission later. As seen in Figure 119, a major assumption in the approach is that an agent

can dispose of a key before visiting the next host in the chain—a requirement that is hard to enforce without the help of a trusted third party or trusted hardware. The method also restricts the agent itinerary to a known path because the application owner must know the number of keys beforehand. The simple approach would guarantee a form of forward data integrity assuming keys are not stolen or deleted properly.

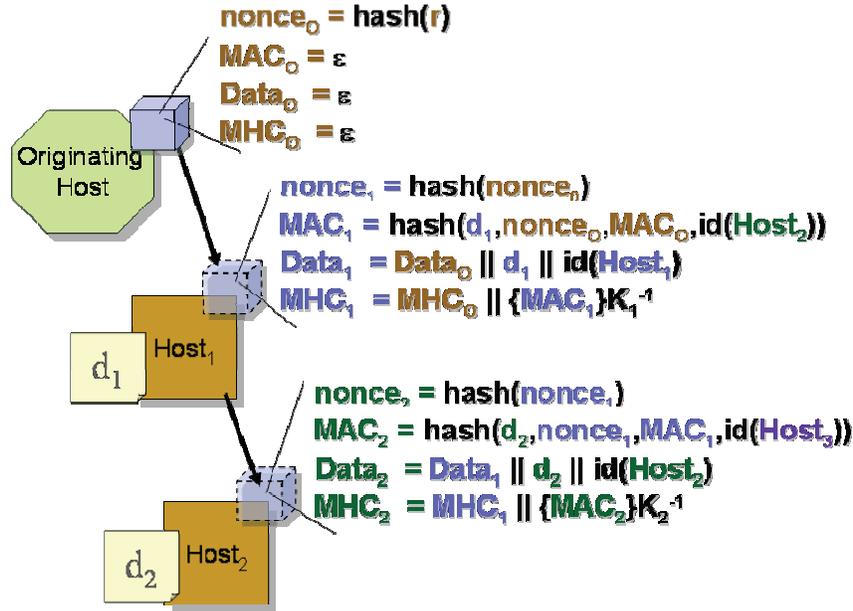


Figure 118: Multi-Hops Protocol

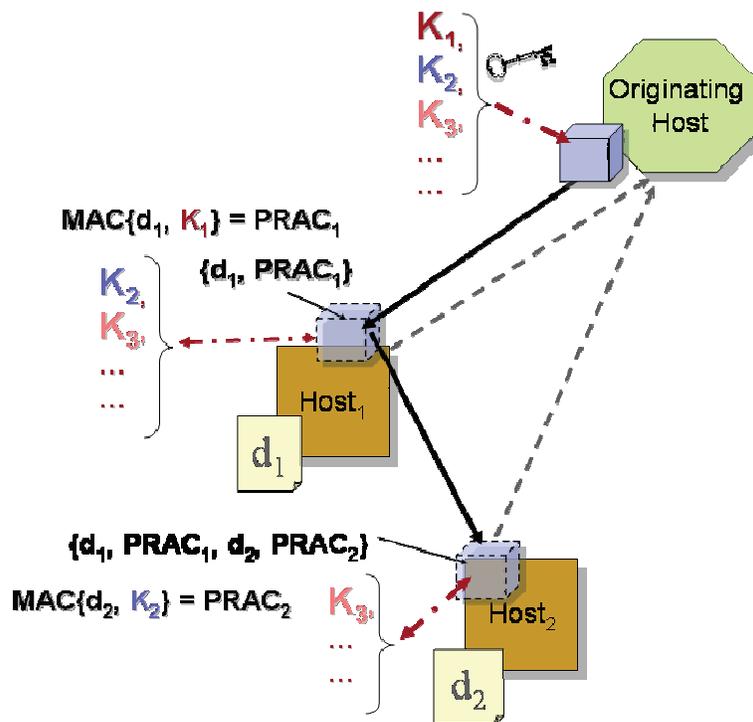
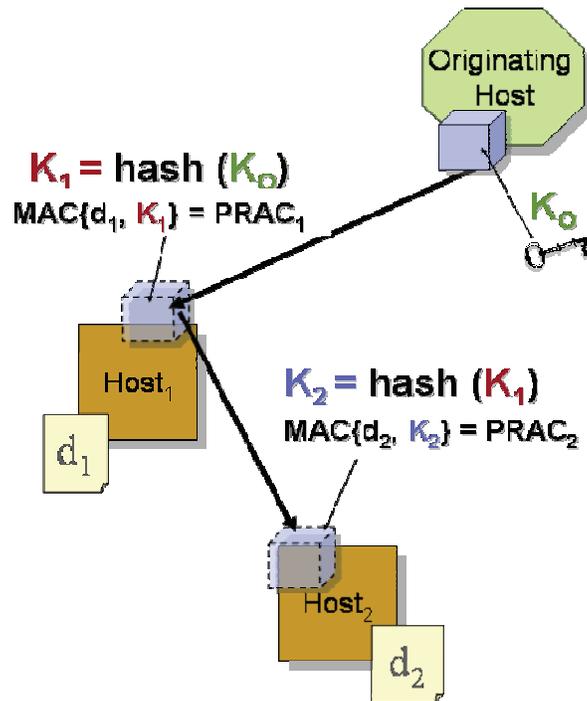


Figure 119: Simple MAC-based PRAC

**MAC-Based PRAC with Hash.** Instead of  $n$  keys that the protocol distributes to some  $n$  servers under the simple PRAC scheme, the owner holds a single key  $K_0$  used to create a one-way data stream to generate a series of keys used per-server. Figure 120 shows how an agent only need carry the original key and then each host produces its per-server key by hashing the previous key. The host still bears the responsibility of erasing knowledge of any key that it generates, very similar to approaches discussed in Section A.4.2 and Section A.4.10.

**Publicly Verifiable PRAC.** To overcome the key theft issue, Yee suggests achieving forward integrity by using a third party time-stamping service and digital signatures—providing a public means of execution verification. Instead of secret symmetric keys, a third alternative uses public keys and signature functions and allows honest nodes to participate in the detection process. The intermediary does not need to know the shared secret between an originating host and an already visited host in order to verify tampering has not occurred. Instead of loading the agent with a set of shared keys, the originating host loads the agent with a set of signature functions, each with its own unique verification function (see Figure 121).

The verification function ( $verif_n(d, c)$  seen in Figure 121) is considered public (a remote host in the itinerary can provide a publicly available certificate as input) while the signature function itself is considered private. The agent signs each partial result with the signature function of that particular host ( $sig_1(d_1)$ ,  $sig_2(d_2)$ , etc. in Figure 121) and subsequent hosts can run the verification function as the agent traverses the network to detect modifications. The protocol relies on an intermediate remote host to delete its signature generation function, however, and there are no guarantees that a malicious host can not alter or change signature or verification functions of hosts yet to be visited (though that can be detected after the agent returns home by the originator).



**Figure 120: PRAC with Hash-Based MAC**

As another variation, the method can reduce the secret signature function to a single version held by the originator and a one-way relationship established as in the PRAC with hash-based MAC version. An agent creates the partial result signature by executing  $sig_n(d_n)$  and then deleting it, as in the normal version. Then the agent creates a new signature and verification

function pair that supports certification of the new signature on future hosts. In this way, the protocol defers signature generation to the executing host and not the originating host (which *might* have less computational power).

As a benefit, the agent itself (as well as other honest hosts) can verify the status of its data storage area. We consider Yee's methods to provide weak forward integrity because it cannot prevent certain data integrity attacks when hosts cooperate maliciously. PRAC incorporates backward chaining but cannot prevent colluding hosts from truncation attacks, especially in the case where an agent visits the first host in a chain again. When the application allows agents to revisit prior hosts, a malicious host can delete data results captured between the first and second visits and alter the path of the agent to exclude other servers completely. A malicious server may also change a previous bid illegally (once it has gathered more information from other hosts in the environment). We must consider forward integrity in the face of such activity differently.

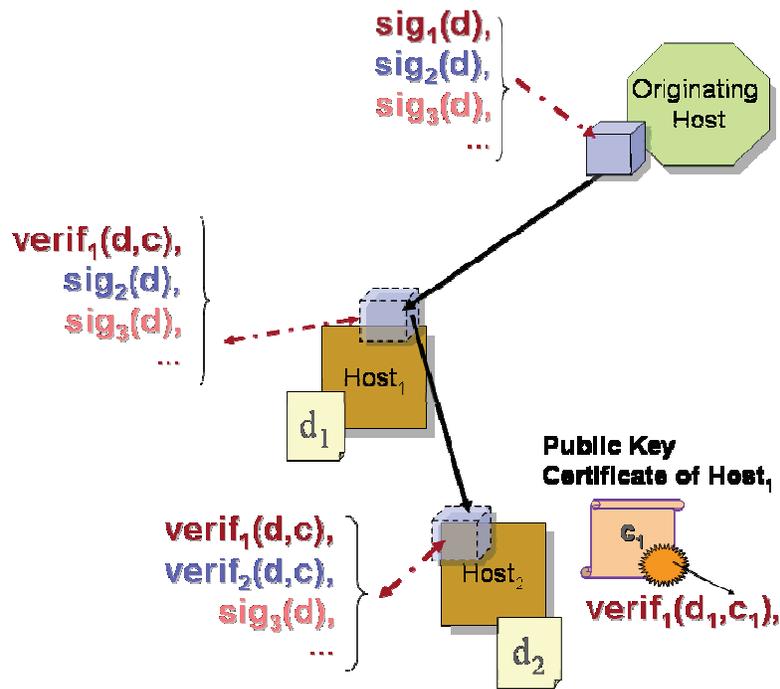


Figure 121: Publicly Verifiable PRACS

#### A.4.8 Hash Chaining

In order to account for collusions, Karjoth *et al.* [144] define at least three alternatives to defend against truncation of agent data when the path of an agent is not known in advance: 1) embed some part of the itinerary in the agent; 2) have the originating host broadcast an agent's path after the fact to allow servers to verify their results were not deleted or altered; or 3) create a verifiable function that computes the next hop in the agent path (forward chaining). Karjoth and colleagues propose four different protocols that incorporate forward chaining and achieve varying levels of strong forward data integrity and confidentiality. We refer to this approach as hash chaining or partial result encapsulation and observe that it resembles the Corradi *et al.* [308] multi-hops protocol. Hash chaining protocols link the current agent state to both the previous state of the agent and the identity of the next visited host—all of which combine to offer stronger resilience against certain collusions.

Karjoth *et al.* [144] define a chain  $O_1, O_2, O_3, O_4, \dots, O_n$  as an ordered set of data items (offers) that are collected by an agent from a set of  $1 \dots n$  hosts. Dependence exists between each element of the chain with a previous element and/or possibly the next element in the chain. Based on the security properties defined in Table 5 and Table 6, the following mechanisms

attempt to strengthen PRAC-based protocols from Yee [31]—making it impossible for an executing host or colluding partner to forge a previous data element. There are three essential elements that define each protocol: the definition of the encapsulated offer, the chaining relationship (hash code), and the protocol for migration, each illustrated in Figure 122 and discussed in detail for each variation. In a hash chain relationship, the encapsulated offer is composed of some set of data (the data result of the current host, a random nonce, a hash code, etc.) that is either signed or encrypted by the remote host. Depending on whether public verification or privacy is more important, future agent servers may be able to read data but not modify it without detection. Likewise, the protocol can hold data private so that only the originating host can read it or detect that integrity violations have occurred.

**Publicly Verifiable Chained Digital Signature (PVCDS).** The first approach Karjoth *et. al* [1998] describe is an encapsulated signature chain that extends Yee’s [31] per-server digital signature. The originating host generates a random number and hashes it together with the identity of the first host in the itinerary. Every subsequent host takes a hash of the previous encapsulated offer and the identity of the next host as the value for the “hash” part of the encapsulation, as seen in Figure 123. The encapsulated offer is a publicly signed package of both this hash value and a set of encrypted data offers (encrypted with the public key of the owner normally). This provides forward and backward chaining which strengthens resilience against modification without detection. Figure 123 gives an overview of the specific PVCDS implementation.

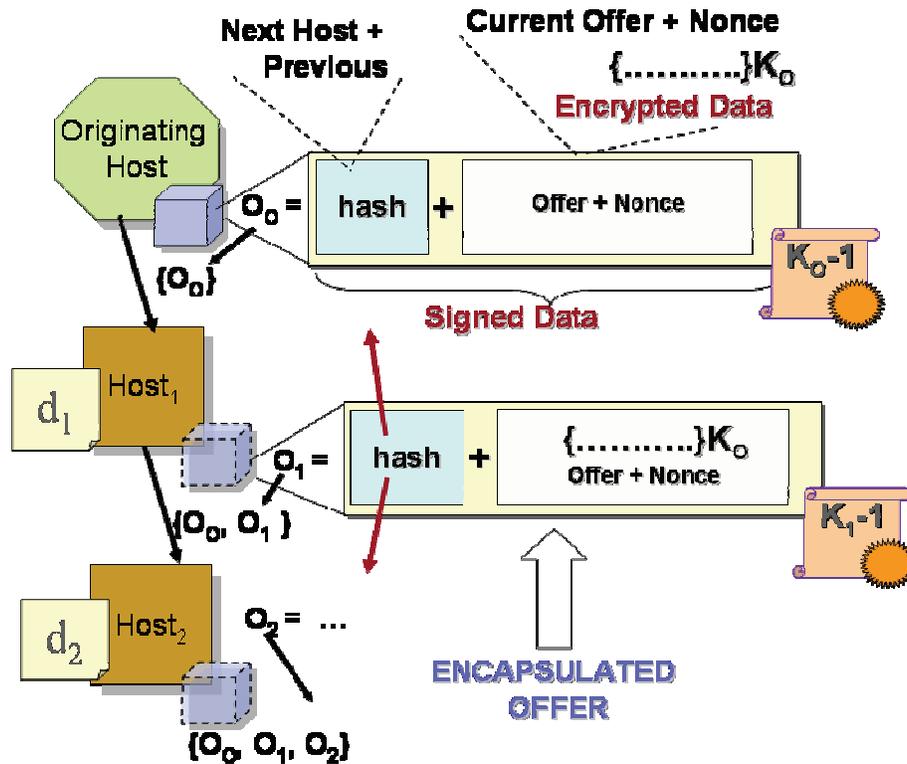
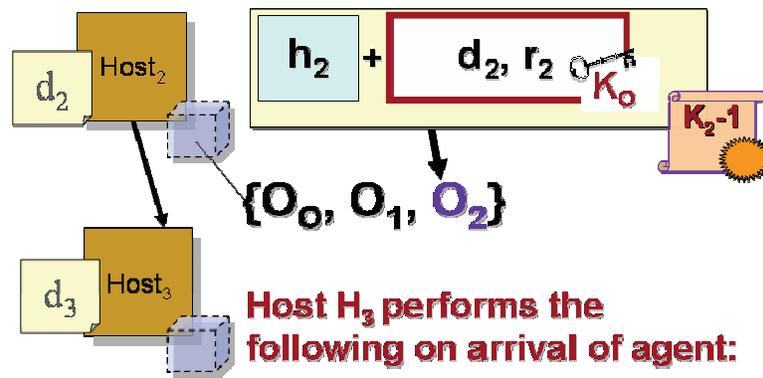


Figure 122: Encapsulated Offers

The originating host begins the hash chain by encrypting the random nonce and a token with its public key. Once the application owner dispatches an agent, the agent collects a set of encapsulated offers, each linked to the previous and next host in the itinerary. Figure 123 illustrates the steps of the protocol for a notional third host (Host<sub>3</sub>) in an itinerary that has just received an agent from host (Host<sub>2</sub>). Host<sub>3</sub>, in Figure 123, relies on the encapsulated offer of the previous host (Host<sub>2</sub>) to derive its own hash value and links its execution explicitly to the next host in the itinerary. In this mechanism, a server can verify links in the chain by decrypting any given  $O_n$  in the set of encapsulated offers. Since each offer is composed of  $h_n$  plus the encryption of

$(d_n, r_n)$ , intermediate hosts perform signature verification en-route to ensure data migrations have not been altered. Whether positive or negative, a server cannot modify its own result even if an agent revisits. Another downside to the approach is owner must determine the itinerary beforehand and must perform final verification of the entire chain sequentially when the agent returns home. PVCDS provides data confidentiality, non-repudiation, strong forward integrity, publicly verifiable forward integrity (signatures of each intermediate host can test underlying links), insertion resilience, and truncation resilience.

**Chained Digital Signature with Forward Privacy.** A slight variation of PVCDS involves swapping the ordering for encryption and signature generation. In this case, a particular encapsulated offer is itself an encryption (as seen in Figure 124 for a notional  $Host_2$ ) of the chained hash ( $h_2$ ) plus a signed copy of its data result with nonce. This variation enforces forward privacy but the protocol does not support publicly *verifiable* forward integrity—only the originating host can decrypt any given encapsulated offer. It is a desired feature of any protocol to enlist the help of honest nodes to perform integrity checks on the agent.



1. **Verify  $O_2$  (public key of  $K_2$ )**
2. **Compute offer =  $d_3$**
3. **Generate  $h_3 = \text{HASH}(O_2, \text{id}(\text{Host}_4))$**
4. **Generate nonce =  $r_3$**
5. **Encrypt  $(d_3, r_3)$  w/ owner's public key  $K_0$**
6. **Sign  $h_3 + (d_3, r_3)K_3$  with private key  $K_3^{-1} = O_3$**
7. **Add  $O_3$  to set  $\{O_0, O_1, O_2, O_3\}$ , send to  $d_4$**

Figure 123: Protocol Interaction of PVCDS

**Chained MAC Protocol.** Since both variations of PVCDS assume a PKI is in operation, Karjoth *et al.* posed two variations for hash chaining that do not rely on this assumption. Instead, the only requirement is that each agent server knows the public key of the originating agent (which is an easier key distribution problem). As an extension to Yee's MAC protocol, Karjoth takes advantage of the property where a key-based message authentication code (MAC) ties an intermediate agent data state to a particular host platform. As seen in Figure 124 each encapsulated offer is afforded privacy by encryption with the originating owners public key ( $K_0$ ) and each subsequent host ( $Host_3$  for example) is provided a hash chain value ( $h_3$ ) that is computed for it by the previous host ( $Host_2$  in Figure 124). This chaining relationship ties together all previous results up to that point in the itinerary and binds the identity of the next host in the chain. The random nonce included with each hash chain and encrypted by the host (for example,  $r_2$  for  $Host_2$  in Figure 124) prevents a future agent platform from being able to replace an encapsulated offer as well. The embedding of the identity of the next host does not provide

authentication of a given server, but allows the originating host to track agent migrations against results embedded in the data state.

Karjoth *et al.* note also that embedding of the hash value itself ( $h_n$ ) into the encrypted offer ( $O_n$ ) could allow the originator to determine which malicious server broke the chaining relationship. The chained MAC protocol, according to the authors, provides data confidentiality, strong forward integrity, and forward privacy. Strong forward integrity is induced here because it is not possible to modify a given encapsulation  $O_n$  without also modifying  $O_{n+1}$  and  $h_{n+1}$  while still maintaining the correct chaining relationship.

**Publicly Verifiable Signature Chains.** A final extension to Yee's [31] publicly verifiable PRAC where a secret signature and verification function pair is embedded in the agent is depicted in Figure 125. In Yee's approach, a server signs its partial result using the signature function carried by the agent and then certifies the next signature verification function that will be used the next host in the itinerary. The host destroys the signature generation function it receives but adds its certified verification function to the agent. We can use both one-time and public key signatures to sign intermediate results and Karjoth *et al.* [144] chose to use only the public key of the originating host to set up the chaining mechanism.

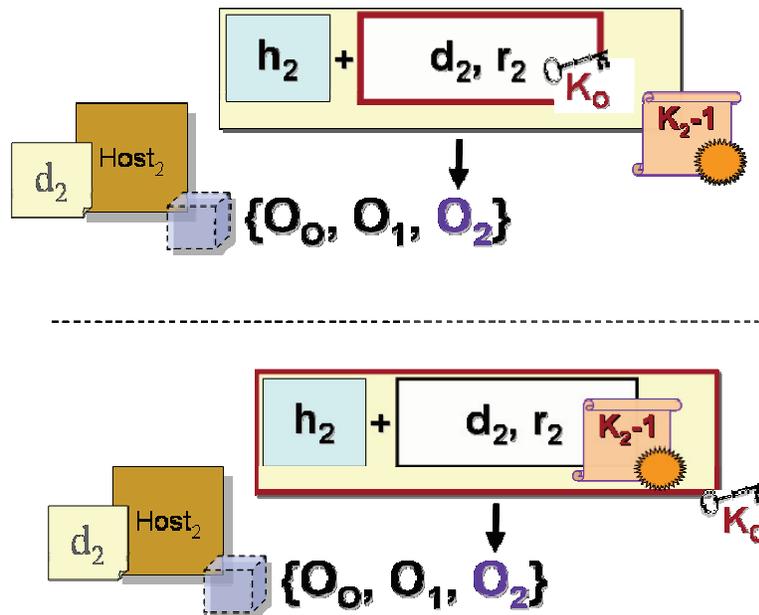


Figure 124: Forward Privacy

Figure 126 illustrates publicly verifiable signature chains that use a one-time public/private key pair generated by each remote host. The generation process relies on a chain starting with the public key of the agent owner. Each host in turn generates and sends a signature key to its successor (seen as  $OTK_3^{-1}$  in Figure 126) and provides certification for the corresponding public key (seen as  $OTK_3$  in Figure 126). The current host signs each offer,  $O_n$ , digitally with the secret key received from the previous host. The hash in this case remains embedded within each encapsulation object and provides a forward and reverse link. This protocol when combined with features from the chained digital signature mechanism provides both publicly verifiable forward integrity and forward privacy.

The protocols of Yee [31] and Karjoth *et al.* [144] have vulnerabilities to include threats from colluding hosts. Roth [109] identifies the root cause as agents which do not have a specifically identifiable kernel (static code), thus allowing oracle and cut-and-paste replay attacks. Cheng and Wei [309] enhanced the publicly verifiable signature scheme of Karjoth *et al.* [144] to shore up truncation-attacks launched by two hosts in partnership. In their approach, a counter-

signature is required from the preceding host (almost like a counter-signed check) before sending the agent to the next host in the path. Zhou *et al.* [310] find further weaknesses in the Cheng/Wei [309] mechanism when loops are present in the agent's itinerary, but are able to deter more advanced truncation attacks by requiring two hosts to co-sign an agent's integrity checksum/hash value.

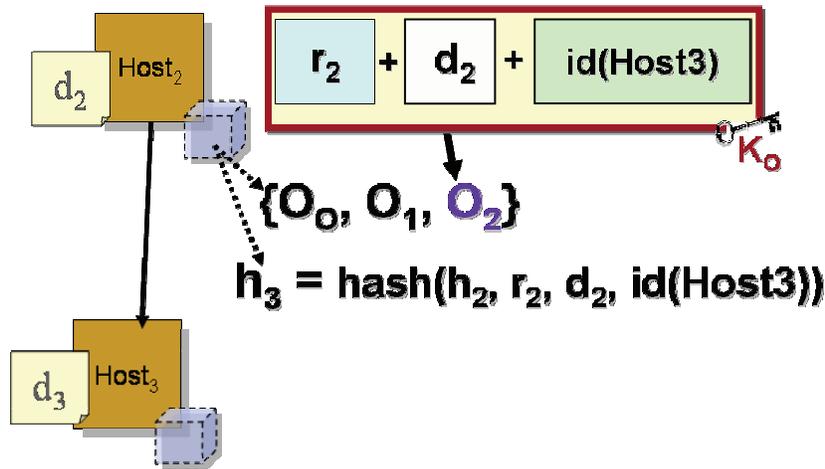


Figure 125: Chained MAC Interaction

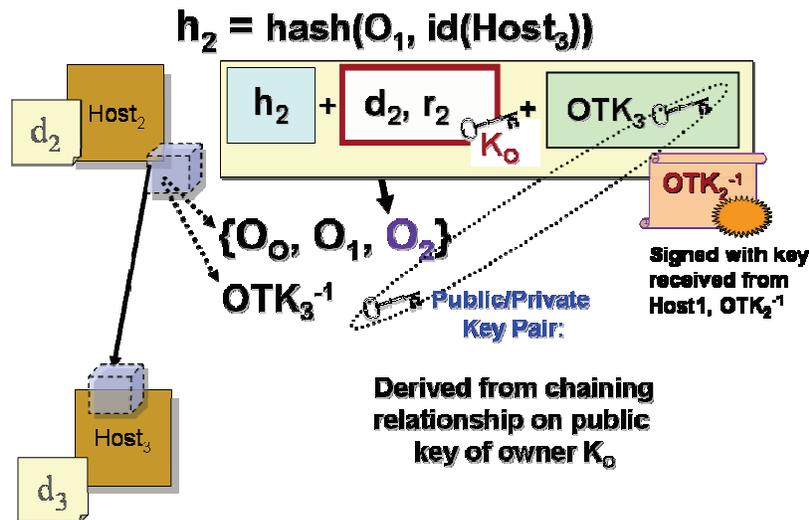


Figure 126: Publicly Verifiable Signature

#### A.4.9 Set Hash Codes

In many ways, we can tie data integrity in the presence of malicious hosts to the itinerary of the agent: how the agent travels in performing its task. An agent (by luck) can visit the only malicious host in a network first and thus have less worry about deletion or substitution. On the other hand, luck might have it that the agent visits the only malicious host in the network just prior to returning to the originating host—putting the previous results of all hosts visited at risk to some form of deletion. Loureiro and his associates [311] A form of protection that uses the integrity checks for a collection of unordered objects, referred to here as set hashing, was proposed by.

The hashing method in this approach makes use of a strong (Sophie Germain) prime  $p$ , where  $p = 2q + 1$  and  $q$  is prime also. Given  $g$  as generator for cyclic group  $Z_p^*$ , for every  $x$  in the set  $\{1$

..  $(q-3)/2$  the following relationship holds true:  $g' = g^{2x+1} \bmod p$  is also a generator for  $Z_p^*$ . This particular construction gives nice features for a set of  $n$  elements and an associated set hash derived with this property: security, commutativity, cancellation, and a computation complexity with only  $2n$  multiplications,  $n$  additions, and one exponentiation. Figure 127 depicts the agent data collection mechanism of Loureiro *et al.* [311] which integrates set hashing.

The agent in Figure 127 visits three hosts, two of which it visits more than once. The first part of the protocol (1a, 1b, 1c in Figure 127) illustrates that the agent owner and every host in the itinerary must establish a pre-existing shared secret before agent dispatch. This of course limits the free-roaming nature of an agent to some degree, but lifts the dependence for a PKI. In the depiction,  $Host_1$  shares key  $K_{A1}$  with the originating  $Host_A$ . We can establish shared keys over public channels readily via operations such as the Diffie-Hellman key exchange.

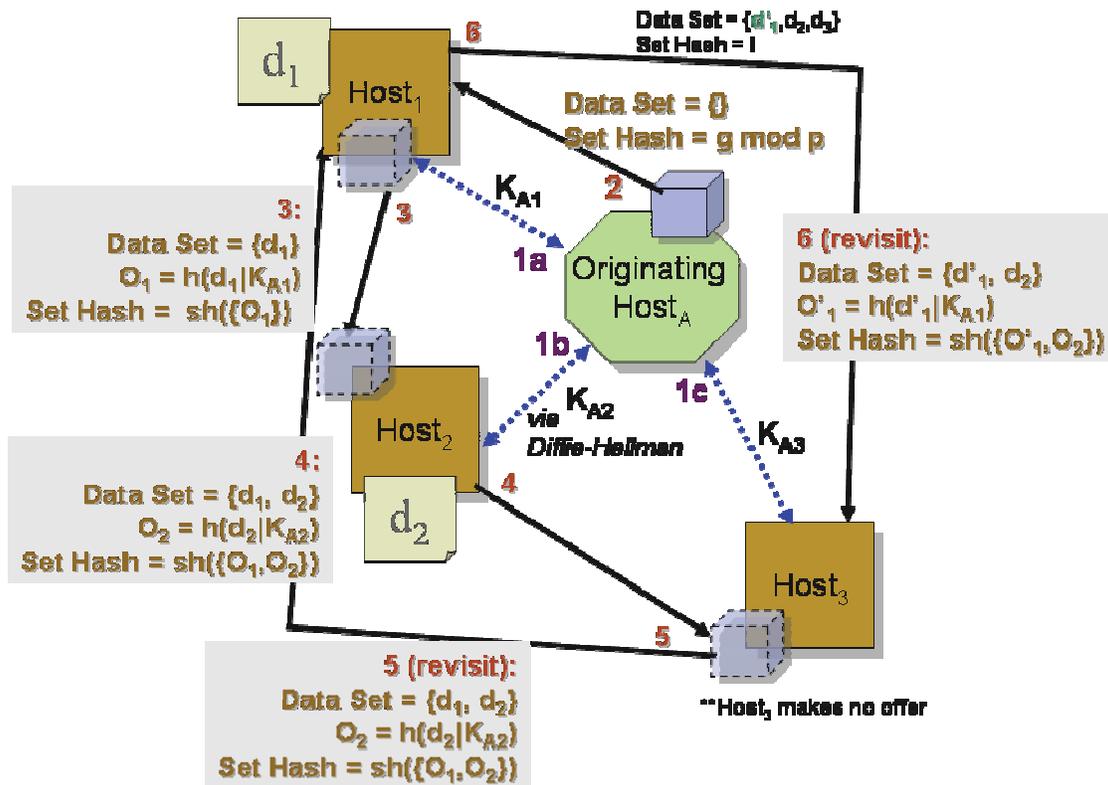


Figure 127: Set Hashing Data Collection

As the agent collects data at each remote host and places them into an embedded data set, it computes a set hash based on the generator  $g$  mentioned above. Each element of the set hash function is created by hashing the data result computed at the remote host ( $d_1$  for first visit of the agent to  $host_1$  and  $d'_1$  the second time the agent visits in Figure 127) and the shared key ( $K_{A1}$ ,  $K_{A2}$ ,  $K_{A3}$ , etc.). The set hash is the data integrity mechanism that inserts set elements to be added in an unordered sequence—and the mathematical properties of the generator allow a host to cancel out a previous data item and reinsert a new one. Set hashing allows an agent to visit a host more than once (a feature *prevented* in [31] and [144]) and allows a host to make no bid at all. It provides a cryptographically based method for data integrity, insertion resilience, and truncation resilience. Strengths include support for *randomized* agent itineraries, no requirement for a public key infrastructure, and the reducibility of the set hashing data collection algorithm to the security of solving a discrete logarithm in a finite field.

The secure data collection protocol does not provide data confidentiality in its basic protocol but supports it as an add-on feature—leaving open applications where results in cleartext might be advantageous or needed. Suen [312] poses the idea of combining set hashing with Vigna's

data collection protocol. This approach allows an agent to carry the execution trace (Vigna) along with the encapsulated set hash of data results so that the application can lift the requirement for the trusted third party to process execution receipts. As with most all data protection mechanisms, set hashing is unfortunately all or nothing: if verification fails when the agent returns to the originating host then the application owner must throw all agent results away.

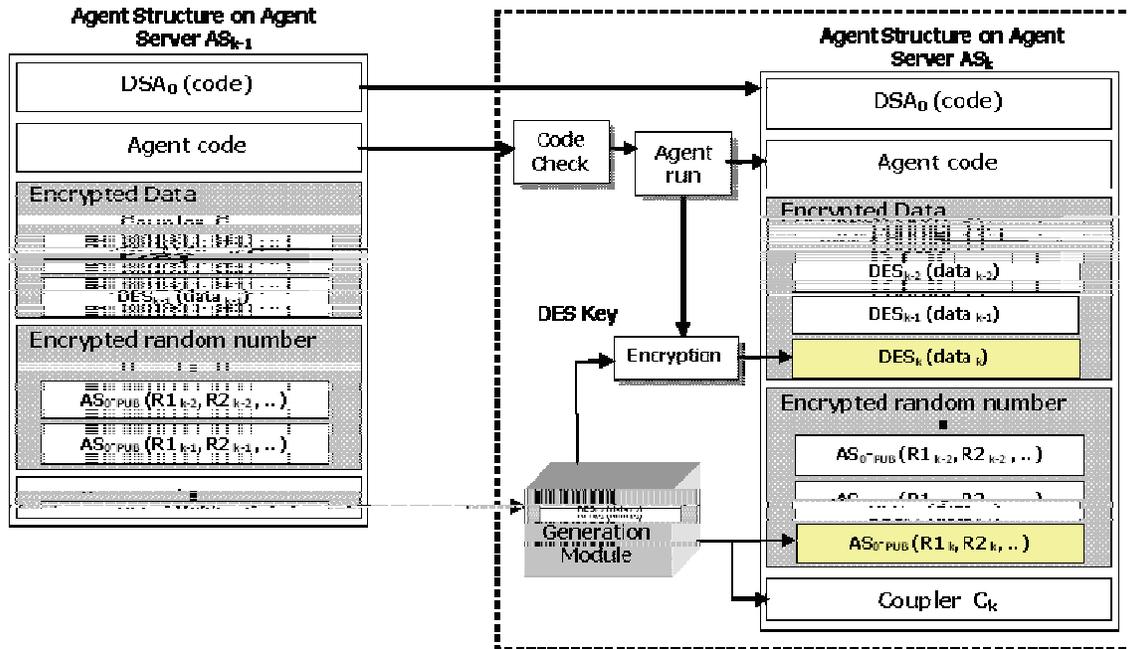


Figure 128: One Time Key Generation Scheme

#### A.4.10 One Time Key Generation (OKGS)

Park *et al.* in [313] proposed another one-time key generation method (OKGS) that integrates a hash function and coupler—resulting in chains of previous results with current results. Figure 128 depicts interaction of an agent with a host. The scheme requires both time-stamping and public key cryptography in order to work. The agent owner signs static code with a digital signature (seen as  $DSA_0(\text{code})$  in Figure 128) and each agent framework verifies the signature of the code before allowing execution. Each agent framework in the itinerary signs its own results with a *private* digital encryption scheme (DES) key as well.

In OKGS, executing hosts use a unidirectional key chain for encryption using the DES algorithm (a symmetric key scheme). The secret key (seen as agent key  $S_k$  in Figure 129) is created by hashing the XOR of a random nonce ( $R1_k$ ) and previous agent coupler ( $C_k$  in Figure 128 and Figure 129). The current executing host XORs the agent key ( $A_k$ ) with another random nonce ( $R2_k$ ) and hashes it to produce the coupler that will be sent to the next host. In order to recover the shared secret key and decrypt agent data state encrypted with it, the originating host must have access to each set of nonces generated by each remote host. For this purpose, the executing host uses the public key of the originating host to encrypt the two random numbers, a timestamp, and a signature of the data with timestamp included.

OKGS provides data confidentiality and forward data integrity, though it does not support publicly verifiable detection. The digital signature of each host's data encrypted with the public key of the sending host is similar to other data encapsulation techniques. By using time stamping, the scheme eliminates replay attacks by setting keys valid for specified transaction periods. OKGS also provides truncation resilience in the face of colluding malicious hosts. Unfortunately,

large overhead is involved in setting up a time stamping service and the data grows linearly with the agent as in other cases where digital signatures are used.

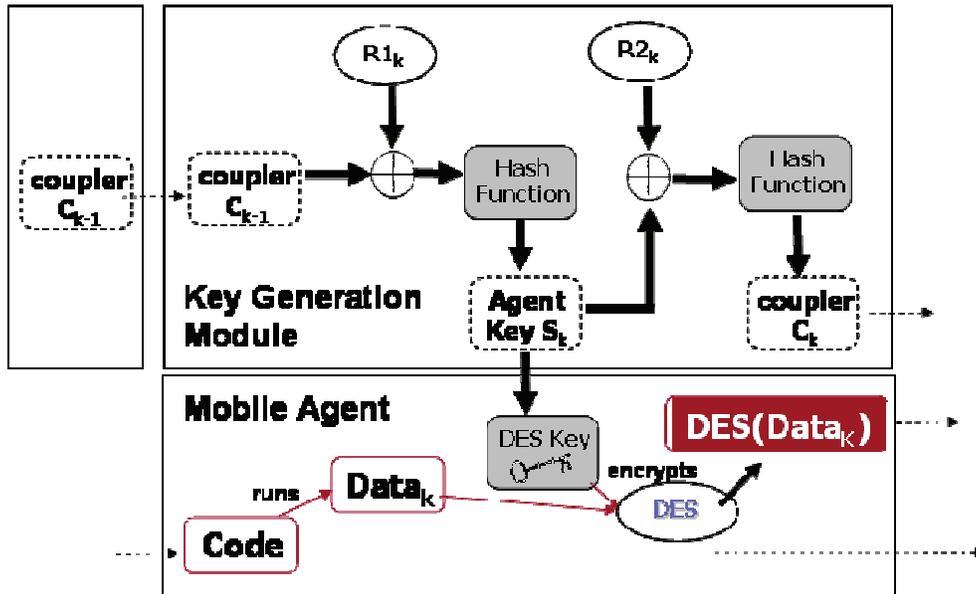


Figure 129: Key Generation Module

#### A.4.11 Configurable Protection

Maggi and Sisto in [107] build upon the data integrity mechanisms proposed by Karjoth *et al.* [144], Corradi *et al.* [308], and Karnik and Tripathi [284] to develop a configurable protocol that can be adapted to varying levels of data integrity properties, depending on the application. Building on the work of Roth [108, 112], they correct the problem of many data protection mechanisms that do not bind the identity of the agent code to its state. An abstract model captures the message exchange sequence between two agent servers in the itinerary of an agent. In the abstract case, an agent consists of two basic elements: static agent code and a set of encapsulated data elements. Table 25 summarizes the abstract model that defines a configurable data protection scheme.

Table 25: Abstract Data Protection Model

Model Part	Definition
$\Pi_o$ (Static Agent Code, <i>Timestamp</i> , $t$ , <i>Signature key of agent owner</i> , $K_o^{-1}$ )	$\{\Pi, t\}K_o^{-1}$
Identity of remote host	$h_n$
Data result computed at remote host $n$	$d_n$
$M_n$ (Encapsulated data computed at host $n$ )	$D_n    C_n$
Set of data items carried by agent	$\{M_o, M_1, \dots, M_n\}$
$h_n \rightarrow h_{n+1}$ (Transmission of agent from host $n$ to host $n+1$ )	$\Pi_o, \{M_o, M_1, \dots, M_n\}$
$D_n$ ( <i>Protected</i> data result or offer gathered at host $n$ — some function of data computed and host identity)	$D(d_n, h_n)$
$C_n$ (Chained protection configuration parameter)	$C(h_n, d_n, C_{n-1}, i_{n+1})$

Four configurable protocols achieve detection of truncation and implement a binding mechanism for agent data state. Each case extends the abstract protocol to incorporate varying security features. As Table 25 illuminates, an agent migrates from host to host carrying a set of encapsulated data results  $\{M_1, M_2, \dots, M_n\}$  and a signed copy of its static code tied to a timestamp. We avoid interleaving attacks in the scheme by using the timestamp associated with the agent's

static code. The system protects the data via some encryption method and links it to the identity of the executing host (either through hashing or nonce). The parameter  $D_n$  is therefore some function of the data and host identity that computed that data. A chained protection configuration parameter  $C_n$  is used to link the data collected ( $d_n$ ) at a given host ( $h_n$ ) to the previous protection check and the identity of the next host to be visited. Each protocol describe by Maggi and Sisto [107] (named *MS1* to *MS4*) defines both configuration parameters ( $D_n$  and  $C_n$ ) to achieve certain levels of protection. In order to achieve data authenticity, for example, a real protocol extends the abstract model by choosing the manner in which we configure  $D_n$  and  $C_n$ . Since authenticity deals only with verifying the identity of a piece of information, no encryption of the data is required. For this level of protection, the following behavior is established:

$$\begin{aligned} D(d_0, h_0) &= \epsilon \\ D(d_n, h_n) &= d_n \\ C(h_0, d_0, C_0, i_1) &= \epsilon \\ C(h_n, d_n, C_{n-1}, i_{n+1}) &= (d_n, \Pi_0)K_n^{-1} \end{aligned}$$

Each intermediate host signs its data result along with the agent kernel using the executing host's signature key. Left open is how the signature key is established (shared secret, asymmetric cryptography, hash-based methods, etc.). The originating host must only be able to verify the signature. This protocol (referred to as *MS1*) supports not only data authenticity but non-repudiability because neither side (agent owner or remote host) can disavow that the data result came from execution of the agent's code. As Maggi and Sisto point out, different agent applications require different levels of security—some that require privacy, some needing publicly verifiable qualities, and some needing protection from multiple colluding hosts. The configurable protocol allows a wide variety of freedom in implementing specific mechanism such as hash algorithm, keying methods, and agent identification. Table 13 summarizes the security properties achieved by the protocols offered.

**Table 13: Protection protocols MS1-MS4**

Protocol	Properties
<b>MS1</b>	Data authenticity Data non-repudiability Trusted data integrity (fixed itinerary)
<b>MS2</b>	MS1 features Data confidentiality Forward privacy Origin confidentiality Trusted data integrity Strong data forward integrity (fixed itinerary)
<b>MS3</b>	MS2 features Strong data forward integrity (weak free-roaming agent)
<b>MS4</b>	Weak trusted data integrity (full free-roaming agent) Data confidentiality (can be added)

#### A.4.12 Modified Set Authentication Code

Research continues to improve upon data protection mechanisms proposed over the last decade. Gunupudi and Tate [223] for example extend the set hash code proposed by Loureiro *et al.* [230]. In the original protocol, the agent owner distributes a shared secret to each host in the agent itinerary. Loureiro *et al.* suggest the use of Diffie-Hellman key exchange for this purpose, while Gunupudi and Tate point out several shortcomings. A random host can be in the agent itinerary, but the establishment of a shared secret requires the agent owner to remain on line for the entire duration of the agent's lifetime. Diffie-Hellman in its basic form can also be susceptible to man-in-the-middle attacks.

In order to correct this deficiency, Gunupudi and Tate [223] develop a modified set authentication code that allows the remote host to generate the secret key and then encapsulate it as part of the agent's data state using encryption under the originator's public key. If updates are required, a host has the choice of reusing its secret key or generating a completely new one. The method has similarities to OKGS where secret key generation takes place at each host in the itinerary. Once the application creates the secret key, each host makes normal use of the message authentication code to produce an encapsulated offer. The owner can apply the set integrity verification function as in the case of the original set hash code approach.

A malicious host cannot modify any data set in the agent without detection because only the original executing host knows the secret key and the each host encrypts their encapsulated items within the agent. A malicious host can still insert fake data elements with a completely different secret key and then encrypt it using the agent owner's public key—thus masquerading as other hosts and introducing bogus data such as bids. As in the original set hash code (by Loureiro *et al.* [230]), the scheme is not resilient to truncation attacks from two or more colluding hosts.

#### A.4.13 Chained IP Protocol

We discuss chained IP mechanisms and other itinerary protection schemes in Section A.3.17, but mention them here for completeness. In the chained IP approach, the owner leaves the route information unencrypted and appendable by each subsequent host in the agent's route. The approach establishes the strength of the chaining mechanism by the identities of the previous host, the current host, and the next host in the itinerary. However, the chaining is susceptible to truncation attacks when one or more malicious hosts collude.

#### A.4.14 ElGamal Encryption

Cartrysse and van der Lubbe [294] propose several mechanisms to protect agents including support for confidential agent communications, task confidentiality, and support for agent signature functions. In terms of providing and collecting confidential data, the authors state that a necessary condition for an encryption algorithm that protects data across multiple parties must have an *E-E-D* property, defined as a chained sequence of asymmetric key encryption and decryption. In particular, when an agent carries confidential data that a remote host will use (similar to anonymous itineraries), other intermediate hosts must not be able to read that information. Figure 130 depicts an encryption scheme that helps support such privacy.

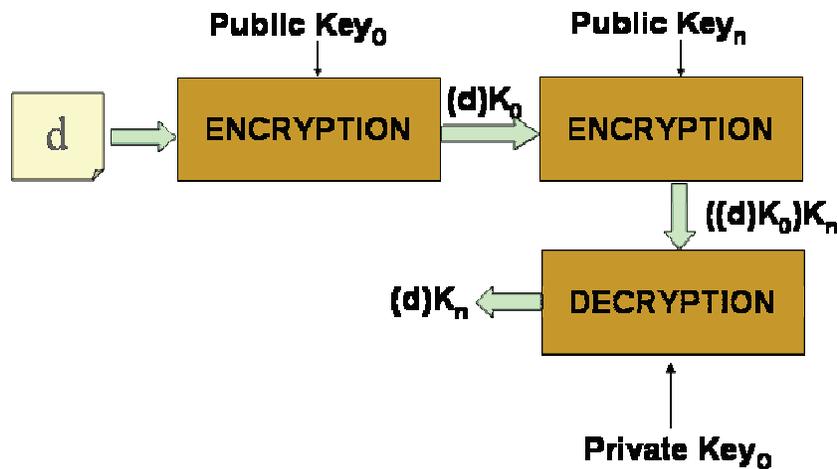


Figure 130: E-E-D Property

A needed property for data confidentiality as the agent traverses a network is for the information to be encrypted by the agent owner (using the  $PK_0$  in Figure 130) and also the public

key of the server where the data is intended ( $PK_n$  in Figure 130). Normally, the encrypted sequence  $((d)K_0)K_n$  can only be decrypted to get the data item  $d$  by using the private signature key of the remote host ( $K_n^{-1}$ ) first. The E-E-D encryption would allow decryption to get back the data item encrypted with public key of the remote host without first using the private key of the remote host  $n$ . The ElGamal encryption scheme [314] is the only suitable algorithm that can support such a property and is proposed by Cartryse and van der Lubbe as a means to protect private data in an agent.

As a result, an agent owner embeds data intended for a specific remote host so that only that host can decrypt and used the data given the agent's private decryption key. We can use different secret decryption keys for different data embedded within the agent to strengthen the scheme. The protocol may divulge data unless hosts follow a correct ordering of operations—requiring a trusted third party to be present to enforce ordering.

#### A.4.15 Protocol Evaluation

To conclude discussion of data protection mechanisms, we note mobile agent protocols suffer from the same weaknesses as normal network protocols that attempt to exchange information or establish trust in a secure way. Roth [112] reiterates that mobile agent protocols certainly have undetected flaws in them and suggests development of a formal way to analyze these protocols. Little research points out the vulnerability of mobile agent protocols—especially those designed to provide malicious host protection—than has been asserted about protocols themselves. A promising area of research is development of both formal methods and verification techniques applicable to existing or newly developed mobile agent architectures.

Traditional network attack techniques combine execution of legal operations in multiple execution strands. Roth [108, 112] describes weaknesses and flaws of protocols proposed by Karjoth *et al.* [144], Karjoth [287], Corradi *et al.* [308] and Karnik and Tripathi [284]. In particular, Roth illustrates how *cut & paste* techniques allow a malicious host to take a portion of one agent and embed it in another agent for use in a parallel attack sessions. Roth also shows how malicious hosts use the *oracle exploit* attack to create their own agent with the purpose of getting other hosts to perform cryptographic operations on them (following the rules of the protocol), thereby decrypting sensitive information and removing privacy mechanisms.

Roth concludes that in every protocol analyzed, non-malicious hosts act as a potential oracle that performs encryptions, signatures, and decryptions on behalf of malicious hosts. He also observes that interleaving attacks pose great problems to posed agent security mechanisms because just digitally signing a mobile agent's static code does not prove ownership or authenticity of the agent. The possibility exists that an adversary can use agent code if the agent state is not bound to the static code: such vulnerabilities exist in several primary data encapsulation protocols. Maggi and Sisto [107] solve this deficiency and take steps to use an agent kernel. Roth [109] also gives suggestions for improvement of each protocol to shore up security vulnerabilities.

A tension exists among the many configuration properties in mobile agent data protection. For example, some systems would be more secure if they provided anonymity (origin confidentiality) of the individual hosts because collusion would be harder [144]. For non free-roaming agents, applications can detect truncation and substitution attacks much easier because the owner knows the list of visited hosts absolutely. Trusted intermediaries provide beneficial security mechanisms, but we general view their presence as reducing interoperability. Knoll *et al.* [79] conclude that reliance on a certification authority can introduce complexity when an agent moves to different realms. The more secure a mobile agent system is, the more proprietary solutions become.

## A.5 Secure Multi-Party Computations

This section provides background material to results presented in Chapter 3 regarding integration of SMC protocols with multiple mobile agent architectures.

Cryptographers have for some time sought how to perform a group function when there are a number of mutually or partially distrusting participants to the operation. Yao's blind millionaire problem [315] is often cited as an early formulation for the two-party case where a function  $z = f(x,y)$  is computed between Alice and Bob—without leaking any information about Alice's input  $x$  or Bob's input  $y$  other than what can be deduced from  $z$  itself. Goldreich and his colleagues in [316] extend secure computation to  $n$  parties—defined in the general case as a publicly available function  $f$  that takes  $n$  private inputs and returns  $n$  private outputs:  $f(x_1, x_2, x_3, \dots, x_n) = (y_1, y_2, \dots, y_n)$ . In some instances, all parties learn the same function output such that  $y_1=y_2=\dots=y_n$ , making the output publicly known.

Secure computation is referred to synonymously as secure multi-party computation (SMC), *secure function evaluation* (SFE) or secure circuit evaluation. Various contributions from active research in the field can be found in [105, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333]. In terms of practical use, [334] summarize privacy-preserving, real-world applications that can be represented as an SMC problem such as database query, scientific computations, intrusion detection, statistical analysis, geometric computations, and data mining. Malkhi *et al.* develop a full programmatic implementation of a two-party secure function evaluator called Fairplay [335] that uses oblivious transfer [319, 320, 321, 336] and one-pass Boolean circuits [315, 316, 327, 322].

SMC protocols typically involve several rounds of interaction between parties and assume different types of communication channels including, for example, private channels between every two parties [317, 318], a broadcast channel [322, 337], and broadcast subsets among player triples [333]. In terms of security, we can reduce the correctness and privacy of *any* protocol to the evaluation of a secure function protocol [322]. In the ideal setting, all parties to an SMC can send their inputs via a secure private channel to a trusted third party that computes the group function and return results fairly.

A primary security result concludes that any function we can compute with polynomial resources (communication and computation) we transform and compute in a *secure* manner using polynomial resources [330]. Corruption in multi-party computations deal either with an *honest-but-curious* (*semi-honest*) adversary that passively reads information from corrupted parties or an *active* (*malicious*) adversary that exerts full control over parties. Privacy of inputs is at issue in passive attacks while correctness of the outputs is more at issue in active attacks. Goldreich concludes in [328] that we can force two parties acting maliciously to behave in a semi-honest manner or else catch them violating the security of the computation.

For any arbitrary function in the presence of an active adversary, we can securely accomplish the computation as long as less than 1/2 of the players have not been corrupted [316]. The unconditional security results found by [317, 318] state that computations can occur as long as less than 1/3 of the players have been corrupted and secure channels exist in both directions between any two players. When we introduce broadcast channels, unconditional security is possible for the computation as long as less than 1/2 of the players are corrupt. Cachin and colleagues [105] reiterate that computation between two unbounded parties with "full information" is not securely possible for arbitrary functions and only limited to trivial functions  $g$  where  $g(x,y)$  reveals  $y$ . We consider these results significant when applying multi-party computations in the realm of mobile agents.

### A.5.1 Evaluation Techniques and Primitives

Yao first posed the idea that we can model a function  $f$  and securely execute it as a Boolean circuit [315, 338] using a protocol known as *secure circuit evaluation*. The circuit can be "scrambled" in a way to secure host inputs and compute the group output. Abadi and Feigenbaum posed a two-player scheme in [321] where one player runs a secret program for

another player who has a secret input. We find other techniques for circuit construction including multi-party cases in works such as [316, 317, 318, 322, 327, 339]. Once we represent the function  $f$  as a circuit, parties must run a protocol to evaluate every gate in the circuit.

Secure primitives in the circuit evaluation process include tools such as oblivious transfer (OT) [340, 341, 342] and verifiable secret sharing (VSS). Work by [333] has sought to find minimal complete primitives to accomplish SMC and characterize security and efficiency of such tools beyond the two-party case. To accomplish secure circuit evaluation, we must encrypt (garble) the original wire signals for both inputs and outputs of the circuit so that the actual wire signals used by the parties no longer have their same semantic meaning. In order to translate inputs and outputs to their true semantic meaning, data is exchanged between two parties in an oblivious manner—typically *1-of-2 OT* [336].

While OT deals with privacy in circuit-based SMC, we can address cheating by verifiable secret sharing which allows a “dealer” to distribute shares of a piece of data among different parties [278, 299]. Normally, parties in the computation must commit to their bits (which become garbled for purposes of evaluation) before they are used. However, no other party could tell whether the scrambled bits actually represent the real semantic meaning of a party’s input. By using sharing techniques, parties give shares of their inputs so that other parties can detect any attempt to alter a commitment. We find discussions on data re-sharing data to prevent a super adversary with control over some set of parties from gathering enough shares to compromise a system in [343, 344].

Not all protocols are as secure as their authors envision. For example, a vulnerability is described in [345] in the constant round circuit evaluation of [322] where private information is leaked when gates within a circuit share a common input wire. Efficiency is also a major issue and much work has been done to improve protocols over time [346, 347, 348, 329, 332]. We note other more efficient methods than Boolean circuits exist, for instance, to represent  $f$  such as permutation branching programs, algebraic circuits, low degree and randomizing polynomials, and matrices over large fields [330]. Hurt and Meier [329] present a protocol that is secure for computing an  $n$ -party function with  $m$  multiplication gates in the presence of less than  $1/3$  actively corrupted players with complexity  $O(mn^2)$ .

Typically, SMC protocols have been adapted for synchronous networks and suffer from computational or communicational complexity too high for use in the real world. Mobile agents operate in asynchronous environments and we therefore must consider other factors before we can apply SMC techniques successfully. Work by Canetti and others [349, 350, 351, 352] have created frameworks characterizing the composable nature of security properties for different protocols operating across asynchronous networks—thus addressing the need to model realistic network environments. As [344, 353] suggest, protocols need to integrate timeouts with distributed computations for asynchronous networks (that model the Internet) and the environment for mobile agent applications.

### **A.5.2 Single Round Computations and Agent Integration**

Mobile agents exhibit three unique properties that make using SMC protocols difficult: autonomy, mobility, and disconnected operations. All of the protocols mentioned thus far have relied on the exchange of information between parties in multiple rounds, including the originator of a function. Agents require non-interactive protocols because the originator of a function may be offline during the actual computation. Autonomy stipulates that the agent does not return home after the first host and can visit some set of known or unknown hosts. Mobility without the help of a trusted third party and minimal communication among parties is a primary goal of agent security schemes.

As discussed in [289, 354], there are two ways to view single round computations between two parties in contrast to traditional secure function evaluation: computing with encrypted data and computing with encrypted functions. CEF represents the mobile agent transaction scheme best and we can extend it easily to a multiple host approach. Sander and Tschudin posed one of the first non-interactive CEF approaches for mobile code execution based on homomorphic encryption in [25]. Researchers have extended their results to include any function implemented by logarithmic-size circuits in [292]. Cachin *et al.* in [105] developed a non-interactive protocol

(which we will refer to as the CCKM scheme) that evaluates all polynomial time functions via the use of a scrambled circuits and oblivious transfer. Table 26 summarizes the nuances between CED, CEF, and normal secure function evaluation.

We derive several important results from [105]:

- 1) for unbounded passive adversary, any function computable by a polynomial-size circuit can be computed securely;
- 2) for a bounded active adversary, any function computable by a polynomial-size circuit can be computed securely, given a public-key framework; and
- 3) any function computable by a polynomial-sized circuit has a one-round secure computation in the model.

We summarize non-interactive SMC approaches and results by [105] and present them in Table 27. The CCKM methodology is foundational to several approaches for mobile agent security based on secure multi-party computation.

Mobile agent applications have brought a practical relevance to development of secure, efficient cryptographic protocol schemes. Cryptographers have stated the goal of SMC as guaranteeing the correctness of a function and the privacy of results among the parties. In mobile code systems, similar notions exist: malicious hosts can spy on the code, state, or results of mobile agents that they execute. Hosts can gain unfair advantages by altering the normal sequence of execution, replaying agent computations using different inputs, or altering the state information present in the agent. Software-only approaches to mobile agent security that are secure, efficient, and removing need for trusted relationships have been the holy grail in the research field for quite some time. There are two primary approaches to integrating SMC protocols with mobile agents: use single agents that implement single-round non-interactive protocols or use multiple agents that execute multi-round SMC protocols in coalition schemes. We discuss approaches and issues with the former next.

**Table 26: Methods of single-round secure function evaluation**

Type	Computation
<b>Computing w/ Encrypted Data (CED)</b>	Alice has input $x$ while Bob holds function $f(\cdot)$ . Alice sends an encrypted version of $x$ to Bob who computes and sends the result back to Alice in a single round of interaction. Alice decrypts the result to get $f(x)$ while Bob does not learn $x$ .
<b>Computing w/ Encrypted Functions (CEF)</b>	Alice holds the function $f(\cdot)$ while Bob holds input $y$ . In one-round, Alice sends to Bob an encrypted version of $f(\cdot)$ who provides his input $y$ . Alice receives back and decrypts Bob's result to learn $f(y)$ but does not learn $y$ while Bob does not learn $f(\cdot)$
<b>Secure Function Evaluation (SFE)</b>	Alice and Bob have private inputs to the function $f(x,y)$ . Alice and Bob jointly compute the function $f(x,y)$ in one round of computation. Alice learns only the result (and nothing more) while Bob learns neither the result nor Alice's private input.

**Table 27: Pertinent Results for Non-Interactive Secure Multi-party Computations**

Contribution	
[317]	Trivial functions where $A$ and $B$ are unbounded
[25]	Functions represented as polynomials, $B$ is bounded
[292]	Functions computable by logarithmic-depth circuits, $B$ is bounded
[105]	Functions computable by polynomial-depth circuits, only $A$ is bounded or both $A/B$ are bounded
[317]	Trivial functions where $A$ and $B$ are unbounded

### A.5.3 Non-Interactive SMC Approaches

To formulate a single-round secure multi-party computation, the following formal notation from [27, 105] is used: an agent originator  $O$  embodies a private function executed by a set of hosts  $H_1, \dots, H_l$ . Two functions— $g_j(\cdot)$  and  $h_j(\cdot)$ —describe the computation of an agent in terms of a state  $x \in X$  and a host input  $z \in Z$ . Figure 131 illustrates the interaction of an agent which is captured by a multi-party computation. The state update function  $g_j$  takes a current state (brought by an agent from the previous host) and the local host input and produces a next state  $x_j$ . The host output function  $h_j$ , illustrated in Figure 132, takes the current state (brought by the agent from the previous host) and its own local input to produce its own local output.

In the CCKM protocol, once we represent the agent computation as a Boolean circuit and encrypted, we require translation tables to map actual signals to scrambled signals. We base the circuit encoding on Yao's two-party SFE protocol in [315]. In order to know what signals to use for their local input, a host performs oblivious transfer with the originator to get a set of scrambled signals, and the originator does not know which signals the other party chooses. We thus establish the following security properties: 1) the originator has privacy of the function; 2) each host has privacy in respect to their local input. The CCKM approach allows for autonomy in the agent path by creating an encrypted circuit that is a cascade of sub-circuits. Each host in the route of an agent's path would receive an encrypted circuit on which their input is applied. However, the CCKM protocol did not address the ability for each host to use the "unencrypted" local output of the agent because the application owner is the only party that can evaluate the encrypted result.

#### Computation: State Update Function

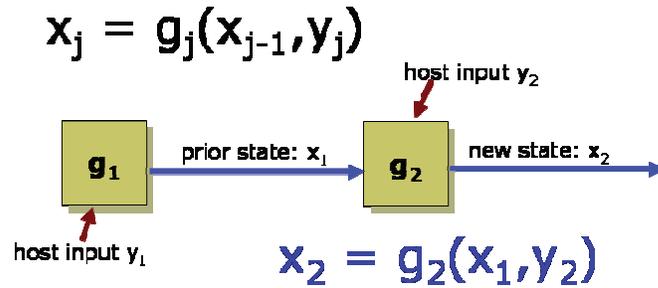


Figure 131: State Update Function

#### Computation: Host Output Function

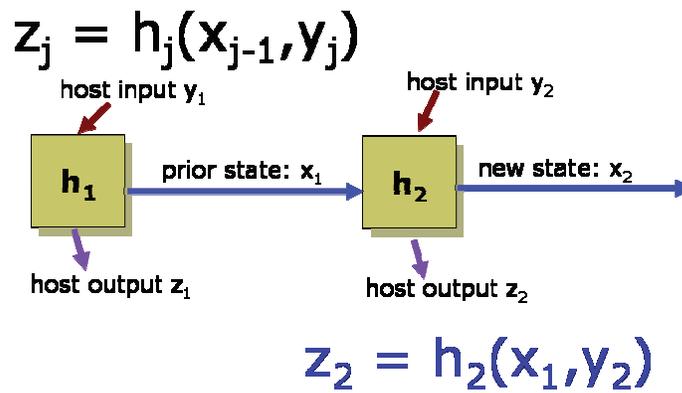
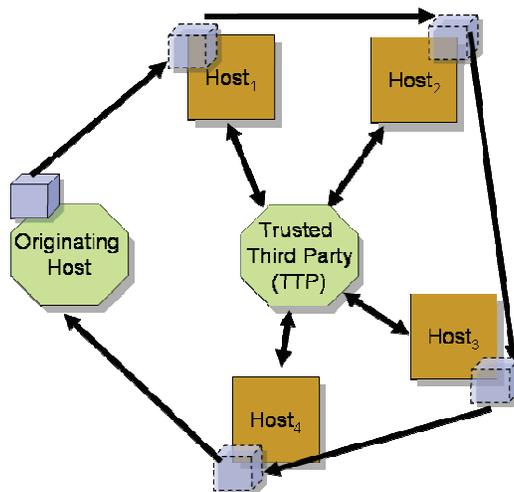


Figure 132: Host Output Function

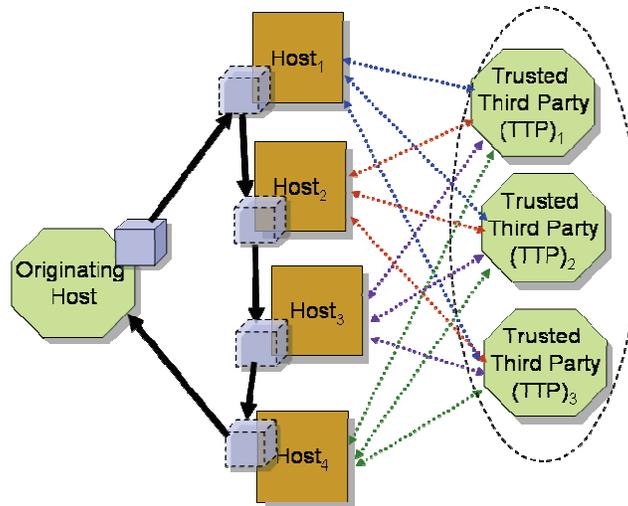
Extending the CCKM approach further, Algesheimer *et al.* [27] produced a non-interactive protocol (which we refer to as the ACCK protocol) similar to the trusted hardware of [110] that would allow for secure decryption of host output when CEF is used. The ACCK scheme, illustrated in Figure 133, makes use of a trusted generic computation service roughly equivalent to the trust we place in a public key infrastructure. To decrypt the output of the agent at the local host, we encrypt the mappings for the semantics of the signals with the public key of the generic service. Each host accomplishes oblivious transfer with the generic service (instead of the originator who may be offline) to decrypt the signals for the output. By using a secure middleman, the ACCK protocol hides inputs, outputs, and computations of all hosts from the originator as well as any other host visited by the agent. The main assumption is that this trusted third party (TTP) does not collude with the originator or with any host, but as proposed would offer a generically secure service for *any* application.



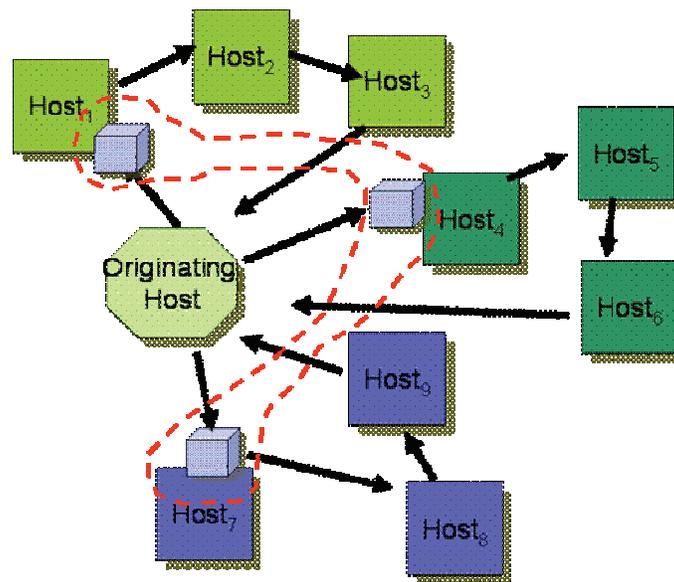
**Figure 133: ACCK Protocol w/ Generic Computation Service**

There have been two extensions proposed to the ACCK protocol that target replacement of the TTP in some form. Zhong and Yang in [299] introduce a cryptographic primitive called verifiable distributed oblivious-transfer (which we refer to as the VDOT protocol) and Tate and Xu in [288] introduce a multi-agent approach utilizing their oblivious threshold decryption (which we refer to as OTD). Figure 134 shows a notional arrangement of parties in the VDOT scheme while Figure 135 shows a notional arrangement of parties in the OTD approach. In the VDOT protocol, we divide mobile agent computations into security-sensitive and non-security-sensitive portions. Under the scheme, we transform code that requires integrity or confidentiality into a garbled Boolean circuit. Instead of interactions with one trusted third party, which has weaknesses involving the corruption of a single server to the detriment of the entire system, VDOT uses several trusted third party servers to replicate the functionality of the TTP. VDOT guarantees with high probability the correctness of receiver's output, enforcement of the code and state privacy, protection from coalitions of malicious hosts and malicious TTPs, and the verification that servers give correct decryption of host signals.

Distribution of trust among a group of servers strengthens the original ACCK protocol and forces a group of servers that hold shares of the decryption to perform the table lookup for circuit signals. The VDOT protocol is general purpose in the sense that each host need only provide an interpreter for garbled circuits. By using distributed oblivious transfer, trusted third parties act as a proxy for agent owners and provide translation tables for host inputs without being able to discover host inputs themselves. Obvious disadvantages to the approach are increased communication complexity (which the authors contend is negligible in practice) and the complexity of breaking a program into security sensitive portions represented by a Boolean circuit.



**Figure 134: Verifiable Distributed Oblivious Transfer Protocol**



**Figure 135: Oblivious Threshold Decryption Protocol**

The OTD protocol of [288] is similar in some regards to VDOT but actually eliminates the trusted third-party requirement altogether. As a primary distinction, their approach relies on multiple agents visiting disjoint sets of the possible host pool. Each of these agents act in a threshold manner (similar to VDOT) to decrypt the encrypted signals for a given host input without relying on the TTP. While the ACCK secure computation service overcame the interaction requirement of Yao’s encrypted circuit evaluation—a limiting factor in the mobile code paradigm—OTD replaces this by means of cryptographic operations and multiple agents that cooperate together. Multiple agents must agree before decryption of the host’s input signals can occur and this in turn prevents cheating by keeping a list of hosts that have already decrypted a signal. Agents return to the originating host where the application decrypts all circuit results and combines them to produce the function result. The security in this method rests on the security of Yao’s secure circuit evaluation, the security of the 1-out-of-2 oblivious transfer, and the strength of threshold cryptography. However, this protocol does not support free-roaming agents and requires knowledge of the set of hosts an agent will visit.

Algesheimer *et al.* in [27] state the ACCK protocol does not require foreknowledge of the agent's path or the hosts that the agent will visit. Their approach upholds the disconnected and autonomous nature of a mobile agent. However, it is not clear whether the number of host,  $\ell$ , must be specified or known beforehand. The OTD and VDOT extensions both assume a known number of hosts or subsets of hosts in order to design the circuit representation of the group function—thus limiting a true free-roaming dynamic itinerary. Though single-round non-interactive protocols reduce the communication overhead for SMC, message sizes increase proportionally, regardless of input or output size. Tate and Xu, for example, state that it roughly takes 9 kilo-bytes to encrypt 32 bits of secret data [288] under this scheme. Zhong and Yang mitigate overhead by keeping security sensitive portions separate from normal programmatic requirements. Using multi-round SMC offers another approach to accomplishing secure transactions with mobile agents, which we analyze now.

### A.5.4 Multi-Round SMC Approaches

Secure multiparty computations have a tradeoff between trust and efficiency. Neven *et al.* [355] were one of the first to envision the use of agents to implement SMC and reduce the overhead of the communication itself. Figure 136 summarizes four different approaches to integrating agents with hosts to accomplish SMC. Figure 136-a illustrates the ideal world where agents carry host inputs to a trusted third party and a protocol is evaluated without the expense of network broadcasts or bidirectional secure channels. In the context of the TTP, all parties can evaluate the protocol and we assume the TTP to behave honestly with respect to host inputs.

We see the most secure but least efficient method in Figure 136-b: here hosts simply become the execution environment and setup a multi-party protocol evaluation. In this case, we face both the computational and communicational complexity inherent in the chosen protocol and only high-speed links (represented by the dotted lines) make such protocols practical. We illustrate single-round approaches discussed in the previous section in Figure 136-c where an agent embodies the circuit for secure evaluation and each host provides private input as the agent migrates. In [355], a hybrid solution as depicted in Figure 136-d involves high-speed communication links present between one or more hosts. Participants in the  $n$ -party protocol send agents carrying their private inputs to one of these intermediate TTPs who then efficiently and securely evaluate the function according to the rules of the protocol.

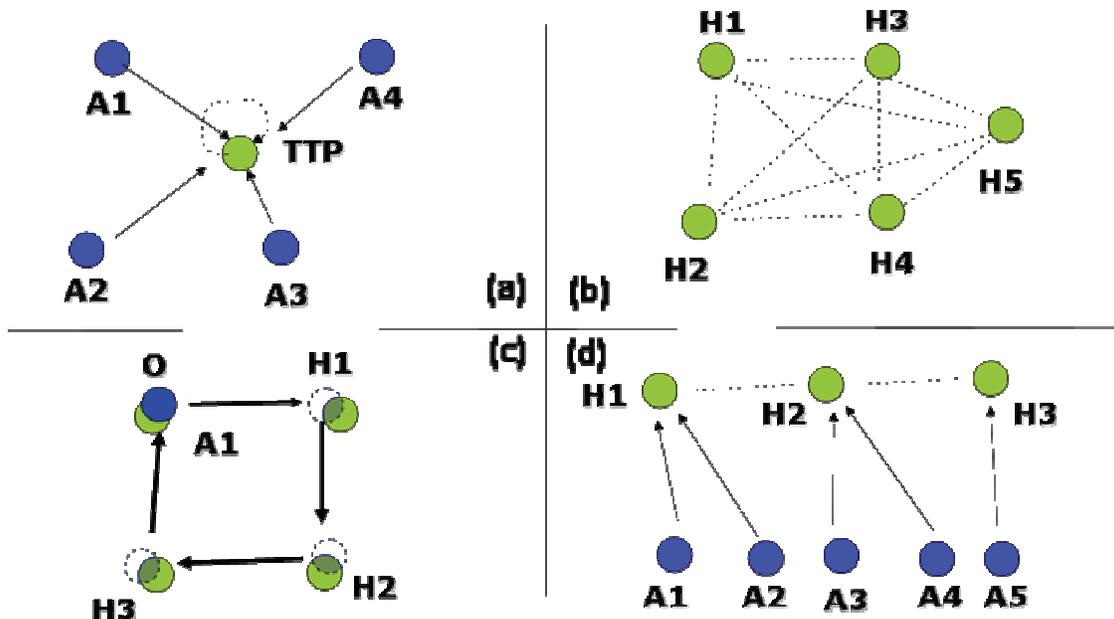
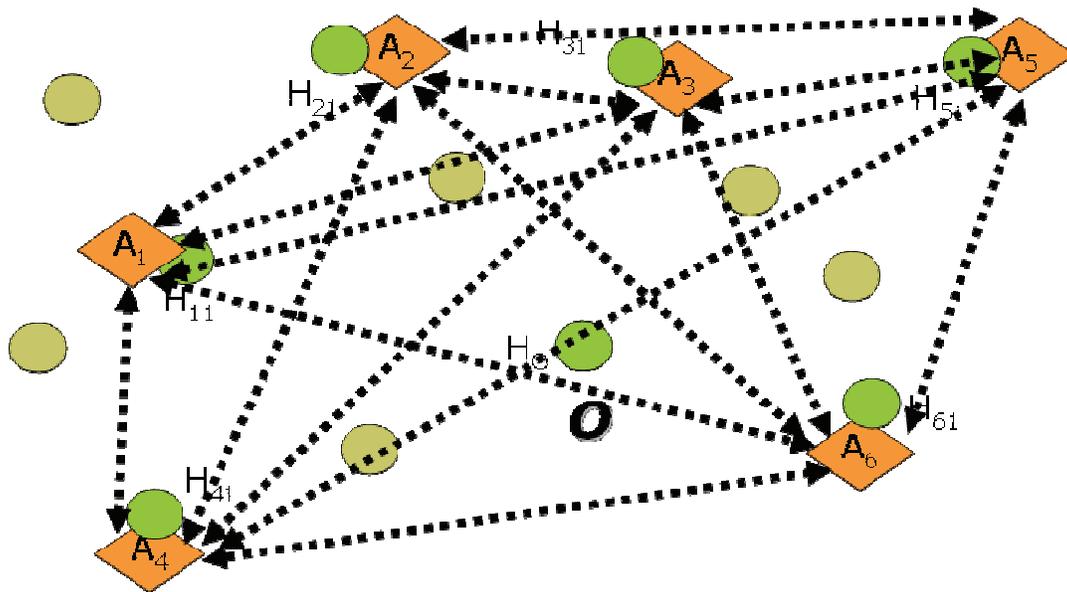


Figure 136: Agent Approaches to SMC

In the realm of mobile agents, as with many real world applications, it is preferable not to rely on a trusted third party and just perform an SMC among the parties of a function. Endsuleit and Mie utilize a group of multiple agents to support such an approach in [344]. In their model, they deploy multiple agents carrying the same realized circuit to remote hosts where parties evaluate rounds of the secure protocols. Figure 137 illustrates that agents are located on some set of hosts and implement *multi-agent* computations based on some underlying SMC protocol. In [344] the authors assume the extensive use of a broadcast channel and suggest the protocol of [317] with an implementation of secret sharing from [278]. In [353], follow-on work suggests the use of more efficient protocols such as those of [329].

In such multiple agent schemes, we can use *any* SMC protocol as long as it meets the composable security properties defined by Canetti [352]. In order to adapt the Canetti model, which assume stationary parties, “slices” are defined [344,353] as periods where a set of  $n$  different hosts executes a community of  $n$  agents with no migrations during that period. Hosts use resharing of data shares via the Ostravsky and Yung method [343] to overcome the adverse affects of migration where malicious hosts can use acquired shares over time to compromise security. The system supports self-repairing code and threshold agreement of computations, as long as up to  $1/3$  of the community (agents or hosts) remains uncompromised. Security results follow because Canetti establishes proof of a secure protocol for  $n$  parties computing a joint function in the presence of an active adversary corrupting up to some  $k$  limited servers. By using such agents to implement a redundantly shared global state of computation and coordinate activity, we can implement a wide variety of SMC protocols. However, as with any multi-round solution, the communication complexity is extremely high and the originator must know a priori which hosts will be part of the computation.



**Figure 137: Multi-Agent Secure Computation**

In [356], another software-only scheme is presented that implements multiple agents acting in a threshold manner similar to [288,344,353]. However, there approach does not suppose the presence of collusions among hosts or rely necessarily on multiparty protocols. Their approach, which is termed Remote Distribution Scheme or RDS, depends on a set of agents that replicate and share a transaction set. In RDS, the system also assumes a publicly known algorithm—a feature that does not necessarily correspond to the mobile agent setting where code privacy is required or application owners implement CEF.

## A.6 Multi-Agent Architectures

This section provides background material to results presented in Chapter 3.

Multi-agent security is distinctly different from mobile agent security in terms of issues. Multi-agent systems that use mobility explicitly, however, can provide security solutions not possible in either the strictly static or the strictly mobile sense. In considering malicious host protection, several solutions [31,277,280] rely on the use of multiple agents. Wang and Tan [357] for example use parallel dispatch of multiple agents to provide route protection. We can accomplish multi-party computations via multiple deployed agents in the network, but we distinguish here the use of multiple *identical* agents (those that perform the same task or have identical static code) from using multiple agents instantiated from multiple different agent *classes*. Both approaches can achieve certain desired security properties with fault tolerance and we discuss the latter form of multiple agents in this section.

Kotzanikolaou *et al.* [227] postulated two separate classes of agents used to conduct secure electronic transactions. A master agent in this approach controls one or more slave agents, with the master agent remaining static at the originating host. The system dispatches slave agents individually to single remote servers in a single-hop fashion, gathering bids or relevant information. Slave agents in this case are not empowered to perform a transaction but instead are specialized to find agreement on pricing or return commitments. The master agent serves the role of information filtering to pick the best offer and possibly dispatching a unique transaction agent to finish the purchase process. We can also use the multi-agent architecture to determine which mobile agents have been victims of malicious behavior. The Sanctuary architecture [31] uses various agent classes to perform different types of services. Agent owners can also create agent groups that define agent sets running and migrating in synchronous activities. Each group is composed of task-specific agents that perform different services such as data query, indexing, and authorization. Merwe and Solms [11] implement trade agents as distributed objects that communicate remotely and accomplish group tasks.

Multiple agents can support group agreement or threshold security mechanisms [259, 288]. Groups of agents can perform undetachable threshold signatures [301], group key establishment, group agreement for purchase, and buddy checking of results [280, 358]. Borselius *et al.* [259] for instance use a subset of multiply deployed agents to complete and authorize a transaction. Their threshold scheme assumes that at least one of many possible transactions will receive enough votes, even in the presence of a minority of malicious hosts. Baumann and Radouniklis [100] set forth an architectural model for e-commerce that uses groups of mobile agents arranged as initiators, administrators, receivers of information, coordinators, and normal members. Their approach is set in the context of a mall-based application that supports group communication, synchronization for task, and termination. Finally, the use of multiple agents for security enhanced mobility is mentioned more frequently in current surveys [21, 48, 64, 359, 360].

Multiple replicated agents and multiple instances of multiple classes of agent can offer greater security advantages when malicious hosts are at work together. In certain application environments, it may also be possible to allow communities or groups of non-compromised hosts to offer protection from one or more possibly malicious hosts. In a military or corporate intranet environment, for example, hosts can be trusted when they are part of the community. Such systems assume adversaries compromise hosts operating normally (from the inside) or capture hosts at some point. Group host protection is similar to using trusted third parties or honest nodes and expands the idea of specialized node acting as base-stations in trusted environments. Guan and his colleagues [361, 362] use specialized hosts to perform security operations within their community in what they term a "police" model. Certain hosts act as policemen to monitor the health or status of agents during their transit.

Cheng and Wei [309] enhance the security of a publicly verifiable signature scheme by introducing two hosts working in partnership. In their approach, the preceding host in an agent itinerary is required to produce counter-signature before sending an agent to the next host in the path. Zhou *et al.* [310] extend this method further by requiring two hosts in agreement with the

current host. Yokoo and Suzuki [293] utilize a set of *trusted* hosts to perform secure multi-party computations with the help of multiple agents. In certain scenarios, even the auctioneer may not be trustworthy and the use of multiple servers working together can help alleviate concerns of data compromise. Societies of cooperating agents *and* hosts may offer the highest possible security mechanisms for mobile agent architectures in the future, especially in the face of malicious collusions.

## A.7 Trust Infrastructures

This section provides background material to results presented in Chapter 4.

We review here foundational concepts appropriate to the realm of trust in security related decision making. Defining *trust* is as precarious as defining the term *agent*—and though researchers do not agree on either term they do discern the importance of both concepts in framing research. We define trust loosely in a previous work [38] as the expectation of behavior among parties and classify several different infrastructures for trust in dynamic networks. Gambetta [363] defines trust as a subjective probability that is non-reflexive, changing, and context driven. We note also that trust is not necessarily Boolean, does not necessarily have to capture human intuition, and can involve third parties to facilitate certain issues. Trust can be transitive and therefore delegated [364] or can be acquired by direct observation [365,366].

### A.7.1 Trust Management in Distributed Environments

The trust management problem, as defined in [36], seeks to formulate a coherent framework for defining policies, actions, relationships, and credentials in terms of trust. Trust management systems such as [35,36,365,366,367] support specification, acquisition, revocation, degradation, and evolution of trust according to some model. As we point out in [38], challenges in these systems revolve around observing trust-related actions and then translating those observations into a decision. In both mobile agents systems and dynamic networks, we must make trust decisions before parties build trust relationships. The overlap of trust models with the development and implementation of mobile agent security frameworks is a key to future support of pervasive computing scenarios. We consider next the adaptation of trust mechanisms specifically to the mobile agent paradigm.

Mobile agent applications and the idealized vision of a global computing scenario share many common characteristics with distributed trust models for dynamic networks: a large number of autonomous parties, principles can have no prior relationship, a trusted or centralized computing base may not exist, virtual anonymity of principles may exist, different administration domains, and hosts have different capabilities for transmission, mobility, and computational power. Grandison and Sloman in [37] point out that trust cannot be hard-coded in applications that require decentralized control in large-scale heterogeneous networks. Mobile agents particularly need to separate the application purpose from the trust management framework if they are to scale well in such environments. Because there is a large commonality between dynamic networks and mobile agent applications, we can readily apply many proposals for defining trust infrastructures in ad hoc networks to mobile agents. Kagal *et al.* [368] suggested the addition of trust to enhance security for mobile computing scenarios and defined trust management activities as those defined by [36]: developing security policies, assigning credentials, checking credentials against policy requirements, and delegating trust to other parties.

Cahil *et al.* in [365] expound the research goals of Secure Environments for Collaboration among Ubiquitous Roaming Entities (*SECURE*)—a project focused primarily on building trust infrastructures for large ad hoc wireless networks. *SECURE* utilizes the use of both risk and trust to determine whether interaction can occur in a scenario where billions of possible collaborators may exist. In their authorization framework, a principle uses trust to decide an interaction based on the risk involved. Carbone *et al.* expound the trust model for *SECURE* in [367] and provide a formal way to reason about the trust relationship among principles. We find interest in the *SECURE* model because they use a trust interval from 0 to 1, which reflects a measure of uncertainty. The trust level in *SECURE* is thus a range with some upper bound—which sets the maximum amount of trust within some factor of unknowing involved.

As Cahil and his colleagues also point out, traditional trust management systems delegate permissions using certificates (credentials) that reduce to very static decisions that do not scale well or allow change over time. They also point out as we do in [38] that trust decisions in

pervasive scenarios *should* come instead from trust information based on two sources: personal observations (previous interactions) and recommendations from other parties we trust (transitive or delegated trust). However, in [365] there is no link provided between security requirements, trust, and application goals for mobile agents specifically—a goal we set forth to accomplish in this paper.

In a similar vein, Capra in [366] defines a formal model for *hTrust*—a mobile network trust framework that defines formation, dissemination, and evolution of trust. Capra reviews several trust management frameworks that have the following limitations: they are suited for centralized servers, they have too high of a computational overhead (for mobile devices), they lack dynamic trust evolution, they lack details about local policy usage, and they lack subjective reasoning capabilities. *hTrust* remedies these shortcomings and incorporates a human notion of trust directly into the framework. *hTrust* models a range of trust values between principles so that parties can distinguish a lack of evidence or knowledge decision from a trust-based decision that reflects specific distrust towards another party. The trust data model also incorporates the notion of time so that relationships degrade when not kept current. The system uses recommendations when a principle has no past history or to partially rely on or trust a third-party assessment.

Just as in human interactions, *hTrust* captures the notion that we favor recommendations from people who gave us good recommendations in the past and likewise reject or downgrade advice from those who have disappointed us in the past. Finally, a key aspect of this model is its incorporation of both a social context (the environment of principles arranged in a network by which recommendations can be used) and a transactional context (the network of services which are supplied by parties in the system). While *hTrust* provides a generic framework for mobile application trust expression, it does not directly deal with mobile agent specific security requirements or attempt to link mechanisms for security to trust levels or the agent lifecycle.

### **A.7.2 Trust and Mobile Agents**

There has also been much work specifically focused on security evaluation and trust expression for *mobile agent* systems. Karjoth *et al.* were one of the first to describe a security model for Aglets—a specific type of mobile agents [369]. The Aglets model includes a set of principles with distinct responsibilities and defines security policies that give access to local resources. The notion of a policy database and user preferences are also included in the model to govern interactions of aglets that have unspecified or unknown security properties. However, the Aglets model does not address host-to-agent malicious interactions or incorporate the notion of trust levels or dynamic trust determination.

Other security management systems designed specifically for mobile agents suffer from the same limitations and focus on *malicious code* protection. Jansen poses a privilege management system for agents in [370] that uses traditional certificate-based policy framework. In this model, host-based policy specification enforces security compared to agent-based attribute certificates. When these policies merge during agent execution, the system determines the security context of the agent. Other works such as [371] describe reconfigurable policy support and surveys such as [64] summarize issues and status with policy-based security implementation. Again, such mechanisms tend to not scale well, tend towards static security interactions, and do not model trust levels for specific security requirements.

Antonopoulos and his colleagues in [372] develop a general-purpose access control mechanism that is distributed in nature and specifically focused on mobile agents. This approach comes closer to expressing trust relationships among principles, but the approach relies on access control lists and preventing malicious code activity as a fundamental basis. Kagal *et al.* extend their delegation trust model to mobile agent scenarios in [41] and address FIPA-specific weaknesses for securing the agent management system and directory facilitator services. Although they address how multiple agents can establish trust when they are previously unknown, their focus is primarily on authentication and they do not consider mobile-specific security requirements and trust expression.

Tan and Moreau [39] develop a more comprehensive model of trust and belief specifically for *mobile agent*-based systems based upon the distributed authentication mechanisms posed by Yahalom *et al.* [35]. They found their trust derivation model on the similarities between distributed

authentication in public key infrastructures and mobile agent trust. The Tan/Moreau framework is limited and simple, however, because it only incorporates the notion of trust associated with using the extended execution tracing security mechanism [33] and does not account for generic security mechanisms or requirements. Their unique contribution in the area is one of the only works that link security mechanisms with

Borrell and Robles with several different colleagues have done significant work to incorporate trust in mobile agent system development [40, 77, 373, 374, 375, 376, 377]. In [40], they present a model that defines trust relationships expressed in the *MARISM-A* mobile agent platform [378]. Although initial work in the area by Robles and his colleagues assumed a static trust expression among agents [373, 374, 375], *MARISM-A* now uses trust relationships that are defined among entities and actions in a mobile agent transaction. We follow similarly with a formal model that defines trust relationships between entities and associates actions and attributes to each relationship. The *MARISM-A* model uses trust relationships to define decisions that permit, obligate, designate, or prohibit certain actions within a certain scope of a mobile agent program. We take a similar approach as Robles by associating agent security mechanisms with trust relationships and by classifying their role as a deterrent, prevention, or correction. In addition to certain similarities with the *MARISM-A* approach, we expound more fully in our model the relationship between the agent application, principles, security mechanisms, trust levels, and trust determination.

Ametller along with Robles and Ortega-Ruiz take the notion of policy-based security mechanisms even further in [377]. In this scheme, agents select the appropriate security mechanism to use by means of a security layer that exists within the agent itself—a layer untied to the underlying agent execution platform. Security layers of the agent interact with hosts to determine which mechanism the agent uses based on predetermined criteria. Assuming a decryption interface and public key infrastructure are in place, the solution gives a novel security addition implemented in JADE and provides a flexible approach to agent security. We build also upon this approach by linking in our model and methodology security requirements for agent applications that system supports via security mechanisms—all of which tie into evaluation and evolution of trust relationships among principles in the mobile environment.

As many authors point out, no *one* security mechanism addresses every security requirement. The use of security mechanisms in fact may establish a certain level of trust (or lack thereof) in the face of certain environmental assumptions about malicious parties. An application level view of security that we propose would bring together a process for selecting a combination of techniques to achieve certain trust levels within a mobile agent system. Even when using mechanisms that establish pure trust (such as tamperproof hardware), other assumptions must be taken into account to make sure security is guaranteed. Trusted hardware or multi-agent secure cryptographic protocols may be warranted or even feasible given certain application environment factors. When such mechanisms are not available, the system can demand lower trust levels and require a more policy-driven approach to make dynamic decisions about agent execution.

We use models in many cases to help describe a particular set of relationships in more precise terms. Models in the security sense do several things such as: help test a particular security policy in terms of completeness or consistency, help document security policies, help conceptualize or design an implantation, or verify that an implementation fulfills a set of requirements [379]. We now present our trust framework for considering mobile agent applications and describe a model for viewing principles, actions, and trust relations in the mobile agent environment.

## APPENDIX B

### TRUST MODEL ELABORATION

In this appendix, we illustrate the descriptive capability of the framework described in Chapter 4. We begin first with a scenario that illustrates the nature of trust related decisions in mobile agent application. Figure 138 depicts the three types of hosts (dispatching, executing, and trusted) and the agent itself.

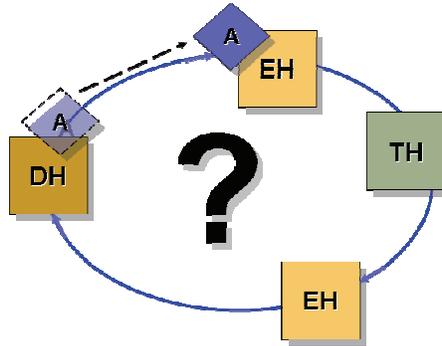


Figure 138: Trust Decisions and Principles in a Mobile Agent Setting

#### B.1 Trust Scenario with Mobility and Agency

We can liken the mobile agent trust problem to a person walking up to you in your office and handing you a floppy disk. They say to you, “I have heard that you like sharing security research information with others. Please run the file on this disk called ‘virus.exe’ on your computer for me and then I’ll walk it to the next security researcher on my list of contacts when you are done. Don’t worry about the output—the program keeps its state every time it executes.” You, acting as the (executing) host, must ask a few probing questions of course. Before considering whether to run the file or not, the mental process begins by first considering the trust you have in the person (application owner) who brought (dispatched) the file (agent). This represents the application owner in the mobile agent environment. Do you know them personally? If not, do you know other people who know them? If not, how will you assess their trustworthiness? Will you ask them for a resume or a list of references first?

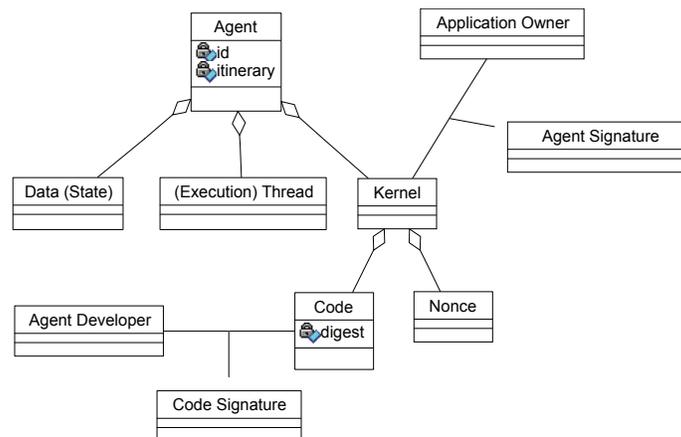
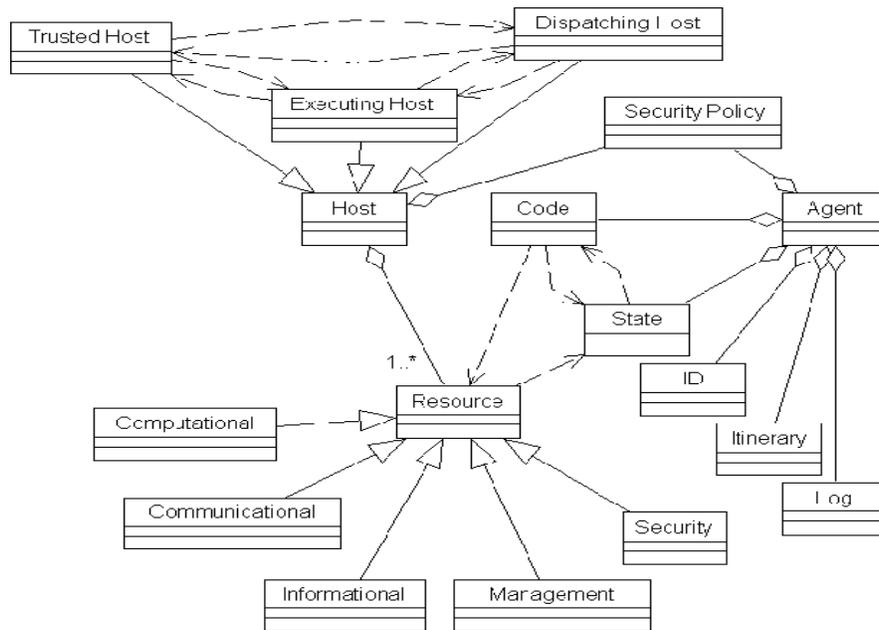


Figure 139: Agent, Application Owner, and Agent Developer Trust Relation

If you do not know the person personally, you may want to see their credentials so that when you ask for references you are sure you are talking about the right person. In the mobile agent context, the application owner determines the *agent signature*: the binding between the dispatcher (sender) of the agent program and the identity of that particular *instance* of the agent code execution. When parties are completely unknown, you might determine trust based on successful accomplishment of testing suites or rapid small transactions. If you know them to be untrustworthy or assume that anyone asking to run files on your computer from a floppy is untrustworthy by default, then the emphatic answer will be “No, I will not run that program.”

If you the person or the person has a history of dealings with you, you may (after verifying they do not have an evil twin) entertain the idea of running ‘virus.exe’ on your computer. Assuming you pass this first hurdle, the next mental process turns to the question of the code itself. Where did ‘virus.exe’ come from? Who authored it? Did the dispatcher author the code or has the developer authored other programs that proved trustworthy? Is the code developer identified with hacker groups or malware or do they work for an official or approved organization? In the mobile agent paradigm, there are limited ways to ascertain such trust relationships. If the person indicates there is a software clearing-house that has reviewed and tested his code for safety and malicious behavior (passing with certified safety properties), we may allay some of our fears. This is equivalent to using a trusted third party to assess trustfulness of the agent code or the code developer a priori. If the person says that the organization’s code development group authored the code, a user may also place implicit trust in in the code. A *code signature* links the developer with the code itself (even though many application owners may use that code in multiple ways and instances over time). However, some other method of proving or verifying code safety may be needed. Figure 139 depicts the relationships between signatures and principles to the mobile agent transaction.



**Figure 140: Host-Agent Trust Relations**

If you know the person carrying the disk (let us say they are a good friend) *and* you know they authored the software *and* you trust them enough to execute the file, the remaining questions might focus on the nature and purpose of ‘virus.exe and the chain of custody of the courier’. What does ‘virus.exe’ do? If the algorithm is private, the person may say “I can’t tell you, you just have to trust me”—which at that point you determine that you still won’t run the program even for a good friend. If the algorithm is public, then you may be relieved to find out that ‘virus.exe’ is a

statistical data gathering program that queries your anti-virus software to see when the last time you updated your virus protection was and how many viruses have been detected in the last month.

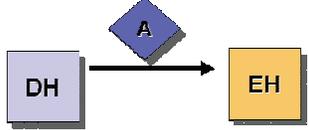
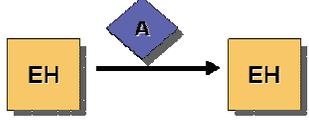
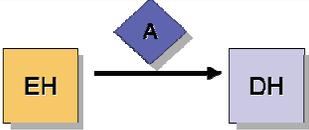
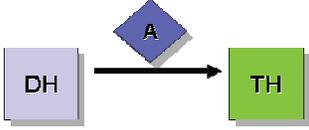
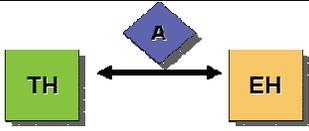
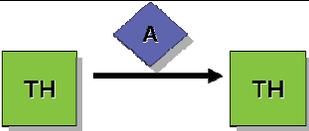
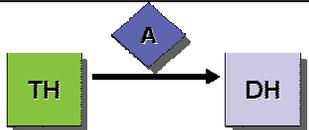
**Table 28: Security Requirements with Associated Detection/Prevention Mechanisms**

		Detection	Prevention
CP	agent code privacy		TPH secure multi-party computations encrypted functions obfuscation multi-agent systems
CI	agent code integrity	code signatures clone detection	code signatures
CF	agent code safety	state appraisal	sandboxing proof carrying code
CA	agent code authenticity	code signatures	code signatures
IP	agent itinerary privacy		anonymous/onion routing bidirectional dispatch
II	agent itinerary integrity	itinerary recording replication and voting	
SI	agent state integrity	state appraisal detection objects protective assertions executing tracing reference states intermediate result protection state transition verification group host operations	TPH intermediate result protection secure multi-party computations encrypted functions environmental key generation undetachable signatures
SP	agent state privacy	sliding encryption	TPH secure multi-party computations encrypted functions obfuscation phoning home multi-agent systems
AA	agent authenticity		signatures
AZ	agent authorization		signatures
AN	agent non-repudiation		signatures
AV	agent availability	time-limited execution w/ trusted third party phoning home	
AY	agent anonymity		trusted third party
HA	host authenticity	host signatures	trusted third party
HN	host non-repudiation	host signatures	trusted third party
HP	host data privacy		secure multi-party computations trusted third party
HY	host anonymity		trusted third party
HV	host availability	state appraisal path histories	sandboxing safe interpreters policy management
HI	host integrity	path histories	sandboxing safe interpreters proof carrying code

You may also want to know who has executed the agent before you. Even though you can verify that the agent code remains unaltered and verify identity of the code developer or the dispatching agent, the floppy contains a mutable state updated by some list of previous parties. Depending on anonymity requirements, you can observe the routing slip of the courier and you notice that the floppy was in the possession of Dr. Evil at some point. How can you be sure that the state of the program and the code for virus.exe do not together cause a buffer overflow attack on your machine when you execute it? If the program were collecting statistics, how could the application owner be sure that Dr. Evil has not altered results in order to skew security decision making? These questions all reflect our desire to ascertain code and data integrity. Figure 140 depicts the host and agent relationships that capture these nuances in our model. We now

describe these trust decision in terms of our model elements, security requirements, and trust levels.

**Table 29: Trust Decisions in Mobile Agent Applications**

Principles Involved	Decision
	<p><b>Agent Dispatch</b> (Dispatching Host → Executing Host) Dispatching host (DH) launches agent A to the first executing host in the itinerary (EH). The agent is an instance of code created by a code developer (CD) and the application owner (AO) is associated with the DH by some managerial role. DH makes itinerary choice decision, EH makes execution decision. DH makes policy decision regarding EH based on security requirements / EH makes policy decision regarding A based on security requirements: EH→CD: CA, CF; EH→DH/AO:HA, HN; A/AO/DH→ EH: CP,Ci,SI,SP,IP,II,HY,HA,HN; EH→ A: AA,AZ,AN,HV,HI,IP,HY,HP</p>
	<p><b>Agent Migration</b> (Executing Host → Executing Host) Executing host (EH) sends agent A to the next executing host in the itinerary (EH). DH makes itinerary choice decision, EH makes execution decision. DH makes policy decision regarding EH based on security requirements / EH makes policy decision regarding A based on security requirements: EH→CD: CA, CF; EH→DH/AO:HA, HN; EH→EH:HA, HN; A/AO/DH→ EH: CP,Ci,SI,SP,IP,II,HY,HA,HN; EH→ A: AA,AZ,AN,HV,HI,IP,HY</p>
	<p><b>Agent Termination</b> (Executing Host → Dispatching Host) The last executing host in the itinerary (EH) sends the agent back to the original dispatching host (DH) where the agent terminates. DH makes itinerary choice decision, EH makes execution decision. DH makes policy decision regarding EH based on security requirements / EH makes policy decision regarding A based on security requirements: EH→CD: CA, CF; EH→DH/AO:HA, HN; A/AO/DH→ EH: CP,Ci,SI,SP,IP,II,HY,HA,HN; EH→ A: AA,AZ,AN,HV,HI,IP,HY</p>
	<p><b>Trusted Dispatch</b> (Dispatching Host → Trusted Host) Dispatching host (DH) launches agent A to a trusted host in the itinerary (TH). The agent is an instance of code created by a code developer (CD) and the application owner (AO) is associated with the DH in some managerial role. The trusted host performs a security function on behalf of the agent and alters trust relations as a result: TH→CD: CA, CF; TH→DH/AO:HA, HN; TH→ A: AA,AZ,AN,HV,HI,IP,HY</p>
	<p><b>Trusted Migration</b> (Executing Host → Trusted Host, Trusted Host → Execution Host) Executing host (EH) sends agent A to a trusted host in the itinerary (TH) or a trusted host migrates the agent to the next EH. Executing hosts receiving agents from trusted hosts may have different security requirements (i.e., implicit trust), and may therefore allow agent execution based on the relationship.</p>
	<p><b>Trusted Transfer</b> (Trusted Host → Trusted Host) Trusted hosts migrate agents from one to the other. Implicit trust, based on their status, is assumed.</p>
	<p><b>Trusted Termination</b> (Trusted Host → Dispatching Host) A trusted host sends the agent back to the original dispatching host (DH) where the agent terminates.</p>

Agent migration presents us with the basis trust decision among parties involved in the mobile agent application. Table 28 provides a (non-exhaustive) summary of detection and prevention mechanisms associated with various security requirements. The integration of any particular mechanism of course varies with complexity and cost. Each principle in the system evaluates trust tuples and computes a requirement for both agent and host security enforcement. Table 29 outlines typical policy decisions supported by the trust framework and a populated trust database.

The trust qualifiers such as foreknowledge, freshness, and level dictate the requirement for either no mechanism or a weak/strong mechanism.

## B.2 Modelling Agent Applications

We introduce here sample mobile agent applications that elaborate the trust framework described in Chapter 4. We use these to illustrate the nuances that exist among principles to include hosts, agents, and entities. These elaborations capture the notions for multiple agent instances, running the same agent in multiple application instances, multiple agent interactions, and trusted host interactions. We define the mobile agent application as follows:

<p><b>Set H of possible hosts:</b>  <math>\{h_0, h_1, h_2, \dots\}</math></p> <p><b>Set A of uniquely identifiable agents:</b>  <math>\{a_0, a_1, \dots\}</math></p> <p><b>Set Y of uniquely identifiable agent states</b></p> <p><b>Agent: (kernel, id, data, itinerary, thread, policy)</b></p>	<p><math>h_x \in \{D \cup E \cup T\}</math>  D: only 1 dispatching host  E: all possible executing hosts  T: all possible trusted hosts</p> <p>Multiple agents with the same static code, multiple agents with different static code, a single agent.</p> <p>For every <math>a_i \in A</math> there is a corresponding set of states <math>Y_i</math>. After execution on <math>k</math> hosts in itinerary:  <math>Y_i = \{Y_{i1}, Y_{i2}, \dots, Y_{ik}\}</math></p> <p>kernel = (code, nonce)<sub>SIG</sub>    agent signature  id = hash (kernel)  itinerary = <math>\{h_0, h_1, h_2, \dots, h_k\}</math>    ordered/unordered  code <math>\in \{c_0, c_1, c_2, \dots\}</math>    static/immutable</p>
---	--

Given the ability to identify agent states from execution run to execution run, we can uniquely identify different runs of the same agent codebase. This means, for example, that an application owner that executes the same agent code twice executes two unique applications—not the same one. Every agent therefore creates a unique application instance when executed. Figure 141 illustrates a basic application where a code developer creates agent code ( $c_1$ ) used by two different application owners. Application owners use the digest and signature of the code to create their own unique agent signature ( $a_{k1}, a_{k2}$ ) based on their own random nonce ( $r_1, r_2$ ). Each application owner has a unique itinerary which includes one or more trusted hosts and different executing hosts.

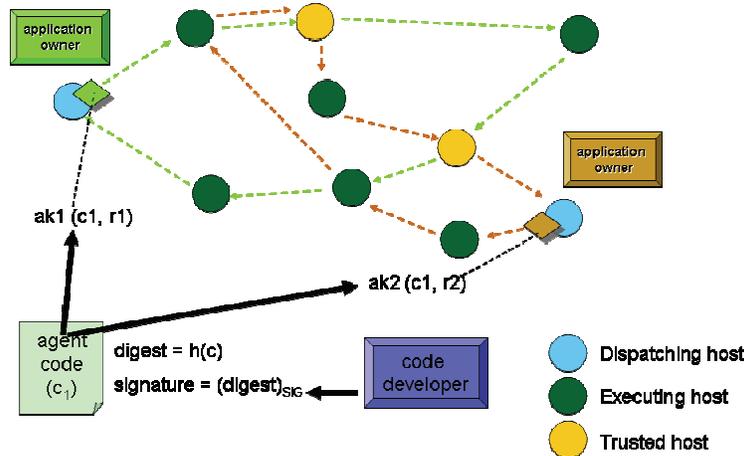
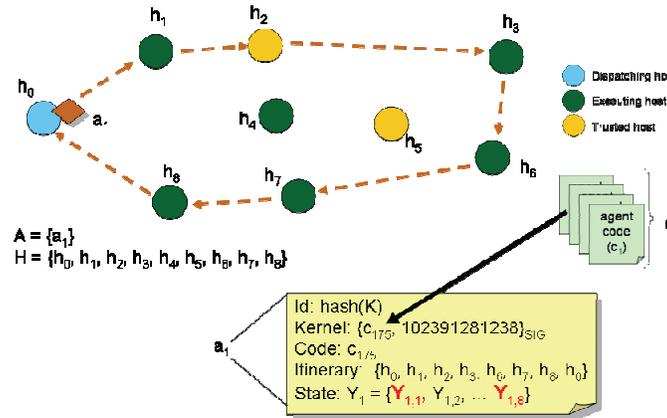


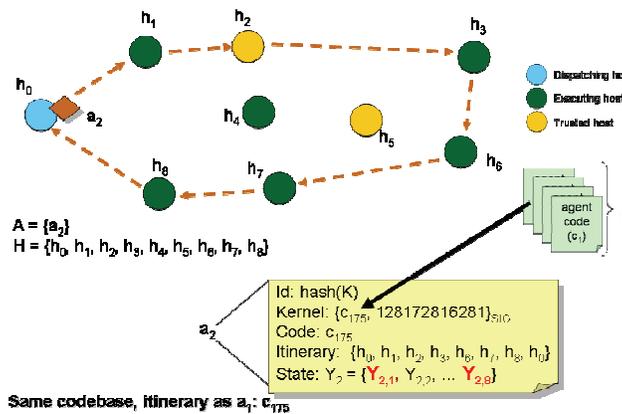
Figure 141: Host and Entity Interactions in Agent Application

Figure 142 illustrates a single agent instance ( $a_1$ ), a single dispatching host  $h_0$ , and illustrates the itinerary among a set of hosts ( $h_1 \dots h_8$ ). The state set  $Y_1$  indicates the data set results unique to the agent with the specified kernel ( $K_{a_1}$ ) and id ( $hash(K)$ ) based on code base  $c_{175}$ .



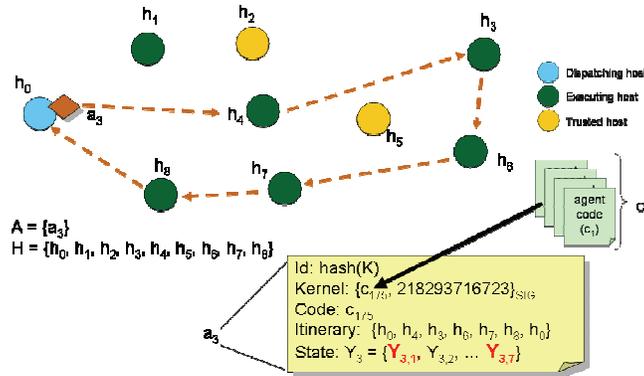
**Figure 142: Agent Code, Unique Agent Instance and State Set**

Figure 143 illustrates another agent instance ( $a_2$ ), a single dispatching host  $h_0$ , and illustrates the itinerary among a set of hosts ( $h_1 \dots h_8$ ). The state set  $Y_2$  indicates the data set results unique to the agent with the specified kernel ( $K_{a_2}$ ) and id ( $hash(K)$ ) based on code base  $c_{175}$ . This elaboration represents multiple agent instances using the same codebase.



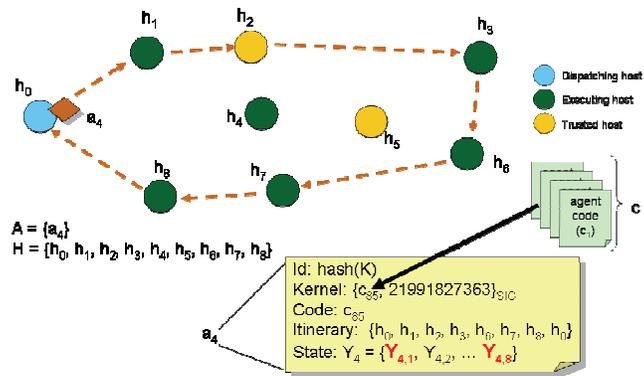
**Figure 143: Same Agent Code used in Different Agent Instance**

Figure 144 illustrates another agent instance ( $a_3$ ), a single dispatching host  $h_0$ , and illustrates the itinerary among a set of hosts ( $h_3 \dots h_8$ ). The state set  $Y_3$  indicates the data set results unique to the agent with the specified kernel ( $K_{a_3}$ ) and id ( $hash(K)$ ) based on code base  $c_{175}$ . This elaboration represents another agent instances using the same codebase but with different itinerary. Figure 145 illustrates an application based on a different codebase ( $c_{85}$ ) but the same itinerary as a previous agent application. The state set  $Y_4$  indicates the data set results unique to the agent with the specified kernel ( $K_{a_4}$ ) and id ( $hash(K)$ ) based on code base  $c_{85}$ : again all unique for this instance. This elaboration represents a different agent application logic sent to a preidentified host set.



Same codebase, different itinerary as  $a_1$ :  $c_{175}$

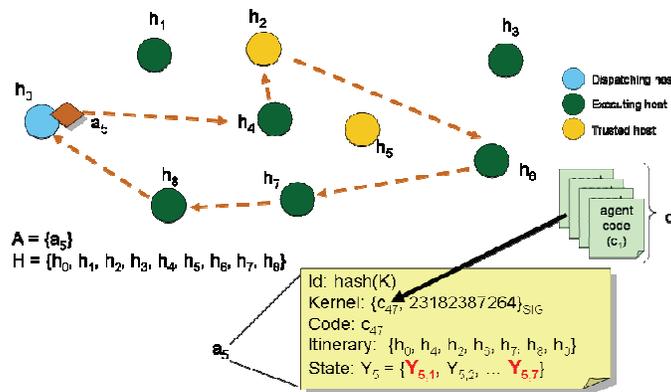
Figure 144: Same Agent Code, Different Agent Itinerary



Same itinerary, different codebase as  $a_1$ :  $c_{65}$

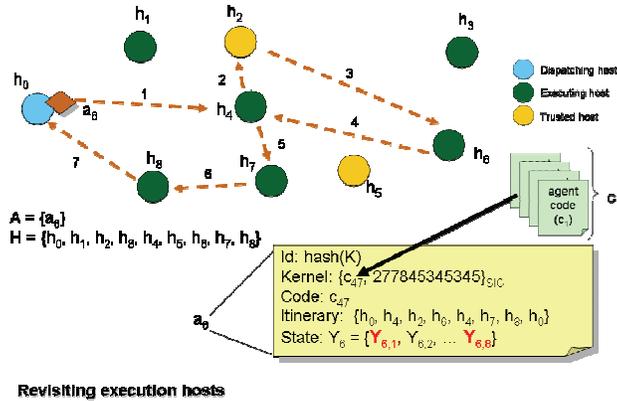
Figure 145: Different Code Base, Same Agent Itinerary

Figure 146 illustrates a different agent codebase used for an agent sent to a completely different host set. The same dispatching host and application owner send these in every case—which represents the possible application of agent code by a single party for different purposes.. In Figure 147, another agent instance uses an itinerary with the capability to revisit prior hosts. This elaboration is common for multi-bid updatable auction agents.



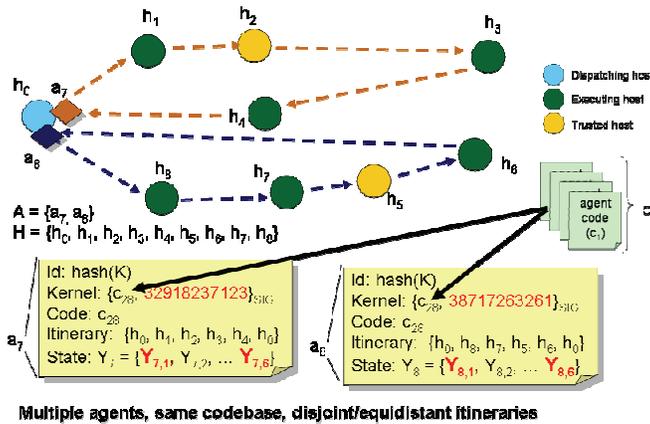
Different itinerary, different codebase as  $a_1$ :  $c_{47}$

Figure 146: Different Codebase, Different Agent Itinerary

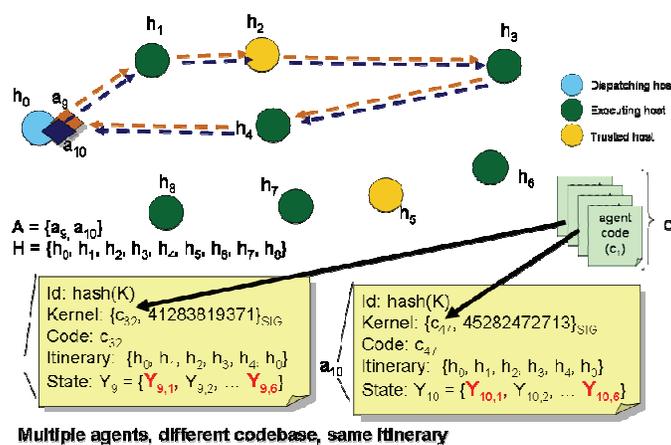


**Figure 147: Agent That Revisits Hosts in Itinerary**

Figure 148 illustrates a dispatching host that sends two agents, each using the same codebase  $C_{28}$ , to a disjoint set of hosts (each has a unique static itinerary). In every case, agents possess a unique ID and kernel, no matter if they visit the same hosts from a prior instance or if they use the same codebase. Several multi-agent protection schemes (including MADIMA), use replicated, identical codebase agents. In Figure 149, multiple agents



**Figure 148: Multiple Agents with Same Codebase, Unique Itineraries**



**Figure 149: Multiple Agents with Same Codebase, Common Itinerary**

## APPENDIX C

### ENUMERATING COMBINATIONAL CIRCUITS

In this appendix, we elaborate the possibilities for enumerating combination circuits. For future work, the use of sequential circuits (and thus the possibility of cycles within the the circuit description) will change the enumeration possibilities for gate inputs to include any previous or future gate.

**Enumeration 1.** Given a single dual input Boolean gate with  $n$  possible inputs, how many combinations of circuits are possible? Since there are  $n$  objects (nodes/inputs) taken 2 at a time, the permutation  $P(n,2)$  returns the possible number of combinations. Since two gates are equivalent if they have the same input, regardless of order, (i.e,  $AND(x_2,x_1) \approx AND(x_1,x_2)$ ), we want permutations where order does not count. The combination  $C(n,2)$  represents the possible combinations:

$$C(n,2) = \frac{n!}{2!(n-2)!}$$

For example: Given a single dual input Boolean gate of type AND with possible inputs of  $\{1,2,3\}$ , the possible combinations of gates given these inputs are:

$$\begin{aligned} AND(1,2) &\approx AND(2,1) \\ AND(1,3) &\approx AND(3,1) \\ AND(2,3) &\approx AND(3,2) \end{aligned}$$

$$n = 3$$

$$C(3,2) = 3! / 2!(3-2)! = 6/2 = 3$$

**Enumeration 2.** If we allow gates to receive identical inputs, then given a single dual input Boolean gate with  $n$  possible inputs, the number of possible combinations of inputs to gate (with repetition) is given by adding the combination  $C(n,2)$  with the possible number of repeated input combinations. For input size  $n$ , there  $n$  possible repeated input combinations (i.e,  $(1,1)$ ,  $(2,2)$ , ...  $(n,n)$ ). Given 2 inputs and  $n$  ways to choose them, the number of possible gate combinations is:

$$C(n,2) + n = \frac{(n)!}{2!(n-2)!} + n$$

For example: Given a single dual input Boolean gate of type AND with possible inputs of  $\{1,2,3\}$ , the possible combinations of gates given these inputs are:

$$\begin{aligned} AND(1,1) \\ AND(1,2) &\approx AND(2,1) \\ AND(1,3) &\approx AND(3,1) \\ AND(2,2) \\ AND(2,3) &\approx AND(3,2) \\ AND(3,3) \end{aligned}$$

$$n = 3$$

$$C(3,2) + 3 = 3! / 2!(3-2)! + 3 = 3 + 3 = 6$$

$$C(n,2) + n = \frac{(n)!}{2!(n-2)!} + n \text{ is equivalent to } C(n+1,2) = \frac{(n)!}{2!(n-2)!}$$

$$C(n+1,2) = \frac{(n+1)!}{2!(n+1-2)!} = \frac{n!(n+1)}{2!(n-1)!} = \frac{n!(n+1)}{2! \frac{n!}{n}} = \frac{n!(n+1)n}{2!n!} = \frac{(n+1)n}{2!} = \frac{n^2+n}{2}$$

$$C(n,2) + n = \frac{n!}{2!(n-2)!} + n = \frac{n!}{2! \frac{n!}{n(n-1)}} + n = \frac{n!n(n-1)}{2!n!} + n = \frac{n(n-1)}{2} + n = \frac{n^2-n}{2} + \frac{2n}{2} = \frac{n^2+n}{2}$$

So, let  $C(n+1,2)$  represent the number of possible combinations of a single dual input Boolean gate with  $n$  possible inputs.

**Enumeration 3.** Given a basis  $\Omega$  with size  $|\Omega|$ , the number of possible combinations of all single dual input Boolean gates over this basis with  $n$  possible inputs is given by the total number of possible input combinations per gate (from Definition 2, assuming we allow replicated inputs) times the total number of possible gate types (give by the basis size  $|\Omega|$ ). Therefore, the number of possible combinations is:

$$C(n+1,2) * |\Omega| = \frac{(n+1)!}{2!(n+1-2)!} * |\Omega| = \frac{|\Omega| (n+1)!}{2(n-1)!}$$

**Enumeration 4.** Let a circuit be defined by its inputs  $n$  and a set of gates  $\{Gt_{n+1}, Gt_{n+2}, \dots, Gt_{n+s}\}$ . We want to find the number of different possible circuits that can be created from the combinations of gates over a basis  $\Omega$  with input size  $n$ . Each gate in the circuit may be defined only by inputs from any previous gate of the circuit. Therefore, the first gate in the circuit has possible combinations based only on the input size of the circuit. All subsequent gates in the circuit can be derived from combinations of inputs that come from any previous gates, including the inputs.

Let  $G_1$  represent the number of possibilities for a 1-gate circuit (size  $s = 1$ ) of input size  $n$  and basis  $\Omega$ , assuming we allow replicated inputs.  $G_1$  is given by Definition 3:

$$G_1 = C(n+1,2) * |\Omega| = \frac{(n+1)!}{2!(n+1-2)!} * |\Omega| = \frac{|\Omega| (n+1)!}{2(n-1)!}$$

**Enumeration 5.** For a 2 gate-circuit (size  $s = 2$ ) with basis  $\Omega$  and input size  $n$ , the total number of possible inputs for Gate 1 ( $G_1$ ) comes from only from the  $n$  inputs  $INPUTS_n = \{x_1, \dots, x_n\}$ . The number of possible gate configuration for Gate 1 is given by:

$$G_1 = C(n+1,2) * |\Omega| = \frac{(n+1)!}{2!(n+1-2)!} * |\Omega| = \frac{|\Omega| (n+1)!}{2(n-1)!}$$

The number of inputs for Gate 2 derives from all possible combinations of  $\{G_1 \cup INPUTS_n\}$ . The possibilities for input to Gate 2 are the number of possible inputs choose 2 (since it is a dual input Boolean gate). This is identical to the previous definition, except we have one more input to choose from:  $n + 1 + 1 = n + 2$ .  $G_2$  is therefore given by:

$$G_2 = C(n+2,2) * |\Omega| = \frac{(n+2)!}{2!(n+2-2)!} * |\Omega| = \frac{|\Omega| (n+2)!}{2n!}$$

The number of possible 2-gate circuits for basis  $\Omega$  and input size  $n$  is given by  $G_1 * G_2$ , since every possible Gate 1 (whose size is given by  $G_1$ ) has  $G_2$  possibilities for Gate 2 with which to complete the 2-gate circuit.

**Enumeration 6.** Given an  $s$ -gate (size  $s$ ) circuit  $C$  with gate set  $\{G_{t_{n+1}}, G_{t_{n+2}}, \dots, G_{t_{n+s}}\}$ , under basis  $\Omega$  and having input size  $n$ , the total number of possible combinations for any given gate  $z$  ( $G_z$ ) in the set of gates for  $C$  is given by the following relationship, from Definition 4 and 5:

$$G_z = C(n+z, 2)^* |\Omega| = \frac{|\Omega| (n+z)!}{2!(n+z-2)!}$$

**Enumeration 7.** Given a circuit  $C$  of size  $s$  with input size  $n$ , the number of possible  $s$ -gate circuit combinations  $G_C$  possible under basis  $\Omega$  is given by the product of the possibilities of each of the individual gate possibilities.

$$G_C = \prod_{i=1}^s G_i$$

From Enumeration 6:

$$G_C = \prod_{i=1}^s C(n+i, 2)^* |\Omega|$$

## APPENDIX D

### PROGRAM ENCRYPTION ARCHITECTURE

The Mobile Agent Software Code Obfuscation Tool (MASCOT) provides a foundational set of applications for implementing and testing basic functionality associated with our theoretic results. We have created a suite of applications that support the analysis work for the techniques described in Section 5.4, 5.6, and 5.7. Figure 150 illustrates the basic architecture to support circuit randomization and program black-box encryption. The Encryption Program Generation Engine (EPGE) provides key-based unique data ciphers with associated recovery mechanisms for use in program concatenation (accomplished by Program Output Concatenation Engine or POCE). EPGE and POCE provide the basic functionality for taking input program  $P$  and creating program  $P'' = E(P,K)$ . The TANGLE portion of the architecture takes program  $P''$  and performs code-level randomization that approximates the random selection of a circuit from the family of circuits that all implement the one-way function  $E$ . The output of the circuit replacement library and TANGLE interaction represents the final encrypted program  $P'$ .

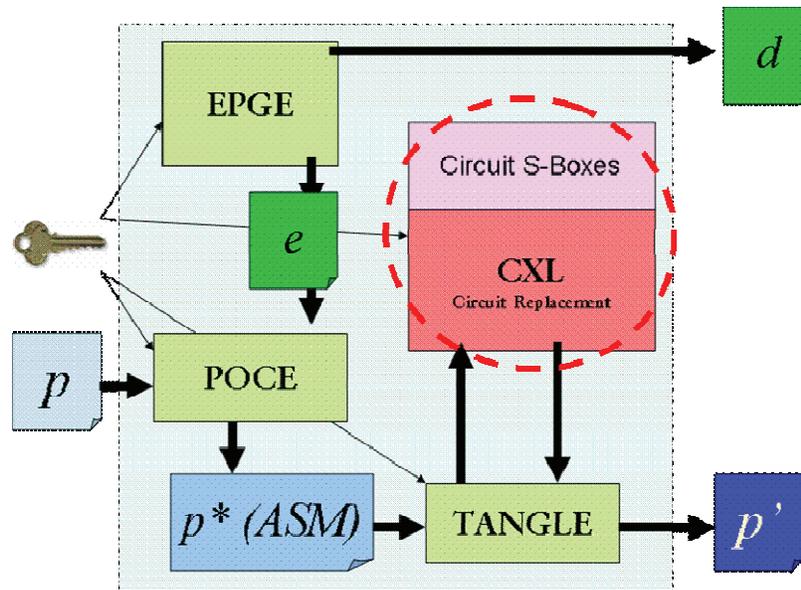
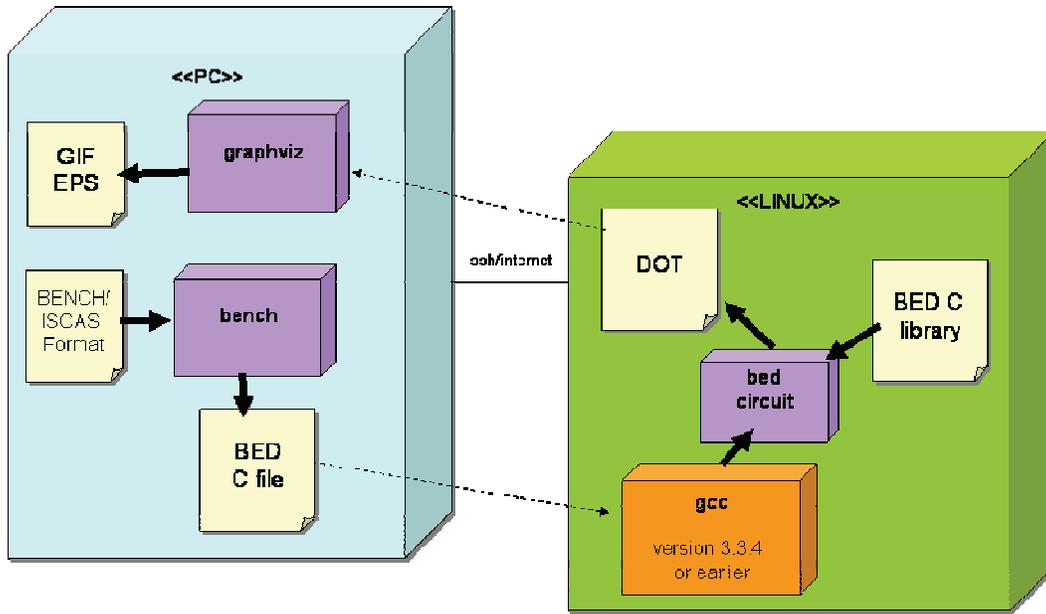


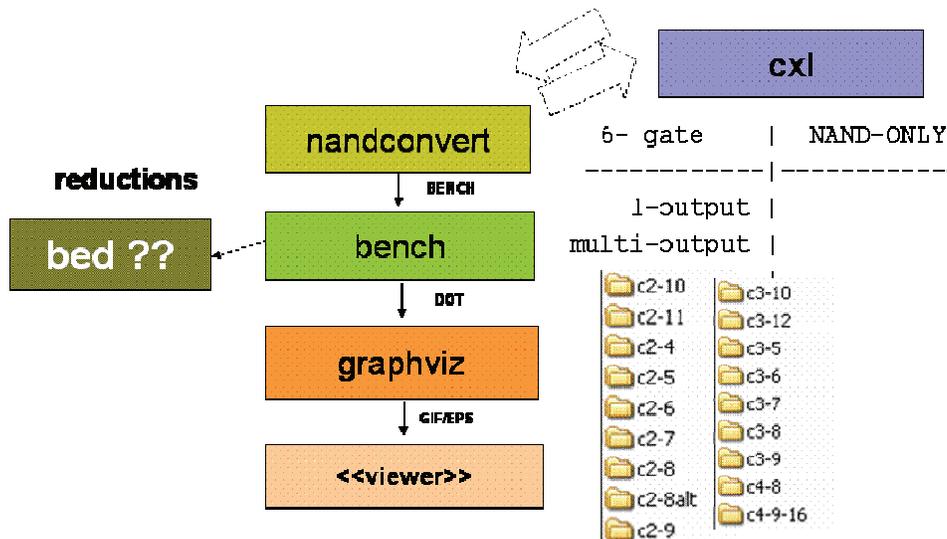
Figure 150: MASCOT Generalized Randomization

Figure 151 gives a deployment diagram for several architecture components that support the generation and viewing of BENCH format circuits. We discuss each component of our system in detail following. The *bench* component takes in BENCH format circuit-description files (discussed next in Appendix E) and outputs a C file that creates a Boolean Expression Diagram (BED) version of the circuit. The BED libraries are currently limited to UNIX/LINUX *gcc* implementation, particularly where *gcc* is version 3.3.4 or earlier. Unfortunately, more current *gcc* versions do not support deprecated features that the original BED libraries use extensively [204]. Future work will involve creating custom BED (or DAG) manipulation libraries for both randomization and visualization. We primarily use the BED libraries for visualization and we would need to expand the library to incorporate native ability to select and replace sub-circuits or DAG sub-graphs with specific properties. We must compile the BED C file (which we can produce on PC or LINUX using standard C calls) under GCC version 3.4.4 on a Linux platform. Once compiled, we take the executable version of the BED circuit file and use it to create a graphical view of the circuit (currently). The executable BED circuit produces a DOT representation (discussed further in Appendix E and F) that graphically describes the DAG in a hierarchical form. The DOT file creates a corresponding graphical file using a standard PC-based tool known as *graphviz*.



**Figure 151: ISCAS, BENCH, BED Deployment Diagram**

The circuit generation library (cxl) currently runs only on LINUX platforms as well and we will in future work port it to work in generic/portable C++ or Java libraries. Figure 152 describes the series of components that we use to interface with the cxl library that requires pregeneration of circuits that supports non-linear selection and replacement. The circuit generation library currently produces binary representation of circuits (not BENCH format) and future work will involve interfacing and porting the cxl library to interact directly with native BENCH format circuit representations. The end-goal is to provide a functionality to read in a BENCH format description, select a sub-circuit from within the DAG representation (with characteristic properties such as fan-in, fan-out, or depth), and then replace that sub-circuit with a semantically equivalent version. Once this functionality is complete, we can then perform this operation in a round-based fashion. Future experimentation will involve determining how many rounds of selection and replacement are necessary to randomize fully an input circuit with measurable properties.



**Figure 152: Circuit Generation Library and Replacement**

We describe now the specific components that support end-to-end program and circuit encryption. Figure 72 (p. 112) and Figure 73 (p. 115) give a high-level overview of our generalized circuit randomization process and our perfect white-box protection technique. We outline the codebase that currently supports program-level and circuit-level analysis work as follows.

**EVALUATE.** We need the ability to take a circuit specification and simulate its gates using input to produce real output. The EVALUATE component provides this functionality by taking an ISCAS BENCH format circuit and an associated input for that circuit. Typically, we require the full  $2^n$  range of  $n$  possible inputs. Figure 153 illustrates that EVALUATE reads the circuit description, allocates appropriate data structures for each gate and input, applies the inputs provided, and evaluates the circuit for each input to produce an associated output. The output of the component is a truth table formatted (input with output) or single data column (output) that represents the full execution of the circuit on all inputs. Currently, both input and output data is in textual (as opposed to binary) form of ASCII ones and zeros (0/1). The figure illustrates the C-17 benchmark circuit after evaluation: all inputs and all outputs in truth table form.

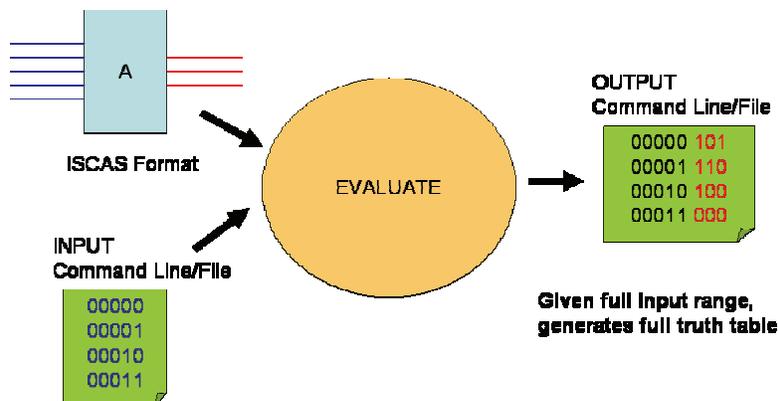
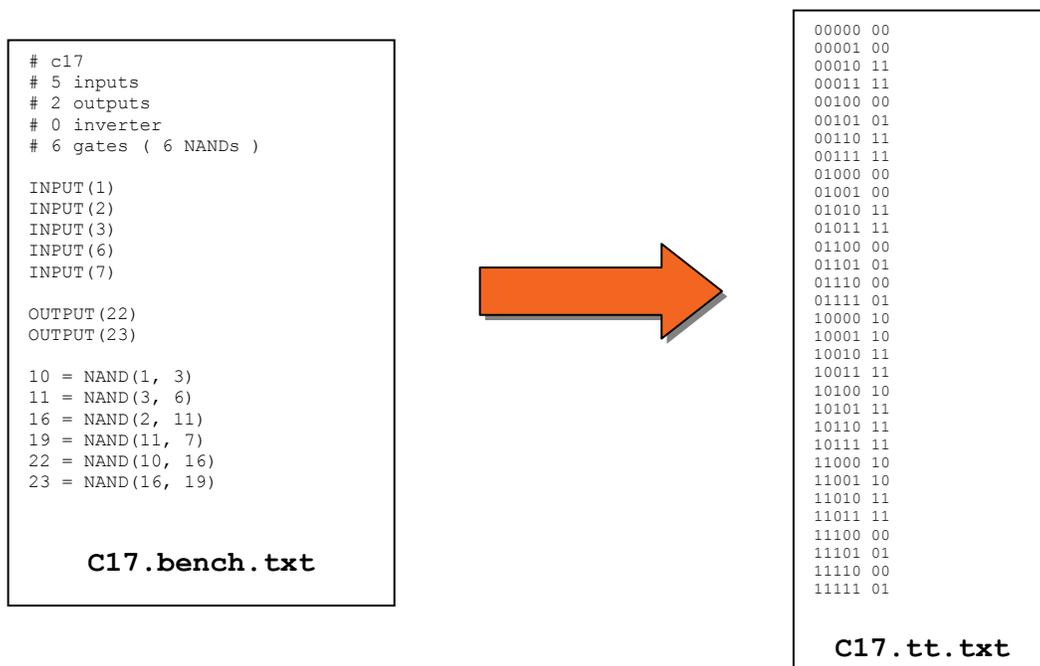


Figure 153: MASCOT-EVALUATE Component

**GENINPUT.** As Figure 154 illustrates, GENINPUT produces a text based elaboration of all inputs given some input size  $n$ . The output file is the  $2^n$  textual form of all inputs from  $\{0\}^n$  to  $\{1\}^n$ . We can then use such input files to evaluate data ciphers or circuits (using EVALUATE) or to produce truth table collections.

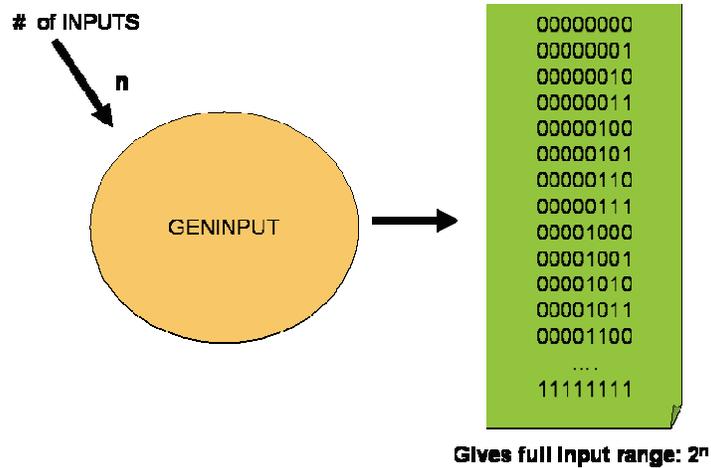


Figure 154: MASCOT-GENINPUT Component

**PAD.** Figure 155 illustrates the PAD component which takes some number of bits and an existing textual file with binary inputs represented as 0/1 ASCII strings. PAD takes the original file and prepends  $n$  number of 0 digits to each input row value. Such functionality is useful for converting circuit or program outputs to larger forms in order to meet input requirements for target circuits (such as 3DESBIN). For elaborating data ciphers outputs given all inputs, such a tool provides the ability to format input properly.

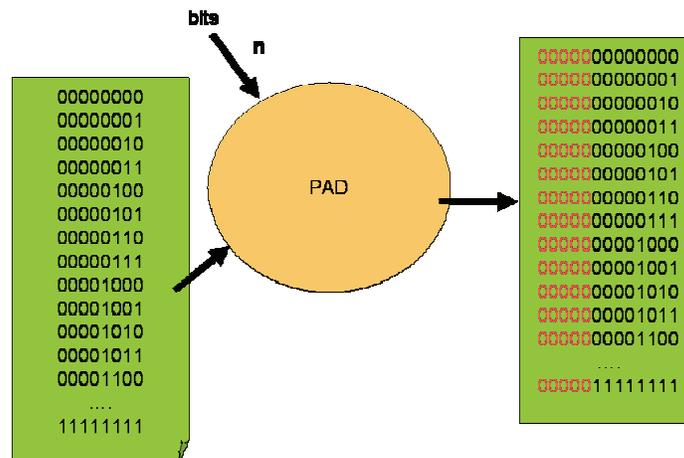


Figure 155: MASCOT-PAD Component

**DEPAD.** Figure 156 illustrates another input formatting tool that allows us to strip away the superfluous 0 bits from a series of binary strings. The program calculates the largest pre-string of 0s common to all strings in the file and then depads that number of bits from each binary string. This component is useful for formatting outputs of ciphers (such as RSABIN) that produce non-uniform output.

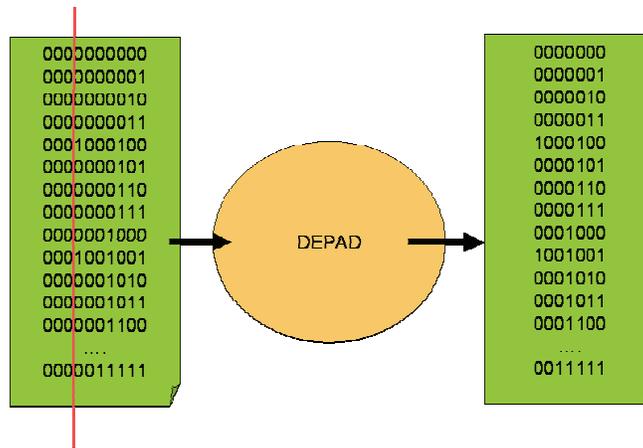


Figure 156: MASCOT-DEPAD Component

**RANDCIRCUIT.** In order to analyze the properties of random circuits based on some basis, an input and output size, and some maximum number of circuits, we find it useful to create circuits with random properties and specified parameters. Figure 157 illustrates that RANDCIRCUIT produces an ISCAS BENCH format specification based on given parameters. Currently, we only consider combination circuit logic and the possibility exists for future sequential consideration. RANDCIRCUIT first ensures that all inputs are used, the total gate size is reached, and that the basis gates are distributed uniformly across wires. Future work will involve changing the core randomization algorithm and examining variance across circuit incarnations. Figure 157 lists the BENCH code for a 10 input, 2 output random circuit with 30 total gates.

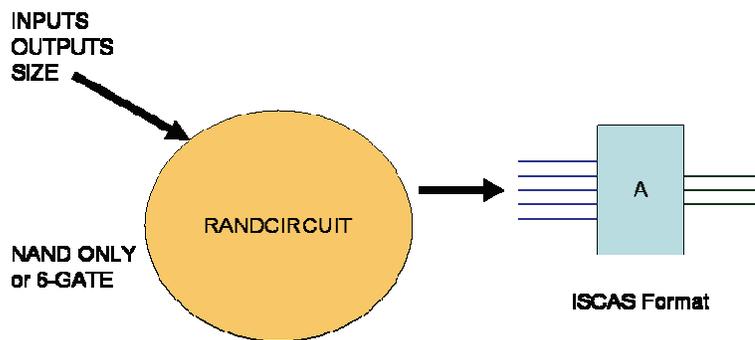


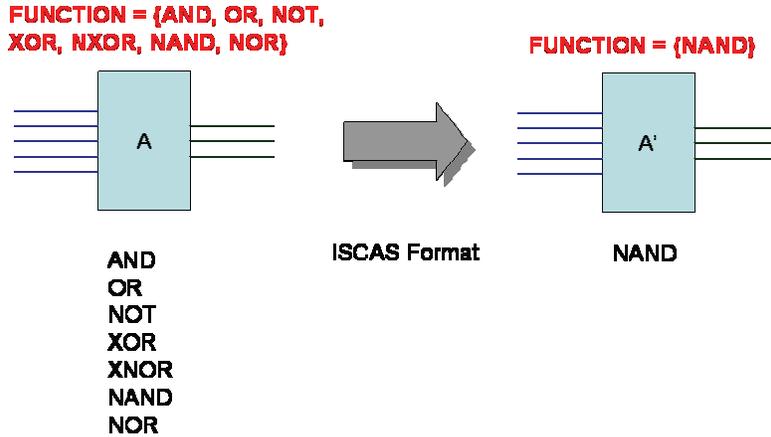
Figure 157: MASCOT-RANDCIRCUIT Component

```
# r10-2-30g
# 10 inputs
# 2 outputs
# 30 total gates
#
# INPUTS
INPUT (C1)
INPUT (C2)
INPUT (C3)
INPUT (C4)
INPUT (C5)
INPUT (C6)
INPUT (C7)
INPUT (C8)
INPUT (C9)
INPUT (C10)

# OUTPUTS
OUTPUT (C39)
OUTPUT (C40)

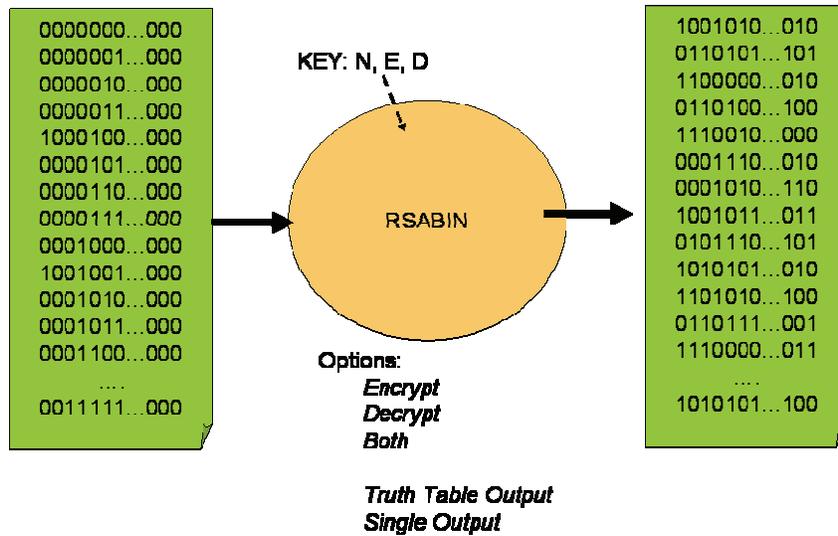
# GATES
C11 = NAND (C8, C6)
C12 = NAND (C1, C6)
C13 = NAND (C9, C6)
C14 = NAND (C8, C6)
C15 = NAND (C7, C6)
C16 = NAND (C1, C6)
C17 = NAND (C5, C5)
C18 = NAND (C4, C10)
C19 = NAND (C2, C3)
C20 = NAND (C11, C16)
C21 = NAND (C11, C14)
C22 = NAND (C18, C16)
C23 = NAND (C19, C11)
C24 = NAND (C10, C14)
C25 = NAND (C15, C19)
C26 = NAND (C20, C11)
C27 = NAND (C18, C18)
C28 = NAND (C13, C15)
C29 = NAND (C13, C22)
C30 = NAND (C23, C14)
C31 = NAND (C25, C16)
C32 = NAND (C13, C14)
C33 = NAND (C14, C17)
C34 = NAND (C19, C15)
C35 = NAND (C23, C29)
C36 = NAND (C33, C29)
C37 = NAND (C34, C27)
C38 = NAND (C17, C18)
C39 = NAND (C13, C36)
C40 = NAND (C22, C15)
```

**NANDCONVERT.** Figure 158 illustrates a rudimentary component for circuit specification manipulation. Specifically, we need the ability to examine uniform bases such as {NAND} and understand their effect on circuit recognition and randomization properties. We provide a NANDCONVERT feature that reads in a circuit description in BENCH format and produces an equivalent NAND only version. The conversion performs a gate-by-gate conversion process that ensures a semantically equivalent circuit version, in BENCH format.



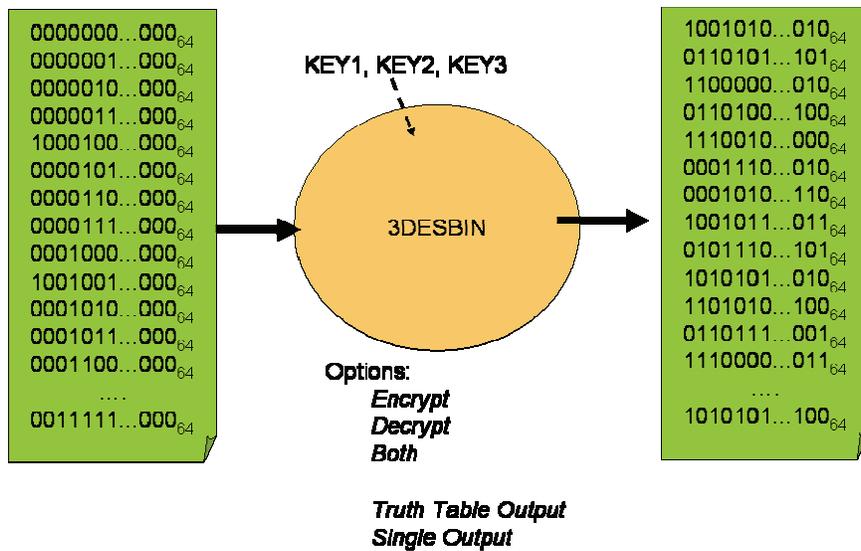
**Figure 158: MASCOT-NANDCOVERT Component**

**RSABIN.** Figure 159 illustrates a useful component to support EPGE operations under both white-box perfect encryption and circuit randomization. As we state in our theoretical results, the beginning point of program encryption are one-way functions. We provide RSABIN as program that will take an RSA key (modulus,E,D) and a sequence of binary string inputs (in ASCII) and then produce the corresponding ciphertext for each string using the RSA algorithm. The program is useful for concatenating outputs of a program as input to an RSA cipher with a specific key. RSABIN has options to format the output either as a single output column or as a truth table version. The program also has the option to perform encrypt or decrypt operations (or both) on given binary string inputs.



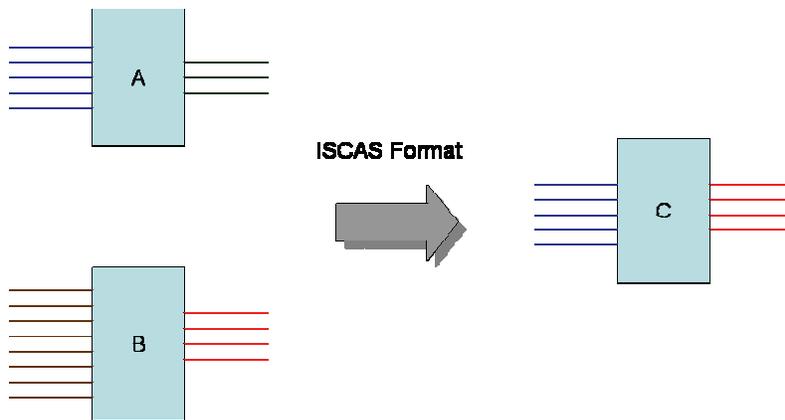
**Figure 159: MASCOT-RSABIN Component**

**3DESBIN.** Much like RSABIN, 3DESBIN provides functionality to create binary string output given binary string input based on a 3DES cipher algorithm. Figure 160 illustrates input as 64-bit binary strings and output as 64-bit binary strings, assuming a given set of 3DES keys (56-bits each). The 3DESBIN component allows the easy creation of truth table formatted output that supports canonical circuit creation and minimization.



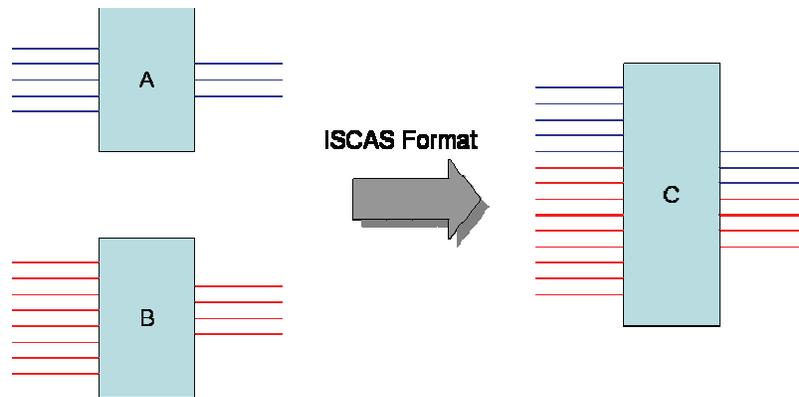
**Figure 160: MASCOT-3DES BIN Component**

**CONCAT.** This component takes two inputs circuits A and B (in ISCAS BENCH format) and creates a new circuit C such that  $\forall x, C(x) = B(A(x))$ . This component supports concatenation of a target program P with a key-embedded data cipher algorithm  $E_k$ . Figure 161 illustrates circuit A (5 input/3 output) and circuit B (8 input, 4 output) concatenated together to produce circuit C (5 input/4 output). In this example, the output of circuit A (3 bits) must be padded to 8 bits (5 pad bits) in order to match the input specification for circuit C.



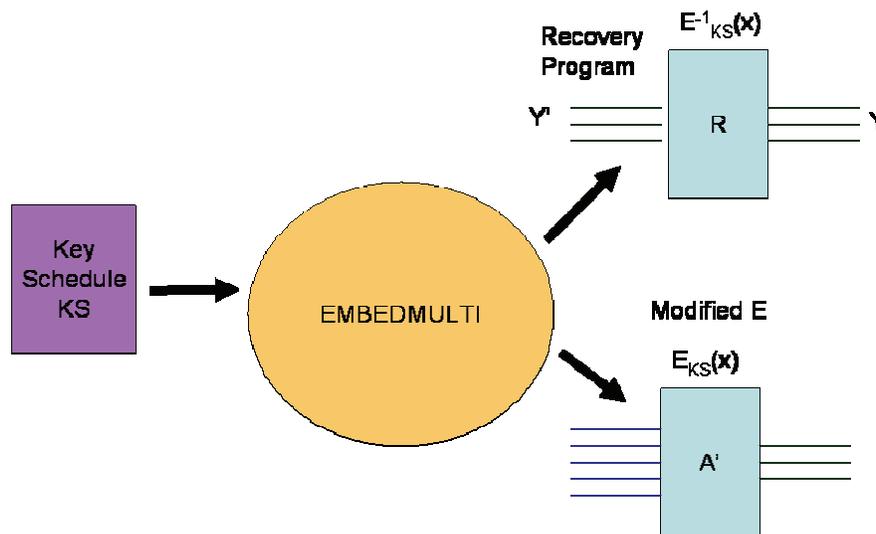
**Figure 161: MASCOT-CONCAT Component**

**MERGE.** This component takes two inputs circuits A and B (in ISCAS BENCH format) and creates a new circuit C such that  $\forall x, C(x) = A(x) \times B(x)$ . The merge functionality indicates that the composite circuit (C) takes the input space of both A and B and produces a binary output string that is the output of  $A(x)$  concatenated with the output of  $B(x)$ . Figure 162 illustrates circuit A (4 inputs/3 outputs) and circuit B (8 inputs/4 outputs) merged to produce circuit C with 4+8 (12) inputs and 3+4 (7) outputs. Merge operations support circuit obfuscation techniques where we introduce multi-function logic in order to confuse or obscure true functionality or logic. We may also use such techniques to provide a smart form of padded input, especially when the merged circuit is a randomly selected circuit with fixed input and output.



**Figure 162: MASCOT-MERGE Component**

**EMBEDMULTI.** Figure 163 illustrates the EMBEDMULTI component within MASCOT that produces a simple, perfect secrecy encryption circuit (A) and appropriate recovery circuit (R) based on a given key schedule. The perfect secrecy cipher provides a useful and simple technique to create experimental encryption circuits used to concatenate the output of a target circuit or program P. The cipher circuits provide a customizable input and output interface for target programs with specific number of output bits.



**Figure 163: MASCOT-EMBEDMULTI Component**

**CIRC2PROG.** Figure 164 illustrates the operation of the CIRC2PROG component that takes any ISCAS BENCH format file and creates a semantically equivalent C program using a custom Boolean component library that we developed in our research. The C program takes the same number of inputs, produces the same number of outputs as the BENCH circuit, and ensures truth table equivalence of both. The custom C library operates on binary string data (in ASCII format) and uses C library routines to simulate individual dual Boolean gate operations (AND, OR, XOR, NOR, etc). The CIRC2PROG component provides a seamless method for creating native binary programs for various architectures (PC, LINUX, etc.) based on BENCH logic. The equivalent C program takes identical input and produces identical output as the BENCH circuit.



**CANONICAL.** Figure 167 illustrates one of the key architectural components to support perfect white-box encryption operations. CANONICAL takes as input a textual truth table file and realizes a complete sum-of-products circuit expression that reproduces the semantics of the truth table. CANONICAL currently does not perform any minimization on the complete SOP form, but we will incorporate such options in future work. The ISCAS format circuit represents the Boolean logic that exactly implements the supplied truth table. This component is the practical realization for perfectly white-box protected circuits created via concatenation using strong data ciphers (P + E).

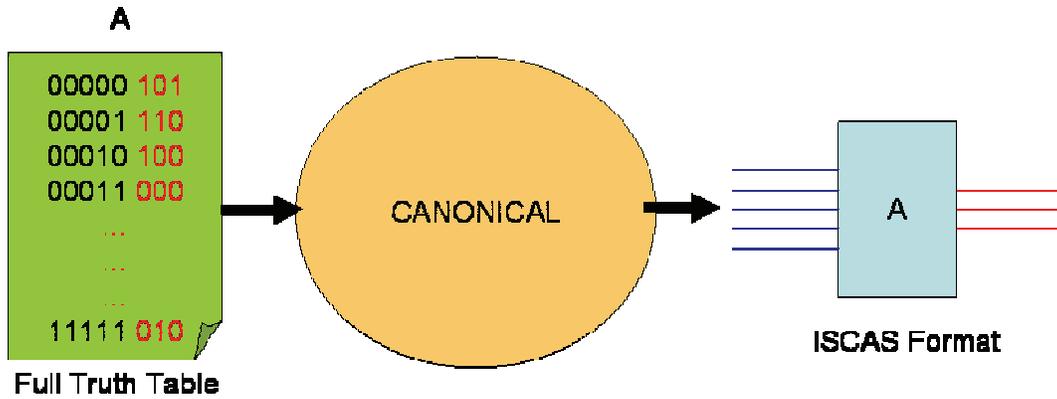


Figure 167: MASCOT-CANONICAL Component

## APPENDIX E

### ISCAS CIRCUIT DEFINITIONS AND BENCH FORMAT

In order to facilitate circuit randomization techniques (described in Section 5.6), we chose a standard textual circuit representation language and a well-known set of benchmark circuits that provide combination logic to work with. Davidson and Harlow provide an overview of the process involved for benchmark-circuit library development in [380]. As they describe, Franc Brglez and Hideo Fujiwara began work in 1984 on initial collection of combinational circuit benchmarks for use in automatic test pattern generation (ATPG). Researchers presented results for using these circuits in 1985 at the International Symposium on Circuits and Systems (ISCAS). The ISCAS '85 benchmarks (seen in Table 30), and their successors, are now available in several net list formats. The Association for Computing Machinery, Design Automation Special Interest Group (ACM/SIGDA) benchmarks as well as the ITC suite provides us with a future body of known logic that includes both sequential and combinational testing possibilities. Hansen *et al.* [203] provide a summary of their reverse engineering work on the ISCAS 85 circuits and Table 30 presents several of their high-level functional graphical representation views.

Figure 168 illustrates the C-17 benchmark in BENCH notation and a sample logic circuit with 5 inputs and 2 outputs (with 5 logic gates). BENCH is a circuit description language originally utilized for describing ISCAS-85 benchmarks but that still remains widely used in academia and industry for testing<sup>18</sup>. Figure 169 illustrates the BENCH format in extended BNF notation. Circuits based on sequential logic use Boolean gates with feedback, such as flip-flops. Such netlists do map to directed acyclic representations because cycles are inherently present with feedback or memory logic. Several academic and industrial tools are equipped to convert BENCH formats to other textual representation forms such as VHDL, Verilog, and library exchange format (LEF).

<b># c17</b>	
<b># 5 inputs</b>	# 3+2 inputs
<b># 2 outputs</b>	# 2 outputs
<b># 0 inverter</b>	
<b># 6 gates ( 6 NANDs )</b>	
<b>INPUT(1)</b>	INPUT (A)
<b>INPUT(2)</b>	INPUT (B)
<b>INPUT(3)</b>	INPUT (C)
<b>INPUT(6)</b>	INPUT (D)
<b>INPUT(7)</b>	INPUT (E)
<b>OUTPUT(22)</b>	OUTPUT (I)
<b>OUTPUT(23)</b>	OUTPUT (J)
<b>10 = NAND(1, 3)</b>	F = NAND (A, B)
<b>11 = NAND(3, 6)</b>	G = NOR (A, C)
<b>18 = NAND(2, 11)</b>	H = AND (B, G)
<b>19 = NAND(11, 7)</b>	I = XOR (D, F)
<b>22 = NAND(10, 18)</b>	J = XOR (E, H)
<b>23 = NAND(18, 19)</b>	

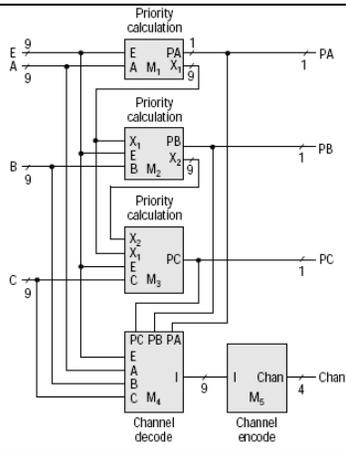
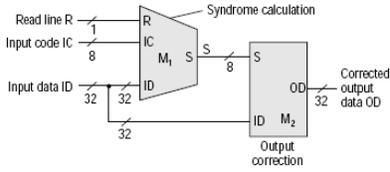
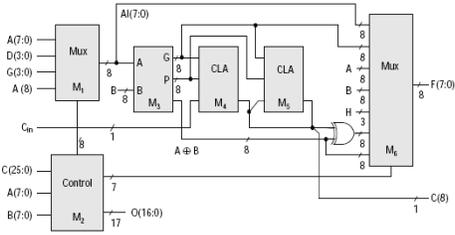
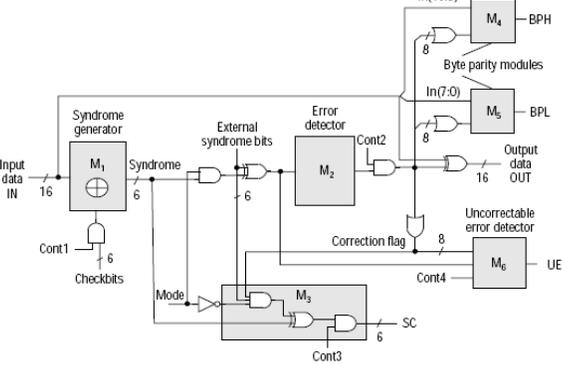
(a) C17-Benchmark Circuit

(b) 5-input/2 Output/5 Gate Circuit

Figure 168: BENCH Circuit Format

<sup>18</sup> See <http://www.fm.vslib.cz/~kes/asic/iscas/>

**Table 30: High Level View of ISCAS-85 Circuit Library**

<p><a href="#">c432</a> : 27-channel interrupt controller</p> <p>36 inputs 7 outputs 160 logic gates 5 major functional blocks</p>	
<p><a href="#">c499/c1355</a>: 32-bit single error correcting circuit</p> <p>41 inputs / 32 outputs 202 (546) logic gates 2 major functional blocks</p>	
<p><a href="#">c880</a>: 8-bit ALU</p> <p>60 inputs 26 outputs 383 logic gates 7 major functional blocks</p>	
<p><a href="#">c1908</a>: 16-bit single error correcting / double error detecting circuit</p> <p>33 inputs / 25 outputs 880 logic gates 6 major functional blocks</p>	
<p><a href="#">c2670</a>: 12-bit ALU and controller</p>	<p>233 inputs / 140 outputs 1,193 logic gates / 7 major functional blocks</p>
<p><a href="#">c3540</a>: 8-bit ALU</p>	<p>50 inputs / 22 outputs 1,699 logic gates / 11 major functional blocks</p>
<p><a href="#">c5315</a>: 9-bit ALU</p>	<p>178 inputs / 123 outputs 2,307 logic gates / 10 major functional blocks</p>
<p><a href="#">c6288</a> : 16x16 multiplier</p>	<p>32 inputs / 32 outputs 2,406 logic gates / 240 major functional blocks</p>
<p><a href="#">c7552</a> : 32-bit adder/comparator</p>	<p>207 inputs / 108 outputs 3,512 logic gates / 8 major functional blocks</p>

```

<netlist> ::= [<comment>]* {<input> [<comment>]*}*
           [<comment>]* {<output> [<comment>]*}*
           [<comment>]* {<gate> [<comment>]*}* [<comment>]*
<input>   ::= INPUT ( <wire> )
<output>  ::= OUTPUT ( <wire> )
<gate>    ::= <single> | <multiple>
<single>  ::= <wire> = <single-op> ( <wire> )
<multiple> ::= <wire> = <multi-op> ( <wire> [, <wire>]* )
<single-op> ::= NOT | BUFFER
<multi-op> ::= AND | OR | XOR | NXOR | NAND | NOR
<wire>    ::= { <letter> | <digit> | <specialchar> }+
<letter>  ::= A | B | C | ... | Z | a | b | c | ... | z
<digit>   ::= 0 | 1 | 2 | ... | 9
<specialchar> ::= ! | @ | _ | - | $ | % | &
<comment> ::= # { <letter> | <digit> | <specialchar> }+

```

Figure 169: BENCH Grammar in Extended BNF Notation

## APPENDIX F

### BOOLEAN EXPRESSION DIAGRAMS

Solving constraint satisfaction problems and formal verification have been catalyst to a myriad of graphical structures that support graph-based Boolean function manipulation: Binary Decision Diagrams (BDD), Reduce Ordered Binary Decision Diagrams (ROBDD), FDD, OBDD, ADD, MTBDD, BMD, KMDD, and BGD to name a few. Anderson and Hulgaard [204] develop Boolean Expression Diagrams (BEDs) as another extension to BDDs; these forms can represent any Boolean function in *linear* space and provide standard graph-based tools for dealing with combinational-level logic problems. BEDs have been useful for efficiently determining circuit equivalence. Although varieties of graphical representation languages exist for combinational circuits, we chose BEDs because they come with an available library of functions in C that prove useful for implementation.

Anderson and Hulgaard provide a library for manipulating and creating BEDs. Their libraries support algorithms that transform a BED into a reduced ordered BDD (ROBDD): one algorithm using the BDD apply-operator and the other exploiting information of the Boolean expression. Standard BDD techniques present NP-complete problems that are infeasible to solve while BED approaches give relatively fast solution possibilities. In particular, Anderson and Hulgaard define a Boolean Expression Diagram in [204] as follows:

**Definition 19. (Boolean Expression Diagram)** *A BED is a directed acyclic graph  $G = (V, E)$  with vertex set  $V$  and edge set  $E$ . The vertex set  $V$  contains three types of vertices: terminal, variable, and operator.*

*A terminal vertex  $v$  has as attributes a value  $value(v) \in \{0, 1\}$*

*A variable vertex  $v$  has as attributes a variable  $var(v)$ , and two sons  $low(v), high(v) \in V$ .*

*An operator vertex  $v$  has as attributes a binary Boolean operator  $op(v)$ , and two sons  $low(v), high(v) \in V$ .*

*The edge set  $E$  is defined by  $\{(v, low(v)), (v, high(v)) \mid v \in V \text{ and } v \text{ is not a terminal vertex}\}$ .  $0$  and  $1$  are the two terminal vertices. Variable vertices correspond to the if-then-else operator  $x \rightarrow f_1, f_0 = (x \wedge f_1) \vee (\neg x \wedge f_0)$ . Operator vertices correspond to their respective Boolean connectives, leading to a correspondence between BEDs and Boolean functions.*

We utilize the code libraries for BEDs to build DAG representations of various combinational circuits in our architecture. We can construct BED nodes in either reduced or unreduced form: A BED program with reductions turned on will produce DAG representations with simplified Boolean expressions in reduced form. BED programs are normally used to define constraint satisfaction problems. In the MASCOT architecture, we utilize the inherent graphical display capabilities of the BED libraries to visualize and analyze combinational circuits. In order to use the BED libraries, a C program must include the BED C/C++ libraries and make appropriate library calls. A basis BED program allocates space for variables, creates bed nodes, and then performs other BDD operations as appropriate.

Our framework only currently produces a graphical version of the BED in DOT<sup>19</sup> notation by using the `bed_io_graph()` library call. DOT is an open-source graphics library that draws directed graphs as hierarchies. It reads attributed files (which we create using BED programs specific to a particular circuit) and writes out drawings in other graphical formats (GIF, JPG, PNG, SVG, Postscript). DOT specifies its own language, which we may use in future work to replace the BED libraries with our own custom DAG manipulation tools. Figure 170 illustrates the custom C

<sup>19</sup> <http://www.graphviz.org/Documentation/dotguide.pdf>

file we create from a BENCH specification (in this case, the C-17 ISCAS benchmark seen in Figure 61 and Figure 62 on p. 92). Once this program is compiled and executed, it produces a DOT file representing the BED graphical representation (either reduced or unreduced) of the circuit.

```
#include <stdio.h>
#include "bed.h"
#include "bedio.h"

int main()
{
    FILE *outFile;

    bed_init( 800*1024, 500*1024);
    bed_node_list *nodes = il_new();

    bed_var vin = bed_new_variables( 256 );

    bed_node c1 = bed_mk_var(vin, bed_false, bed_true);
    il_append (nodes, c1);
    bed_node c2 = bed_mk_var(vin+1, bed_false, bed_true);
    il_append (nodes, c2);
    bed_node c3 = bed_mk_var(vin+2, bed_false, bed_true);
    il_append (nodes, c3);
    bed_node c6 = bed_mk_var(vin+3, bed_false, bed_true);
    il_append (nodes, c6);
    bed_node c7 = bed_mk_var(vin+4, bed_false, bed_true);
    il_append (nodes, c7);
    bed_node c10 = bed_mk_op (BED_NAND, c1, c3);
    il_append (nodes, c10);
    bed_node c11 = bed_mk_op (BED_NAND, c3, c6);
    il_append (nodes, c11);
    bed_node c16 = bed_mk_op (BED_NAND, c2, c11);
    il_append (nodes, c16);
    bed_node c19 = bed_mk_op (BED_NAND, c11, c7);
    il_append (nodes, c19);
    bed_node c22 = bed_mk_op (BED_NAND, c10, c16);
    il_append (nodes, c22);
    bed_node c23 = bed_mk_op (BED_NAND, c16, c19);
    il_append (nodes, c23);

    if ( ( outFile = fopen("c17.dot", "w") ) == NULL ) {
        printf ("File could not be opened\n");
        exit(0);
    }

    bed_io_graph (outFile, nodes);
    bed_done();

    close (outFile);
    return 0;
}
```

**Figure 170: BED C Program Created from C-17 BENCH Specification**

To illustrate the graphical properties of the same circuit and the variability of circuit representation themselves, we consider next an example circuit description (P) with 3 inputs and 2 outputs. We can represent the circuit under different bases as well:  $\Omega = \{\text{AND, OR, NAND, NOR, XOR, XNOR}\}$  and  $\Omega = \{\text{NAND}\}$ . We show also the truth table view for the larger base circuit.

INPUT (3) INPUT (2) INPUT (1)  OUTPUT (7) OUTPUT (6)  4 = AND (3,2) 5 = OR (4,1) 6 = XOR (4,3) 7 = NAND (5,6)	<table border="1" style="border-collapse: collapse; margin: auto;"> <thead> <tr> <th>X1</th> <th>X2</th> <th>X3</th> <th>4</th> <th>5</th> <th>Y6</th> <th>Y7</th> </tr> </thead> <tbody> <tr> <td></td> <td></td> <td></td> <td>AND(3,2)</td> <td>OR(4,1)</td> <td>XOR(4,3)</td> <td>NAND(5,6)</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>0</td> <td>1</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>1</td> </tr> </tbody> </table>	X1	X2	X3	4	5	Y6	Y7				AND(3,2)	OR(4,1)	XOR(4,3)	NAND(5,6)	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0	1	0	0	0	0	1	0	1	1	1	1	0	1	1	0	0	0	1	0	1	1	0	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	1	1	0	1	INPUT (3) INPUT (2) INPUT (1)  OUTPUT (7) OUTPUT (6)  4a = NAND (3, 2) 4 = NAND (4a, 4a) 5a = NAND (4, 4) 5b = NAND (1, 1) 5 = NAND (5a, 5b) 6a = NAND (4, 4) 6b = NAND (3, 3) 6c = NAND (3, 6a) 6d = NAND (4, 6b) 6 = NAND (6c, 6d) 7 = NAND (5, 6)
X1	X2	X3	4	5	Y6	Y7																																																																		
			AND(3,2)	OR(4,1)	XOR(4,3)	NAND(5,6)																																																																		
0	0	0	0	0	0	1																																																																		
0	0	1	0	0	1	1																																																																		
0	1	0	0	0	0	1																																																																		
0	1	1	1	1	0	1																																																																		
1	0	0	0	1	0	1																																																																		
1	0	1	0	1	1	0																																																																		
1	1	0	0	1	0	1																																																																		
1	1	1	1	1	0	1																																																																		

**P,  $\Omega = \{\text{AND, OR, NAND, NOR, XOR, XNOR}\}$**

**P, Truth Table**

**P,  $\Omega = \{\text{NAND}\}$**

**Figure 171: Example Circuit P – Gate Level and Truth Table View**

Using example circuit P, we now illustrate several variations of this circuit with different BED manifestations. Figure 172 shows the original and all-NAND versions of P in BED format—in both reduced and unreduced forms. Figure 173 illustrates the canonical sum-of-products view of circuit P (including the all-NAND version) and their corresponding reduced BED views. Figure 174 illustrates our framework’s ability to characterize the graphical changes induced by change of base (to all-NAND) and to visualize the reductions inherent in the BED structure.

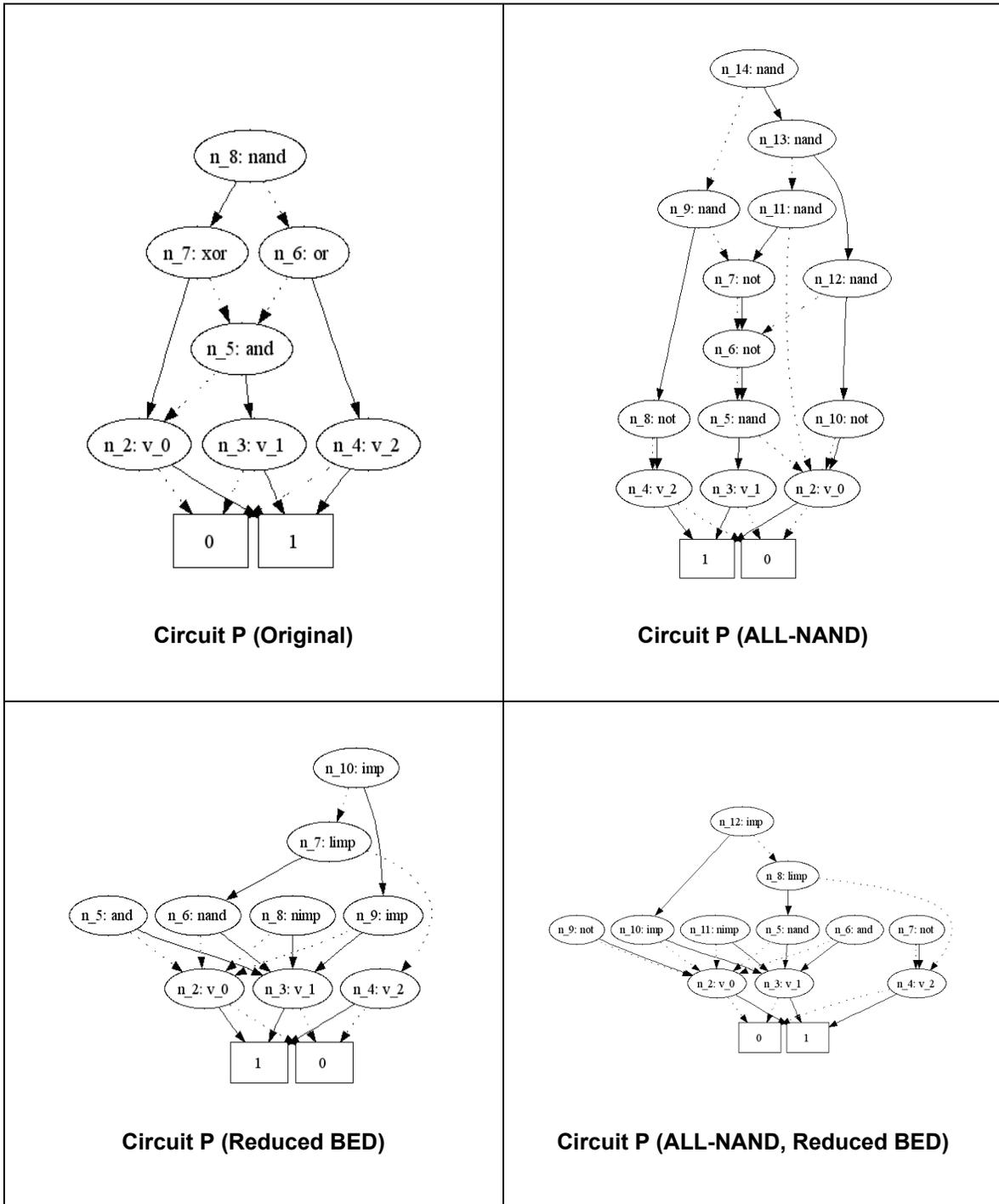


Figure 172: Example Circuit P in BED Notational Views

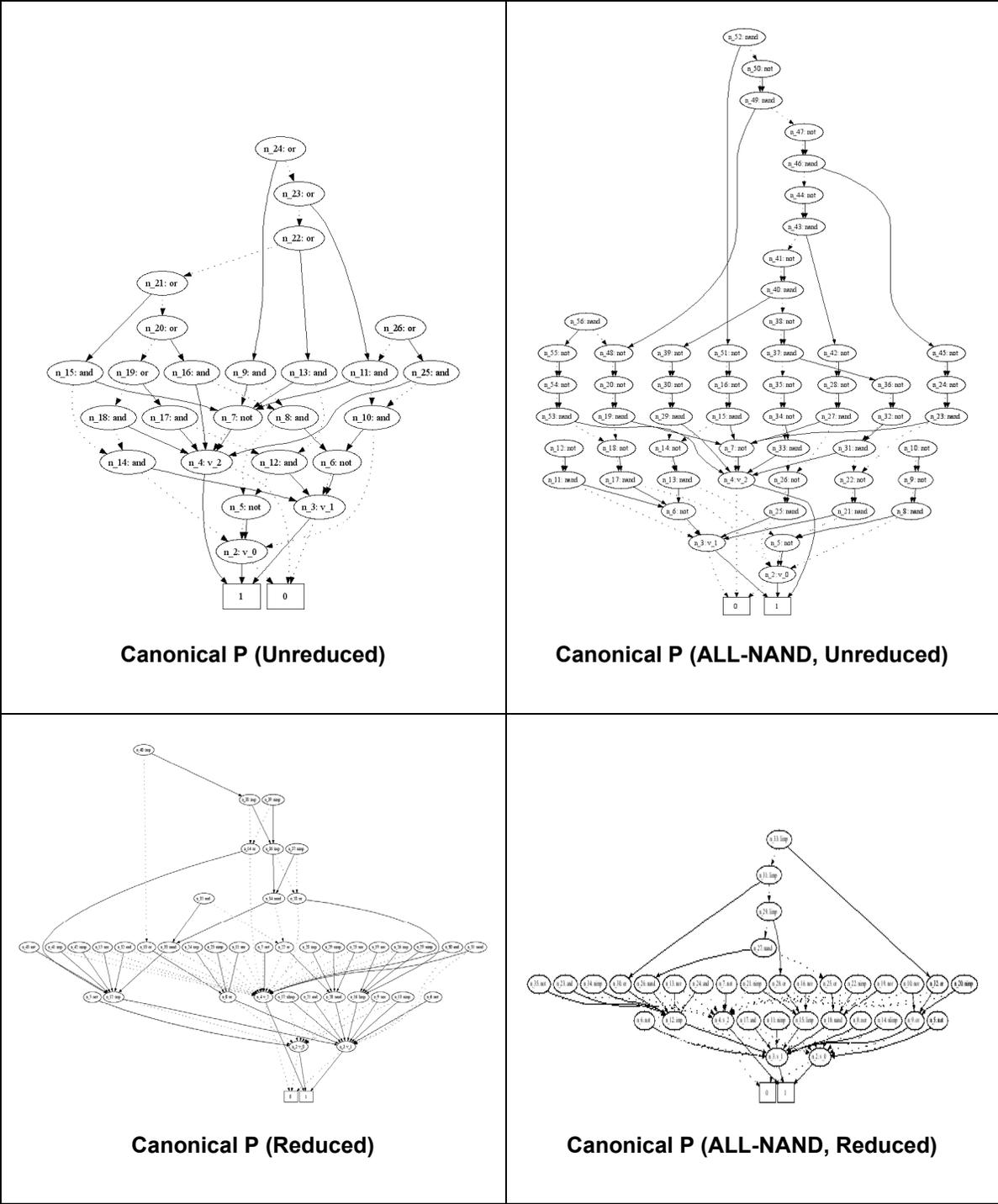


Figure 173: Example Circuit P (Canonical SOP) in BED Notational Views

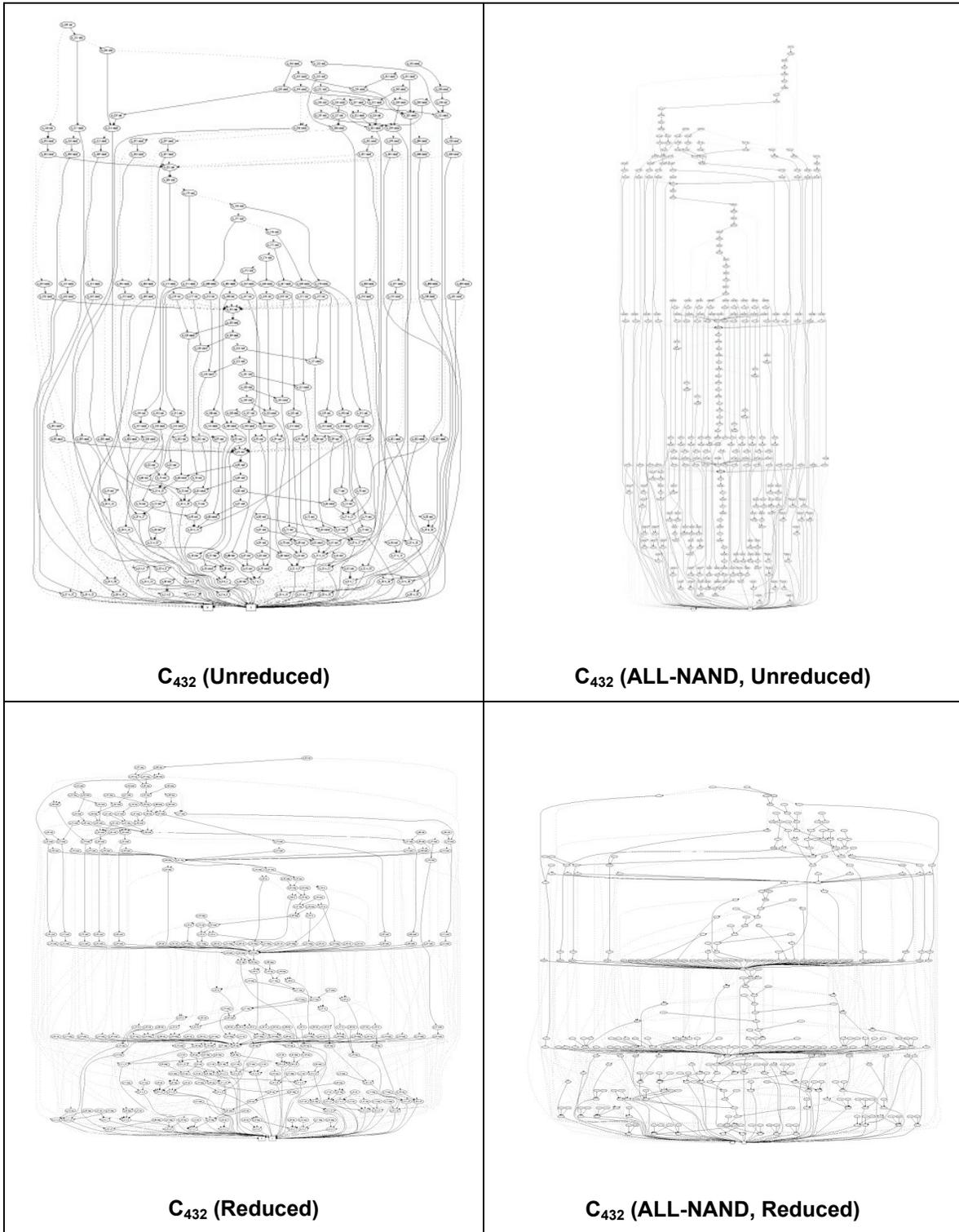


Figure 174: ISCAS C<sub>432</sub> in BED Notational Views

## REFERENCES

- [1] Bradshaw, J., Greaves, M., Holmback, H., Jansen, W., Karygiannis, T., Silverman, B.G., Suri, N., and Wong, A., "Agents for the Masses?" *IEEE Intelligent Systems*, 1999, no. March/April, pp. 53-63.
- [2] Bradshaw, J.M., "An Introduction to Software Agents", in *Software Agents*, 1997, AAAI Press, Menlo Park, Calif., USA: Cambridge, MA, pp. 3-46.
- [3] Jennings, N.R. and Wooldridge, M., "Applications of Intelligent Agents", in *Agent Technology: Foundations, Applications, and Markets*, Jennings, N. and Wooldridge, M.J. (eds.), 1998, Springer-Verlag: New York, NY, pp. 3-28.
- [4] Labrou, Y., Finin, T., and Peng, Y., "Agent Communication Languages: The Current Landscape", *IEEE Intelligent Systems*, 1999, vol. 14, no. 2, pp. 45-52.
- [5] Jennings, N.R., Sycara, K., and Wooldridge, M., "A Roadmap of Agent Research and Development", *Journal of Autonomous Agents and Multi-Agent Systems*, 1998, vol. 1, no. 1, pp. 7-38.
- [6] Picco, G.P., "Mobile Agents: An Introduction", *Journal of Microprocessors and Microsystems*, 2001, vol. 25, no. 2, pp. 65-74.
- [7] Ghezzi, C. and Vigna, G., "Mobile Code Paradigms and Technologies: A Case Study", in *Proceeding of the 1<sup>st</sup> International Workshop on Mobile Agents (MA '97)*, vol. 1219 of Lecture in Computer Science, 1997, Springer-Verlag, pp. 39-49.
- [8] Fuggetta, A., Picco, G.P., and Vigna, G., "Understanding Code Mobility", *IEEE Transactions on Software Engineering*, 1998, vol. 24, no. 5, pp. 342-361.
- [9] Carzaniga, A., Picco, G.P., and Vigna, G., "Designing Distributed Applications with a Mobile Code Paradigm", in *Proc. of the 19th International Conference on Software Engineering*, 1997, Boston MA, USA: ACM Press.
- [10] Gray, R.S., Cybenko, G., Kotz, D., Peterson, R.A., and Rus, D., "D'agents: Applications and Performance of a Mobile-Agent System", *Software--Practice and Experience*, 2002, vol. 32, no. 6, pp. 543-573.
- [11] Merwe, J. and Solms, S., "Electronic Commerce with Secure Intelligent Trade Agents", in *Proceedings of ICICS'97*, vol. 1334 of Lecture Notes in Computer Science, 1997, Springer-Verlag, pp. 452-462.
- [12] Kramer, K.H., Minar, N., and Maes, P., "Tutorial: Mobile Software Agents for Dynamic Routing", *Mobile Computing and Communications Review*, 1999, vol. 3, no. 2.
- [13] Baldi, M., Gai, S., and Picco, G.P., "Exploiting Code Mobility in Decentralized and Flexible Network Management", in Rothermel, K. and Popescu-Zeletin, R. (eds.), *Mobile Agents*, vol. 1219 of Lecture Notes in Computer Science, 1997, Springer, pp. 13-26.
- [14] Falchuk, B. and Karmouch, A., "Agentsys, a Mobile Agent System for Digital Media Access and Interaction on the Internet", in *Proc. of the IEEE GLOBECOM '97*, 1997, Phoenix, AZ.
- [15] Pozo, S., Gasca, R.M., and Gómez, M.T., "Securing Mobile Agent Based Tele-Assistance Systems", in *Proc. of the 1st International Workshop on Tele-Care and Collaborative Virtual Communities in Elderly Care (TELECARE 2004)*, 2004, Porto, Portugal.
- [16] Robles, S., Navarro, G., Pons, J., Rifà, J., and Borrell, J., "Mobile Agents Supporting Secure Grid Environments", in *Proc. of the Euroweb 2002 Conference*, 2002, Oxford, UK: British Computer Society, pp. 195-197.
- [17] Lubke, D. and Gomez, J.M., "Applications for Mobile Agents in Peer-to-Peer-Networks", in *Proc. of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)*, 2004.
- [18] Chess, D.M., Grosz, B., Harrison, C., Levine, D., Parris, C., and Tsudik, G., "Itinerant Agents for Mobile Computing", *Journal IEEE Personal Communications*, 1995, vol. 2, no. 5, pp. 34-49.
- [19] Samaras, G., "Mobile Agents: What About Them? Did They Deliver What They Promised? Are They Here to Stay?," in *Proc. of the IEEE International Conference on Mobile Data Management (MDM'04)*, 2004, Berkeley, California.
- [20] Roth, V., "Obstacle to the Adoption of Mobile Agents", In *Proc. of the IEEE International Conference on Mobile Data Management (MDM'04)*, 2004, Berkeley, CA.
- [21] Bierman, E. and Cloete, E., "Classification of Malicious Host Threats in Mobile Agent Computing", in *Proc. of the 2002 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology*, 2002, Port Elizabeth, South Africa.
- [22] Jansen, W. and Karygiannis, T., *NIST Special Publication 800-19: Mobile Agent Security*, National Institute of Standards and Technology, 2000.

- [23] Farmer, W.M., Guttman, J.D., and Swarup, V., "Security for Mobile Agents: Issues and Requirements", in *Proc. of the 19th National Information Systems Security Conference*, 1996, Baltimore, MD.
- [24] Chess, D.M., "Security Issues in Mobile Code Systems," in G. Vigna (ed.), *Mobile Agents and Security*, vol. 1419 of Lecture Notes in Computer Science, 1998, Springer-Verlag, p. 1-14.
- [25] Sander, T. and Tschudin, C.F., "Protecting Mobile Agents against Malicious Hosts", in Vigna, G. (ed.), *Mobile Agent Security*, vol. 1648 of Lecture Notes in Computer Science, 1998, Springer-Verlag, pp. 44-60.
- [26] Cartrysse, K. and van der Lubbe, J.C.A., "Providing Privacy to Agents in an Untrustworthy Environment", in *Handbook of Privacy and Privacy-Enhancing Technologies*, 2003, pp. 79-96.
- [27] Algesheimer, J., Cachin, C., Camenisch, J., and Karjoth, G., "Cryptographic Security for Mobile Code", in *Proc. of the IEEE Symposium on Security and Privacy*, 2001, pp. 2-11.
- [28] Riordan, J. and Schneier, B., "Environmental Key Generation Towards Clueless Agents", in Vigna, G. (ed.), *Mobile Agents and Security*, vol. 1419 of Lecture Notes in Computer Science, 1998, Springer-Verlag, pp. 15-24.
- [29] Borselius, N., Mitchell, C.J., and Wilson, A.T., "On Mobile Agent Based Transactions in Moderately Hostile Environments", in *Proc. of the Advances in Network and Distributed Systems Security, IFIP TC11 WG11.4 First Annual Working Conference on Network Security*, 2001, KU Leuven, Belgium: Kluwer Academic Publishers, Boston.
- [30] Kotzaniolaou, P., Burmester, M., and Chrissikopoulos, V., "Secure Transactions with Mobile Agents in Hostile Environments", in *Proc. of the 5<sup>th</sup> Australasian Conference on Information Security and Privacy*, 2000.
- [31] Yee, B., "A Sanctuary for Mobile Agents", in Vitek, J. and Jensen, C. (eds.), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, vol. 1603 of Lecture Notes in Computer Science, 1999, Springer-Verlag, pp. 261-274.
- [32] Wilhelm, U.G., Staamann, S., and Buttyán, L., "Introducing Trusted Third Parties to the Mobile Agent Paradigm", in *Secure Internet programming: Security Issues for Mobile and Distributed Objects*, vol. 1603 of Lecture Notes in Computer Science, 2001, Springer-Verlag, pp. 469 - 489.
- [33] Tan, H.K. and Moreau, L., "Extending Execution Tracing for Mobile Code Security", in *Proc. of the 2nd Intl Workshop on Security of Mobile MultiAgent Systems (SEMAS'2002)*, 2002, Bologna, Italy.
- [34] Hohl, F., "Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts", in Vigna, G. (ed.), *Mobile Agents and Security*, vol. 1419 of Lecture Notes in Computer Science, 1998, Springer, pp. 92-113.
- [35] Yahalom, R., Klein, B., and Beth, T., "Trust Relationships in Secure Systems-a Distributed Authentication Perspective", in *Proc. of the IEEE Symposium on Research in Security and Privacy*, 1993.
- [36] Blaze, M., Feigenbaum, J., and Lacy, J., "Decentralized Trust Management", in *Proc. of the IEEE Conference on Security and Privacy*, 1996, Oakland, CA.
- [37] Grandison, T. and Sloman, M., "A Survey of Trust in Internet Applications", *IEEE Communications Surveys & Tutorials*, 2000, vol. 4th Quarter.
- [38] Burmester, M. and Yasinsac, A., "Trust Infrastructures for Wireless, Mobile Networks", *WSEAS Transactions on Telecommunications*, 2004, vol. 3, no. 1, pp. 377-382.
- [39] Tan, H.K. and Moreau, L., "Trust Relationships in a Mobile Agent System", in Picco, G. (ed.), *Proceedings of the 5th IEEE International Conference on Mobile Agents*, vol. 2240 of Lecture Notes in Computer Science, 2001, Springer-Verlag, pp. 15-30.
- [40] Robles, S. and Borrell, J., "Trust in Mobile Agent Environments", in *Proc. of the 7<sup>th</sup> Spanish Meeting about Cryptology and Information Security*, 2002, Oviedo.
- [41] Kagal, L., Finin, T., and Joshi, A., "Developing Secure Agent Systems Using Delegation Based Trust Management", in *Proc. of the 2nd International Workshop on Security of Mobile MultiAgent Systems (SEMAS'2002)*, 2002, Bologna, Italy.
- [42] Lin, C., Varadharajan, V., Wang, Y., and Mu, Y., "On the Design of a New Trust Model for Mobile Agent Security", in *1<sup>st</sup> International Conference on Trust and Privacy in Digital Business (TrustBus'04)*, Zaragoza, Spain, vol. 3184 of Lecture Notes in Computer Science, 2004, Springer-Verlag, pp. 60-69.
- [43] Zachary, J. and Brooks, R., "Bidirectional Mobile Code Trust Management Using Tamper Resistant Hardware", *Mobile Networks and Applications*, 2003, vol. 8, no. 2, pp. 137-143.
- [44] McDonald, J.T. and Yasinsac, A., "Of Unicorns and Random Programs", To appear in *Proc. of the 3rd IASTED International Conference on Communications and Computer Networks (IASTED/CCN)*, 2005, Marina del Rey, CA.

- [45] Odell, J., "Objects and Agents Compared", *Journal of Object Technology*, 2002, vol. 1, no. 1, pp. 41-53.
- [46] Lange, D., "Mobile Objects and Mobile Agents: The Future of Distributed Computing?" in Jul, E. (ed.), *Proceedings of ECOOP'98*, vol. 1445 of Lecture Notes in Computer Science, 1998, Springer-Verlag, pp. 1-12.
- [47] Franklin, S. and Graesser, A., "Is It an Agent, or Just a Program? A Taxonomy for Autonomous Agents", in *Proc. of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, 1996, pp. 21-35.
- [48] Cubillos, C.F. and Guidi-Polanco, F., "Security Issues on Agent-Based Technologies", in *Proc. of the VIP Scientific Forum of the International IPSI-2003 Conference*, 2003.
- [49] Rao, A. and Georgeff, M., "Bdi Agents: From Theory to Practice", in *Proc. of the 1st International Conference on MAS (ICMAS' 95)*, 1995, San Francisco, CA.
- [50] Decker, K., Williamson, M., and Sycara, K., "Matchmaking and Brokering", in *Proc. of the 2nd International Conference on Multi-Agent Systems (ICMAS-96)*, 1996, pp. 432.
- [51] Object-Management-Group, "Agent Technology Green Paper", *Agent Working Group, OMG Document ec/2000-03-01, Version 1.0*. 2000.
- [52] Tomic, P.T. and Agha, G.A., "Towards a Hierarchical Taxonomy of Autonomous Agents", in *Proc. of the IEEE International Conference on Systems, Man and Cybernetics*, 2004, The Hague, The Netherlands.
- [53] Wooldridge, M. and Jennings, N.R., "Intelligent Agents: Theory and Practice", *The Knowledge Engineering Review*, 1995, vol. 10, no. 2, pp. 115-152.
- [54] Jennings, N.R., "Building Complex, Distributed Systems: The Case for an Agent-Based Approach", *Communications of the ACM*, 2001, vol. 4, no. 4, pp. 35-41.
- [55] Weiss, G., "Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence", 1999, Cambridge, MA: The MIT Press.
- [56] Gilbert, D., Aparicio, M., Atkinson, B., Brady, S., Ciccarino, J., Grosz, B., O'Connor, P., Osisek, D., Pritko, S., Spagna, R. and Wilson, L., "IBM Intelligent Agent Strategy", White Paper, 1995.
- [57] Rothermel, K. and Schwehr, M., "Mobile Agents", *Encyclopedia for Computer Science and Technology*, Kent, A. and Williams, J.G. (eds.), 1998, New York: M. Dekker Inc.
- [58] Lange, D. and Oshima, M., "Seven Good Reasons for Mobile Agents", *Communications of the ACM*, 1999, vol. 42, no. 3, pp. 88-89.
- [59] Kotz, D. and Gray, R.S., "Mobile Agents and the Future of the Internet", *ACM Operating Systems Review*, 1999, vol. 33, no. 3, pp. 7-13.
- [60] Chess, D.M., Harrison, C.G., and Kershenbaum, A., "Mobile Agents: Are They a Good Idea?" in Vitek, J. and Tschudin, C.F. (eds.), *Mobile Object Systems: Towards the Programmable Internet*, vol. 1222 of Lecture Notes in Computer Science, 1997, Springer-Verlag, pp. 46-48.
- [61] Riordan, J. and Schneier, B., "Environmental Key Generation Towards Clueless Agents", in Vigna, G. (ed.), *Mobile Agents and Security*, vol. 1419 of Lecture Notes in Computer Science, 1998, Springer-Verlag, pp. 15-24.
- [62] Milojevic, D., "Mobile Agent Applications", *IEEE Concurrency*, 1999, vol. 7, no. 3, pp. 80-90.
- [63] Hohl, F., "A Framework to Protect Mobile Agents by Using Reference States", in *Proc. of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, 2000.
- [64] Bellavista, P., Corradi, A., Federici, C., Montanari, R., and Tibaldi, D., "Security for Mobile Agents: Issues and Challenges", in *Handbook of Mobile Computing*, 2004.
- [65] Schoeman, M. and Cloete, E., "Architectural Components for the Efficient Design of Mobile Agent Systems", in *Proc. of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology*, 2003: South African Institute for Computer Scientists and Information Technologists, pp. 48-58.
- [66] Altmann, J., Gruber, F., Klug, L., Stockner, W., and Weippl, E., "Using Mobile Agents in Real World: A Survey and Evaluation of Agent Platforms", in *Proc. of the 2<sup>nd</sup> International Workshop on Infrastructure for Agents, MAS, and Scalable MAS at the 5th International Conference on Autonomous Agents*, 2001, Montreal, Canada: ACM Press.
- [67] Wong, D., Paciorek, N., and Moore, D., "Java-Based Mobile Agents", *Communications of the ACM* 42(3), 1999, pp. 92.
- [68] Fuggetta, A., Picco, G.P., and Vigna, G., "Understanding Code Mobility", *IEEE Transactions on Software Engineering*, 1998, vol. 24, no. 5, pp. 342-361.
- [69] Rothermel, K., Hohl, F., and Radouniklis, N., "Mobile Agent Systems: What Is Missing?" in *Proc. of the International Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)*, 1997, Cottbus, Germany, pp. 111-124.

- [70] Thorn, T., "Programming Languages for Mobile Code", *ACM Computing Surveys*, 1997, vol. 29, no. 3, pp. 213-239.
- [71] Vitek, J. and Castagna, G., "Towards a Calculus of Secure Mobile Computations", in *Proc. of the Workshop on Internet Programming Languages*, 1998, Chicago, IL.
- [72] Serugendo, G., Muhugasa, M., and Tschudin, C., "A Survey of Theories for Mobile Agents", *World Wide Web*, 1998, vol. 1, no. 3, pp. 139-153.
- [73] Borselius, N. and Mitchell, C.J., "Securing FIPA Agent Communication", in *Proc. of the 2003 International Conference on Security and Management (SAM'03)*, 2003, Las Vegas, Nevada, USA: CSREA Press, pp. 135-141.
- [74] Yee, B., "Monotonicity and Partial Results Protection for Mobile Agents", in *Proc. of the 23rd International Conference on Distributed Computing Systems*, 2003, Providence, Rhode Island.
- [75] Ordille, J.J., "When Agents Roam, Who Can You Trust?" in *Proc. of the First Annual Conference on Emerging Technologies and Applications in Communications*, 1996.
- [76] Wilhelm, U.G. and Staamann, S., "Protecting the Itinerary of Mobile Agents", in *Proc. of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations*, 1998, pp. 135-145.
- [77] Borrell, J., Robles, S., Serra, J., and Riera, A., "Securing the Itinerary of Mobile Agents through a Non-Repudiation Protocol", in *Proc. of the 33<sup>rd</sup> Annual 1999 International Carnahan Conference on Security Technology*, 1999: IEEE Press, pp. 461-464.
- [78] Chen, P. and Chang, S., "An Itinerary-Diagram-Based Approach for Mobile Agent Application Development", *Tamkang Journal of Science and Engineering*, 2000, vol. 3, no. 3, pp. 209-227.
- [79] Knoll, G., Suri, N., and Bradshaw, J.M., "Path-Based Security for Mobile Agents", in *Proc. of the 1<sup>st</sup> International Workshop on Security of Mobile Multiagent (SEMAS 2001)*, Electronic Notes in Theoretical Computer Science, vol. 63, 2001, Montreal, Canada: Elsevier, pp. 1-16.
- [80] Aridor, Y. and Lange, D.B., "Agent Design Patterns: Elements of Agent Application Design", in *Proc. of the Second International Conference on Autonomous Agents (AGENTS'98)*, 1998: ACM Press, pp. 108-115.
- [81] Tahara, Y., Ohsuga, A., and S., H., "Agent System Development Method Based on Agent Patterns", in *Proc. of the IEEE International Conference on Software Engineering (ICSE'99)*, 1999: IEEE Computer Society, pp. 356-367.
- [82] Satoh, I., "Selection of Mobile Agents", in *Proc. of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'2004)*, 2004: IEEE Computer Society, pp. 484-493.
- [83] Milner, R., Parrow, J., and Walker, D., "A Calculus of Mobile Processes, Part I and II", *Journal of Information and Computation*, 1992, vol. 100, no. 1, pp. 1-77.
- [84] Milner, R., "Communicating and Mobile Systems: The  $\pi$ -Calculus", 1999: Cambridge University Press.
- [85] Serugendo, G., Muhugasa, M., and Tschudin, C., "A Survey of Theories for Mobile Agents", *World Wide Web*, 1998, vol. 1, no. 3, pp. 139-153.
- [86] Honda, K. and Tokoro, M., "An Object Calculus for Asynchronous Communication", in America, P. (ed.), *Proceedings of the ECOOP '91*, vol. 512 of Lecture Notes in Computer Science, 1991, Springer-Verlag, pp. 133-147.
- [87] Riely, J. and Hennessy, M., "Distributed Processes and Location Failures", in Verlag, S. (ed.), *ICALP '97*, vol. 1256 of Lecture Notes in Computer Science, 1997, pp. 471-481.
- [88] Hennessy, M. and Riely, J., "Resource Access Control in Systems of Mobile Agents", *Information and Computation*, 2002, vol. 173, no. 1, pp. 82-120.
- [89] Swell, P., Wojciechowski, P.T., and Pierce, B., "Location-Independent Communication for Mobile Agents: A Two-Level Architecture", in *Workshop on Internet Programming Languages*, vol. 1686 of Lecture Notes in Computer Science, 1998, Springer.
- [90] Abadi, M. and Gordon, A.D., "A Calculus for Cryptographic Protocols: The Spi Calculus", *Information and Computation*, 1999, vol. 149, no. 1, pp. 1-70.
- [91] Fournet, C., Gonthier, G., Levy, J., Marnaget, L., and Remy, D., "A Calculus of Mobile Agents", in *Proceedings of CONCUR '96*, vol. 1119 of Lecture Notes in Computer Science, 1996, Springer, pp. 406-421.
- [92] Cardelli, L. and Gordon, A., "Mobile Ambients", in *Proceedings of Foundations of Software Science and Computer Structures*, vol. 1378 of Lecture Notes in Computer Science, 1998, pp. 140-155.
- [93] Chandy, K. and Misra, J., "Parallel Program Design: A Foundation", 1988, Boston, MA: Addison-Wesley.
- [94] Picco, G., Roman, G., and McCann, P.J., "Reasoning About Code Mobility with Mobile Unity", *ACM Transactions on Software Engineering and Methodology*, 2001, vol. 10, no. 3, pp. 338 - 395.

- [95] McCann, P.J. and Roman, G., "Compositional Programming Abstractions for Mobile Computing", *IEEE Transaction on Software Engineering*, 1998, vol. 24, no. 2, pp. 97-110.
- [96] Vitek, J. and Castagna, G., "Seal: A Framework for Secure Mobile Computations", in *Internet Programming Language (ICCL'98) Workshop*, vol. 1686 of Lecture Notes in Computer Science, 1999, Springer, pp. 47-77.
- [97] Blanchet, B. and Aziz, B., "A Calculus for Secure Mobility", in *Proc. of the 8<sup>th</sup> Asian Computing Science Conference (ASIAN'03)*, 2003, Mumbai, India.
- [98] Heintz, N. and Riecke, J., "The SLAM Calculus: Programming with Secrecy and Integrity", in *Proc. of the Conference Record of the ACM Symposium on Principles of Programming Languages*, 1998, San Diego: ACM Press.
- [99] Foundation for Intelligent Physical Agents, *FIPA00067, FIPA Agent Message Transport Service Specification*, 2000. Available: <http://www.fipa.org/repository>.
- [100] Baumann, J. and Radouniklis, N., "Agent Groups in Mobile Agent Systems", in *Proceedings of the International Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)*, 1997.
- [101] Poslad, S., Calisti, M., and Charlton, P., "Specifying Standard Security Mechanisms in Multi-Agent Systems", in *Proceedings of the AAMAS 2002 Workshop on Deception, Fraud And Trust*, 2002, Bologna, Italy.
- [102] Thirunavukkarasu, C., Finlin, T., and Mayfield, J., "Secret Agents - a Security Architecture for Kqml Agent Communication Languages", in *Proc. of the Intelligent Information Agents Wrkshp (CIKM'95)*, 1995, Baltimore, MD.
- [103] Tan, J., Titkov, L., and Neophytou, C., "Securing Multi-Agent Platform Communication", in *Proc. of the Working Notes Second International Workshop on Security of Mobile Multiagent Systems*, 2002, pp. 66-72.
- [104] Labrou, Y., Finin, T., and Peng, Y., "The Interoperability Problem: Bringing Together Mobile Agents and Agent Communication Languages. " in *Proc. of the Hawaii International Conference On System Sciences*, 1999, Maui, Hawaii, pp. 45-52.
- [105] Cachin, C., Camenisch, J., Kilian, J., and Müller, J., "One-Round Secure Computation and Secure Autonomous Mobile Agents", in Montanari, U., Rolim, J.P., and Welzl, E. (eds.), *Proceedings of the 27th International Colloquium on Automata Languages and Programming (ICALP)*, vol. 1853 of Lecture Notes in Computer Science, 2000, Springer-Verlag, pp. 512-523.
- [106] Hohl, F., "A Model of Attacks of Malicious Hosts against Mobile Agents", in *Proc. of the 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations*, 1998, France.
- [107] Maggi, P. and Sisto, R., "A Configurable Mobile Agent Data Protection Protocol." in *Proc. of the AAMAS'03*, 2003, Melbourne, Australia.
- [108] Roth, V., "On the Robustness of Some Cryptographic Protocols for Mobile Agent Protection", in *Mobile Agents*, vol. 2240 of Lecture Notes in Computer Science, 2001, Springer-Verlag.
- [109] Roth, V., "Empowering Mobile Software Agents", in *Proc. of the 6th IEEE Mobile Agents Conference*, vol. 2535 of Lecture Notes in Computer Science, 2002, Springer-Verlag, pp. 47-63.
- [110] Loureiro, S. and Molva, R., "Mobile Code Protection with Smartcards", in *Proc. of the ECOOP 2000 Workshop on Mobile Object Systems*, 2000, Cannes, France.
- [111] Sander, T. and Tschudin, C., "Towards Mobile Cryptography", in *Proc. of the IEEE Symposium on Security and Privacy*, 1998.
- [112] Roth, V., "Programming Satan's Agents", in *Proc. of the 1st International Workshop on Secure Mobile Multi-Agent Systems*, 2001, Montreal, Canada.
- [113] Richards, M., "The State of Security Standards for Mobile Agents", in *Proc. of the Decision Sciences Institute 2002 Annual Meeting*, 2002, pp. 268-272.
- [114] Poslad, S. and Calisti, M., "Towards Improved Trust and Security in FIPA Agent Platforms", in *Autonomous Agents*, 2000: Barcelona, Spain.
- [115] Poslad, S., Calisti, M., and Charlton, P., "Specifying Standard Security Mechanisms in Multi-Agent Systems", in *Proc. of the AAMAS 2002 Workshop on Deception, Fraud And Trust*, 2002, Bologna, Italy.
- [116] Poslad, S., Buckle, P., and Hadingham, R., "The FIPA OS Agent Platform: Open Source for Open Standards", in *Proc. of the 5th Int'l Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, 2000, Manchester, UK, pp. 355-368.
- [117] Poslad, S. and Calisti, M., "Standardizing Agent Interoperability: The FIPA Approach", in *Proc. of the Autonomous Agents 2000*, Lecture Notes in Artificial Intelligence: Multi-Agent Systems and Applications, 2001: Springer-Verlag, pp. 98-117.

- [118] Foundation for Intelligent Physical Agents, *FIPA 98 Part 10, Version 1.0: Agent Security Management Specification (Obsolete)*, 1998. Available: <http://www.fipa.org/repository/obsoletespecs.html>.
- [119] Foundation for Intelligent Physical Agents, *FIPA Abstract Architecture Specification, Version L*, 2004. Available: <http://www.fipa.org/repository/architecturespecs.html>.
- [120] Foundation for Intelligent Physical Agents, *FIPA00023, FIPA Agent Management Specification*, 2000. Available: <http://www.fipa.org/repository>.
- [121] Roth, V., Pinsdorf, U., and Binder, W., "Mobile Agent Interoperability Revisited", in *Proc. of the 5th IEEE International Conference on Mobile Agents*, 2001, Atlanta, Georgia.
- [122] Pogg, A., Rimassa, G., and Tomaiuolo, M., "Multi-User and Security Support for Multi-Agent Systems", in *Proc. of the Workshop from Objects to Agents (WOA 2001)*, 2001, Modena, Italy.
- [123] Zhang, M., Karmouch, A., and Impey, R., "Towards a Secure Agent Platform Based on FIPA", in *Proceedings of International Conference on Mobile Agents for Telecommunication Applications (MATA 2001)*, vol. 2164 of Lecture Notes in Computer Science, 2001, Springer-Verlag, pp. 277-289.
- [124] Schoder, D. and Eymann, T., "The Real Challenges of Mobile Agents", *Communications of the ACM* 43(6), 2000, pp. 111-112.
- [125] Tschudin, C., "Chapter 18: Mobile Agent Security", *Intelligent Information Agents: Agent-Based Information Discovery and Management on the Internet*, Klusch, M. (ed.), 1999, Springer-Verlag, 431-445.
- [126] Vigna, G., "Mobile Agents: Ten Reasons for Failure", in *Proc. of the IEEE International Conference on Mobile Data Management (MDM'04)*, 2004, Berkeley, CA: IEEE Computer Society, pp. 298-299.
- [127] Johansen, D., "Mobile Agents: Right Concept, Wrong Approach", in *Proc. of the IEEE International Conference on Mobile Data Management (MDM'04)*, 2004, Berkeley, CA: IEEE Computer Society.
- [128] Wong, H.C. and Sycara, K., "Adding Security and Trust to Multi-Agent Systems", in *Proc. of the Autonomous Agents '99 (Workshop on Deception, Fraud and Trust in Agent Societies)*, 1999, Seattle, WA, pp. 149-161.
- [129] Borselius, N., "Mobile Agent Security", *Electronics & Communication Engineering Journal* 14(5), 2002, pp. 211-218.
- [130] Borselius, N., "Security in Multi-Agent Systems", in *Proc. of the 2002 International Conference on Security and Management (SAM'02)*, 2002, Las Vegas, NV, USA: CSREA Press, pp. 31-36.
- [131] Wells, D., Pazandak, P., Nodine, M., and Cassandra, A., "Adaptive Defense Coordination in Multi-Agent Systems", in *Proc. of the 1st IEEE Symposium on Multi-Agents Security and Survivability (MASS'04)*, 2004, Philadelphia, PA.
- [132] Bresciani, P., Giorgini, P., Mouratidis, H., and Manson, G., "Multi-Agent Systems and Security Requirements Analysis", in Lucena, C., et al. (eds.), *Advances in Software Engineering for Multi-Agent Systems*, vol. 2940 of Lecture Notes in Computer Science, 2004, Springer-Verlag.
- [133] Mouratidis, H., Giorgini, P., and Manson, G., "Modeling Secure Multiagent Systems", in *Proc. of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems*, 2003, Melbourne, Australia, pp. 859 - 866.
- [134] Braynov, S. and Jadliwala, M., "Detecting Malicious Groups of Agents", in *Proc. of the 1st IEEE Symposium on Multi-Agents Security and Survivability (MASS'04)*, 2004, Philadelphia, PA.
- [135] Parks, R., Jung, R., and Ramotowski, K., "Attacking Agent Based Systems", in *Proc. of the 1st IEEE Symposium on Multi-Agents Security and Survivability (MASS'04)*, 2004, Philadelphia, PA.
- [136] Orso, A., Harrold, M., and Vigna, G., "MASSA: Mobile Agents Security through Static/Dynamic Analysis", in *Proc. of the ICSE Wrkshp on Software Engineering and Mobility*, 2001, Toronto, Ontario.
- [137] Zachary, J., "Protecting Mobile Code in the Wild", *IEEE Internet Computing* 7(2), 2003, pp. 2-6.
- [138] Wilhelm, U., Staamann, S., and Buttyan, L., "On the Problem of Trust in Mobile Agent Systems", in *Proc. of the IEEE Network and Distributed Systems Security Symposium*, 1999, San Diego, CA, pp. 11-13.
- [139] Swarup, V., "Trust Appraisal and Secure Routing of Mobile Agents", in *Proc. of the DARPA Workshop on Foundations for Secure Mobile Code*, 1997.
- [140] Reiser, H. and Vogt, G., "Threat Analysis and Security Architecture of Mobile Agent Based Management Systems", in *Proc. of the Network Operations and Management Symposium*, 2000, Honolulu, Hawaii.
- [141] Dobrev, S., Flocchini, P., Prencipe, G., and Santoro, N., "Finding a Black Hole in an Arbitrary Network: Optimal Mobile Agents Protocols", in *Proc. of the 21st ACM Symp. on Princ. of Distr. Computing (PODC 2002)*, 2002, pp. 153-162.

- [142] McDonald, J.T., Yasinsac, A. and Thompson, W., "Mobile Agent Data Integrity Using Multi-agent Architecture," in *Proc. of the Int'l Workshop on Security in Parallel and Distributed Systems (PDCS 2004)*, San Francisco, CA, 2004.
- [143] McDonald, J.T., "Hybrid Approach for Secure Mobile Agent Computations," in *Proc. of the Secure Mobile Ad-hoc Networks and Sensors Workshop (MADNES '05)*, vol. 4074 of Lecture Notes in Computer Science, 2005, Springer-Verlag, pp. 38-53.
- [144] Karjoth, G., Asokan, N., and Gulcu, C., "Protecting the Computation Results of Freeroaming Agents", in Rothermel, K. and Hohl, F. (eds.), in *Proc. of the 2nd International Workshop, Mobile Agents 98*, vol. 1477 of Lecture Notes in Computer Science, 1998, Springer-Verlag, pp. 195-207.
- [145] Villate, Y., Illarramendi, A., and Pitoura, E., "Data Lockers: Mobile-Agent Based Middleware for the Security and Availability of Roaming Users Data", in *Proc. of the 7th Int'l Conf. on Cooperative Information Systems (CoopIS 2000)*, vol. 1901 of Lecture Notes in Computer Science, 2000, Springer, pp. 275-286.
- [146] Chow, S., Eisen, P., Johnson, H., and van Oorschot, P. C., "A White-box DES Implementation for DRM Applications," *Proc. of the 2nd ACM Workshop on Digital Rights Management (DRM 2002)*, vol. 2696 of Lecture Notes in Computer Science, 2003, pp. 1-15.
- [147] Chow, S., Eisen, P., Johnson, H., and van Oorschot, P. C., "White-Box Cryptography and an AES Implementation", in *Proc. of the Ninth Workshop on Selected Areas in Cryptography (SAC 2002)*.
- [148] Schneider, F., "Towards Fault Tolerant and Secure Agency," in *Proc. of the 11<sup>th</sup> Int. Workshop on Distributed Algorithms*, vol. 1320 of Lecture Notes in Computer Science, Springer-Verlag, Berlin Germany, 1997.
- [149] H. Vogler, T. Hunkleemann and M. Moschgath, "An Approach for Mobile Agent Security and Fault Tolerance Using Distributed Transactions," in *Proc. Int'l Conference on Parallel and Distributed Systems (ICPADS'97)*, , Seoul, December 1997, pp.268-274.
- [150] Feigenbaum, J., Pinkas, B., Ryger, R., and Saint Jean, F., "Secure Computation of Surveys", in *Proc. of the EU Workshop on Secure Multiparty Protocols*, 2004.
- [151] McDonald, J.T. and Yasinsac, A., "Application Security Models for Mobile Agent Systems," in *Proc. of the 1st Int'l Workshop on Security and Trust Management*, Milan, Italy, 2005, Electronic Notes in Theoretical Computer Science, vol. 157, no. 3, 25 May 2006, pp. 43-59.
- [152] McDonald, J.T. and Yasinsac, A., "Trust in Mobile Agent Systems", *Technical Report TR-050330*, Dept. of Computer Science, Florida State University, March 2005, available <http://www.cs.fsu.edu/research/reports/TR-050330.pdf>.
- [153] Kalogridis, G., Mitchell, C. J., and Clemo, G., "Spy Agents: Evaluating Trust in Remote Environments," in *Proc. of the Intl Conf. on Security and Management*, Las Vegas, NV, 2005.
- [154] Yasinsac, A. and McDonald, J.T., "Foundations for Security Aware Software Development Education," in *Proc. of the 39th Annual Hawaii Int'l Conference on System Sciences (HICSS'06)*, 2006, p. 219.
- [155] Thompson, W., Yasinsac, A., and McDonald, J. T., "Semantic Encryption Transformation Scheme," in *Proc. of the Int'l Workshop on Security in Parallel and Dist. Systems (PDCS 2004)*, San Francisco, CA, 2004.
- [156] McDonald, J.T. and Yasinsac, A., "Program Intent Protection Using Circuit Encryption," to appear, in *Proc. of 8<sup>th</sup> Int'l Symposium on Systems and Information Security*, Sao Paulo, Brazil, Nov. 8-10, 2006.
- [157] Yasinsac, A. and McDonald, J.T., "Tamper Resistant Software through Intent Protection," unpublished manuscript.
- [158] McDonald, J.T. and Yasinsac, A., "On the Possibility of Perfectly Secure Obfuscation for Bounded Input-Size Programs," submitted to *4th Theory of Cryptography Conference (TCC'07)*, Feb. 21-24, 2007, Amsterdam, The Netherlands (decision October 2006).
- [159] Sander, T. and Tschudin, C.F., "On Software Protection Via Function Hiding", in *Proc. of the Second International Workshop on Information Hiding*, vol. 1525 of Lecture Notes in Computer Science, 1998, pp. 111-123.
- [160] Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M., "Architectural Support for Copy and Tamper Resistant Software", in *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOSIX)*, 2000, pp. 169-177.
- [161] Lie, D., Mitchell, J., Thekkath, C.A., and Horowitz, M., "Specifying and Verifying Hardware for Tamper-Resistant Software", in *Proc. of the IEEE Symposium on Security and Privacy*, 2003, Berkeley, CA., pp. 166.

- [162] Ogiso, T., Sakabe, Y., Soshi, M., and Miyaji, A., "Software Tamper Resistance Based on the Difficulty of Interprocedural Analysis", in *Proc. of the 3rd International Workshop on Information Security Applications (WISA 2002)*, 2002, pp. 437-452.
- [163] Wang, C., Hill, J., Knight, J., and Davidson, J., "Software Tamper Resistance: Obstructing Static Analysis of Programs", *Technical Report CS-2000-12*, University of Virginia, 2000.
- [164] Aucsmith, D., "Tamper-resistant Software: An Implementation," *Proc. of the 1st Int'l Workshop on Information Hiding*, vol. 1174 of *Lecture Notes in Computer Science*, pp. 317-333. London, UK: Springer-Verlag, 1996,
- [165] Wang, C., "A Security Architecture for Survivability Mechanisms," PhD thesis, Department of Computer Science, University of Virginia, 2000.
- [166] Palsberg, J., Krishnaswamy, S., Minseok, K., Ma, D., Shao, Q., and Zhang, Y., "Experience with Software Watermarking," in *Proc. of the 16th Annual Computer Security Applications Conference, ACSAC '00*, IEEE, 2000, pp. 308-316.
- [167] Collberg, C. and Thomborson, C., "Software Watermarking: Models and Dynamic Wmbeddings," in *Principles of Programming Languages 1999*, POPL'99, Jan. 1999.
- [168] D'Anna, L., Matt, B., Reisse, A., Vleck, T.V., Schwab, S., and LeBlanc, P., "Self-Protecting Mobile Agents Obfuscation Report", Technical Report #03-015, Network Associates Labs. 2003.
- [169] Collberg, C., Thomborson, C., and Low, D., "Breaking Abstractions and Unstructuring Data Structures", in *Proc. of the IEEE International Conf. Computer Languages (ICCL'98)*, 1998.
- [170] Collberg, C., Thomborson, C., and Low, D., "A Taxonomy of Obfuscating Transformations", Technical Report, Department of Computer Science, University of Auckland, New Zealand, 1997.
- [171] Ostrovsky, R. and Skeith, W., "Private Searching on Streaming Data," *CRYPTO '2005*, 2005.
- [172] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S. P., and Yang, K., "On the (Im)possibility of Obfuscating Programs," in *Proc. of CRYPTO '01*, J. Kilian, Ed. Santa Barbara, California: Springer-Verlag, vol. 2139 of *Lecture Notes in Computer Science*, Aug. 19-23 2001, pp. 1-18.
- [173] Goldwasser, S. and Kalai, Y., "On the Impossibility of Obfuscation with Auxiliary Input," in *Proc. of the 46th Annual IEEE Symp. on Foundations of Computer Science*, 2005.
- [174] Appel, A., "Deobfuscation is in NP," unpublished manuscript, preprint available from <http://www.cs.princeton.edu/~appel/papers/deobfus.pdf>, 2002.
- [175] Varnovsky, N. and Zakharov, V., "On the Possibility of Provably Secure Obfuscating Programs," *Perspectives of System Informatics*, vol. 2890 of *Lecture Notes in Computer Science*, pp. 91-102, 2003.
- [176] Lynn, B., Prabhakaran, M., and Sahai, A., "Positive Results and Techniques for Obfuscation," *EUROCRYPT'04*, 2004.
- [177] Canetti, R., "Towards Realizing Random Oracle: Hash Functions that Hide All Partial Information," *CRYPTO'97*, pp. 455-469.
- [178] Wee, H., "On Obfuscating Point Functions," in *Proc. of ACM STOC'05*, pp. 523-532, May 22-24, 2005.
- [179] Chow, S., Gu, Y., Johnson, H. and Zakharov, V.A., "An Approach to the Obfuscation of Control-Fow of Sequential Computer Programs," in *Proc. of ISC 2001*, vol. 2200 of *Lecture Notes in Computer Science*, pp. 144-155, 2001.
- [180] Yu, Y., Leiwo, J., and Premkumar, B., "Hiding Circuit Topology From Unbounded Reverse Engineers," in L. Batten and R. Safavai-Naini (Eds.): in *Proc. of ACISP 2006*, vol. 4058 of *Lecture Notes in Computer Science*, pp. 171-182, 2006.
- [181] Yu, Y., Leiwo, J., and Premkumar, B., "Securely Utilizing External Computing Power," in *Proc. of the IEEE Int'l Conf. on Information Technology: Coding and Computing (ITCC'05)*, 2005.
- [182] Ogiso, T., Sakabe, Y., Soshi, M., and Miyaji, A., "Software Obfuscation on a Theoretical Basis and its implementation," *IEICE Trans. Fundamentals*, vol E86-A, no. 1, January 2003.
- [183] Collberg, C.S. and Thomborson, C., "Watermarking, tamper-proofing, & obfuscation - tools for software protection," *IEEE Trans. on Software Engin.*, vol. 28, pp. 735-746, 2002.
- [184] Drape, S., "Obfuscation of abstract data types," Doctoral thesis, Department of Computer Science, St. John's College, University of Oxford, UK.
- [185] Collberg, C. S., Thomborson, C., and Low, D., "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," in *Proc. of the 25th ACM Symposium on Principles of Programming (POPL1998)*, 1998.
- [186] Wang, C., "A Security Architecture for Survivability Mechanisms," PhD thesis, Department of Computer Science, University of Virginia, 2000.
- [187] Diffie, W. and Hellman, M.E., "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644-654.

- [188] Goldreich, O. *Foundations of Cryptography*. Cambridge University Press, 2001.
- [189] McCabe, T., "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.
- [190] Harrison, W. and Magel, K., "A Complexity Measure Based on Nesting Level." *SIGPLAN Notices*, vol. 16, no. 3, pp. 63–74, 1981.
- [191] Munson, J. and Khoshgoftaar, T., "Measurement of Data Structure Complexity," *Journal of Systems Software*, vol. 20, no. 3, pp. 217–225, 1993.
- [192] Herzberg, A. and Pinter, S., "Public Protection of Software," *ACM Trans. Comput. Syst.*, vol. 5, no. 4, pp. 371–393, 1987.
- [193] MacBride, J., Mascioli, C., Marks, S., Tang, Y., Head, L., and Ramachandran, R.P., "A Comparative Study of Java Obfuscators," in Tsai, W. and Hamza, M. (ed), *Proc. of Software Engineering and Applications (SEA 2005)*, Phoenix, AZ, 2005.
- [194] Menezes, A., van Oorschot, P., Vanstone, S., *Handbook of Applied Cryptography*. CRC Press, 1996.
- [195] Adida, B. and Wikström, D., "Obfuscated Ciphertext Mixing," *IACR Eprint Archive*, no. 394, 2005.
- [196] Kolmogorov, A.N., "Logical Basis for Information Theory and Probability Theory," *IEEE Trans. Inform. Theory*, vol. IT-14, pp. 662-664, Sept. 1968.
- [197] Chaitin, G.J., "Information-Theoretic Computational Complexity," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 10-15, 1974.
- [198] Harrison, W., "An Entropy-Based Measure of Software Complexity," *IEEE Transactions on Soft. Eng.*, vol. 18, no. 11, pp. 1025-1029, Nov. 1992.
- [199] Wegener, I., *The Complexity of Boolean Functions (Wiley-Teubner Series in Computer Science)*. John Wiley & Sons, 1987.
- [200] Brglez, F. and Fujiwara, H., "A Neutral Netlist of 10 Combinational Benchmark Circuits," in *Proc. of the IEEE Int'l Symp. On Circuits and Systems*, IEEE Press, pp. 695-698, 1985.
- [201] Basto, L., "First Results of ITC'99 Benchmark Circuits," *IEEE Design & Test of Computers*, vol. 17, no. 3, July/Sept. 2000.
- [202] Edwards, S., "The Challenges of Hardware Synthesis from C-like Languages," in *Proc. of the Conference on Design, Automation and Test in Europe*, pp. 66-67, IEEE Computer Society, 2005.
- [203] Hansen, M., Yalcin, H., and Hayes, J. P., "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering," *IEEE Design and Test*, vol. 16, pp. 72-80, 1999.
- [204] Anderson, H. R. and Hulgaard, H., "Boolean Expression Diagrams," in *Proc. of the IEEE Symposium on Logic in Computer Science (LICS'97)*, 1997.
- [205] Jain, J., Yan, A., Fujita, M., and Sangiovanni-Vincentelli, A., "A Survey of Techniques for Formal Verification of Combinational Circuits," in *Proc. of the 1997 Int'l Conf. on Computer Design (ICCD'97)*, IEEE Press, 1997.
- [206] Bryant, R., "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677-691, Aug. 1986.
- [207] Boutaleb, K., Jegou, P., and Terrioux, C., "Strong learnt (no)goods in ROBDDs for solving structured CSPs," in *Proc. of the AAAI 2006 Workshop on Learning for Search*, American Association for Artificial Intelligence, 2006.
- [208] Hulgaard, H., Williams, P. F., and Andersen, H. R., "Equivalence Checking of Combinational Circuits using Boolean Expression Diagrams," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 18, pp. 903-917, 1999.
- [209] Shannon, C.E., "Communication Theory of Secrecy Systems", *Bell System Technical Journal*, vol. 28, no. 4, pp. 656-715, 1949.
- [210] Kerckhoff, A., "La Cryptographie Militaire", *Journal des Sciences Militaires*, vol. IX, pp. 5-83, Jan. 1883.
- [211] Feistel, H., "Cryptography and Computer Privacy," *Scientific American*, vol. 228, pp. 15-23, 1973.
- [212] Nyberg, K., "Perfect Nonlinear S-Boxes," in *Advances in Cryptology (EUROCRYPT '91)*, 1991.
- [213] Brickell, E. F., Moore, J. H., and Purtil, M. R., "Structure in the S-boxes of the DES," in *Advances in Cryptology (CRYPTO'86)*, vol. 263, pp. 3-9, A. M. Odlyzko, Ed. New York: Springer-Verlag, 1987.
- [214] Dawson, M. and Tavares, S., "An Expanded Set of S-box Design Criteria Based on Information Theory and its Relation to Differential-like Attacks," in *Advances in Cryptology (EUROCRYPT '91)*, 1991.
- [215] Gargiulo, J. "S-Box modifications and their effect on DES-like encryption systems," White Paper, Information Security Reading Room, SANS Institute, 2002.
- [216] Webster, A. and Tavares, S., "On the Design of S-Boxes," in *Advances in Cryptology (CRYPTO '85)*, pp. 523-534, 1985.

- [217] Meier, W. and Staffelbach, O., "Nonlinearity Criteria for Cryptographic Functions," in *Advances in Cryptology (EUROCRYPT '89)*, 1989.
- [218] Pieprzyk, J. and Finkelstein, G., "Towards Effective Nonlinear Cryptosystem Design," *IEE Proceedings*, Part E, vol. 135, pp. 325-335, 1988.
- [219] Gregg, J., *Ones and Zeros: Understanding Boolean Algebra, Digital Circuits, and the Logic of Sets*. IEEE Press, 1998.
- [220] Mendelson, E. *Schaum's Outline Series: Theory and Problems of Boolean Algebra & Switching Circuits*. McGraw-Hill Book Company, 1970.
- [221] Bellare, M., Desai, A., Jokipii, E., and Rogaway, P., "A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation," in *Proc. of the 38th Symp. on FOCS*, IEEE, 1997.
- [222] McDonald, J.T., Yasinsac, A., and Thompson, W., "A Survey of Mobile Agent Security", Technical Report TR-050329, Dept. of Computer Science, Florida State University, March 2005, available <http://www.cs.fsu.edu/research/reports/TR-050329.pdf>.
- [223] Gunupudi, V. and Tate, S., "Performance Evaluation of Data Integrity Mechanisms for Mobile Agents", in *Proc. of the International Conference on Information Technology: Coding and Computing (ITCC'04)*, 2004, Las Vegas, NV, USA, pp. 62-69.
- [224] Sobrado, I., "Evaluation of Two Security Schemes for Mobile Agents", *ACM SIGCOMM Computer Communication Review*, 2001, vol. 31, no. 2, pp. 2-19.
- [225] Fischmeister, S., Vigna, G., and Kemmerer, R.A., "Evaluating the Security of Three Java-Based Mobile Agent Systems", in Picco, G. (ed.), *Proc. of the 5th IEEE International Conference on Mobile Agents*, vol. 2240 of Lecture Notes in Computer Science, 2001, Springer-Verlag, pp. 31-40.
- [226] Milagres, F., Moreira, E., Pimentao, J., and Sousa, P., "Security Analysis of a Multi-Agents System in Eu's DeepSia Project", in *Proc. of the EBR'2002 - First Seminar On Advanced Research In Electronic Business 2002*, 2002, Rio de Janeiro, pp. 155-162.
- [227] Kotzanikolaou, P., Katsirelos, G., and Chrissikopoulos, V., "Mobile Agents for Secure Electronic Transactions", *Recent Advances in Signal Processing Communications*, Mastorakis, N. (ed.), 1999, River Edge, NJ: World Scientific Engineering Society, 363-368.
- [228] O'Malley, S., Self, A., and DeLoach, S., "Comparing Performance of Static Versus Mobile Multiagent Systems", in *Proc. of the National Aerospace and Electronics Conference (NAECON)*, 2000, Dayton, OH.
- [229] Thomas, R., "A Survey of Mobile Code Security Techniques", in *Proc. of the Proceedings of the 22nd National Information Systems Security Conference*, 1999.
- [230] Loureiro, S., Molva, R., and Roudier, Y., "Mobile Code Security", in *Proc. of the ISYPAR 2000 (4ème Ecole d'Informatique des Systèmes Parallèles et Répartis)*, Code Mobile, 2000: Toulouse, France.
- [231] Wahbe, R., Lucco, S., Anderson, T., and Graham, S., "Efficient Software-Based Fault Isolation", in *Proc. of the 14th ACM Symposium on Operating System Principles*, ACM SIGOPS Operating Systems Review, 1993: ACM Publishers, pp. 203-216.
- [232] Gosling, J., Joy, B., and Steele, G., *The JAVA Language Specification*, 1996: Addison Wesley.
- [233] Gong, L., "JAVA Security: Present and near Future", *IEEE Micro*, 1997, vol. 17, no. 3, pp. 14-19.
- [234] Gong, L., *Inside Java 2 Platform Security*, 1999: Addison Wesley.
- [235] Ousterhout, J.K., "Scripting: Higher-Level Programming for the 21st Century", *IEEE Computer*, 1998, pp. 23-30.
- [236] Ousterhout, J.K., Levy, J., and Welch, B., "The Safe-TcL Security Model", in Vigna, G. (ed.), *Mobile Agents and Security*, vol. 1419 of Lecture Notes in Computer Science, 1998, Springer, pp.
- [237] Gong, L. and Schemers, R., "Signing, Sealing, and Guarding JAVA Objects", in Vigna, G. (ed.), *Mobile Agents and Security*, vol. 1419 of Lecture Notes in Computer Science, 1998, Springer-Verlag, pp. 206-216.
- [238] Hopwood, D.A., "Comparison between JAVA and ActiveX Security", 1997, updated: December 1997. Available: <http://www.users.zetnet.co.uk/hopwood/papers/compsec97.html>.
- [239] Farmer, W.M., Guttman, J.D., and Swarup, V., "Security for Mobile Agents: Authentication and State Appraisal", in Bertino, E., et al., (eds.), *Computer Security--Proceedings of the European Symposium on Research in Computer Security (ESORICS 96)*, vol. 1146 of Lecture Notes in Computer Science, 1996, pp. 118-130.
- [240] Necula, G.C., "Proof-Carrying Code", in *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, Paris, France, pp. 106-119.
- [241] Lee, P. and Necula, G., "Research on Proof-Carrying Code for Mobile-Code Security", in *Proc. of the DARPA Workshop on Foundations for Secure Mobile Code*, 1997, Monterey, CA.

- [242] Appel, A.W., "Foundational Proof-Carrying Code", in *Proc. of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, 2001: IEEE Computer Society, pp. 247.
- [243] Feigenbaum, J. and Lee, P., "Trust Management and Proof-Carrying Code in Secure Mobile Code Applications: Position Paper", in *Proc. of the DARPA Workshop on Foundations for Secure Mobile Code*, 1997.
- [244] Westhoff, D., Schneider, M., Unger, C., and Kaderali, F., "Protecting a Mobile Agent's Route against Collusions", in *Proc. of the 6th Annual International Workshop on Selected Areas in Cryptography*, vol. 1758 of Lecture Notes in Computer Science, 1999, Springer, pp. 215-225.
- [245] Bradshaw, J.M., Beautement, P., Bunch, L., and Drakunov, S., "Making Agents Acceptable to People", in *Handbook of Intelligent Information Technology*, 2003, IOS Press: Amsterdam, Netherlands.
- [246] Bradshaw, J., Cabri, G., and Montanari, R., "Taking Back Cyberspace", *IEEE Computer*, 2003, vol. 36, no. 7.
- [247] Bradshaw, J.M., Suri, N., Kahn, M., Sage, P., Weishar, D., and Jeers, R., "Terraforming Cyberspace: Toward a Policy-Based Grid Infrastructure for Secure, Scalable, and Robust Execution of Java-Based Multi-Agent Systems", in *Proc. of the Autonomous Agents 2001 Workshop on Scalable Agent Infrastructure*, 2001, Montreal, Canada.
- [248] Scott, D., Beresford, A., and Mycroft, A., "Spatial Security Policies for Mobile Agents in a Sentient Computing Environment", in Pezz'e, M. (ed.), *Proc. of FASE 2003*, vol. 2621 of Lecture Notes in Computer Science, 2003, Springer, pp. 102-117.
- [249] Corradi, A., Dulay, N., Montanari, R., and Stefanelli, C., "Policy-Driven Management of Mobile Agent Systems", in *Proc. of the Int'l Workshop on Policies for Distributed Systems and Networks - Policy 2001*, vol. 1995 of Lecture Notes in Computer Science, 2001, Springer-Verlag, pp. 214-229.
- [250] Jansen, W., "Countermeasures for Mobile Agent Security", *Computer Communications, Special Issue on Advanced Security Techniques for Network Protection*, 2000.
- [251] Neuman, B., "Proxy-Based Authorization and Authentication and Accounting for Distributed Systems", in *Proc. of the 13th Int'l Conference on Distributed Computing Systems*, 1993, pp. 283-291.
- [252] Lampson, B., Abadi, M., Burrows, M., and Wobber, E., "Authentication in Distributed Systems: Theory and Practice", *ACM Transactions on Computer Systems*, 1992, vol. 10, pp. 265-310.
- [253] Berkovits, S., Guttman, J.D., and Swarup, V., "Authentication for Mobile Agents", in Vigna, G. (ed.), *Mobile Agents and Security*, vol. 1419 of Lecture Notes in Computer Science, 1998, Springer-Verlag, pp. 114-136.
- [254] Romao, A. and Da Silva, M., "Proxy Certificates: A Mechanism for Delegating Digital Signature Power to Mobile Agents", in *Proc. of the Workshop on Agents in Electronic Commerce*, 1999, pp. 131-140.
- [255] Bellare, M. and Miner, S., "A Forward-Secure Digital Signature Scheme", in Weiner, M. (ed.), *Advances in Cryptology-CRYPTO '99*, vol. 1666 of Lecture Notes in Computer Science, 1999, pp. 431-448.
- [256] Krawczyk, H., "Simple Forward-Secure Signatures from Any Signature Scheme", in *Proc. of the 7th ACM Conference on Computer and Communications Security*, 2000, pp. 108-115.
- [257] Lee, B., Kim, H., and Kim, K., "Secure Mobile Agent Using Strong Nondesignated Proxy Signature", in *Proc. of ACISP*, vol. 2119 of Lecture Notes in Computer Science, 2001, Springer-Verlag, pp. 474-486.
- [258] Kotzanikolaou, P., Burmester, M., and Chrissikopoulos, V., "Dynamic Multi-Signatures for Secure Autonomous Agents", in *Proc. of the DEXA-MDDS Conference*, 2001: IEEE Computer Society, pp. 587-591.
- [259] Borselius, N., Mitchell, C.J., and Wilson, A., "Undetachable Threshold Signatures", in *Proc. of the 8th IMA Int'l Conference: Cryptography and Coding*, vol. 2260 of Lecture Notes in Computer Science, 2001, Springer-Verlag, pp. 239-244.
- [260] Kim, H., Baek, J., Lee, B., and Kim, K., "Computing with Secrets for Mobile Agent Using One-Time Proxy Signature", in *Proc. of the SCIS 2001*, 2001, Oiso, Japan, pp. 845-850.
- [261] Yi, X., Siew, C.K., and Syed, M.R., "Digital Signature with One-Time Pair of Keys", *Electronics Letters*, 2000, vol. 36, no. 2, pp. 130-131.
- [262] Roth, V. and Jalali-Sohi, M., "Access Control and Key Management for Mobile Agents", *Computers & Graphics*, 1998, vol. 22, no. 4, pp. 457-461.
- [263] Roth, V. and Jalali-Sohi, M., "Concepts and Architecture of a Security-Centric Mobile Agent Server", in *Proc. of the 5th International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, 2001, Dallas, Texas: IEEE Computer Society, pp. 435-442.

- [264] Rasmusson, L. and Janson, S., "Simulated Social Control for Secure Internet Commerce", in *Proc. of the New Security Paradigms Workshop*, 1996, Lake Arrowhead, CA: ACM Press, pp. 18-26.
- [265] Meadows, C., "Detecting Attacks on Mobile Agents", in *Proc. of the DARPA Workshop on Foundations for Secure Mobile Code*, 1997, Monterey CA.
- [266] Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., and Jakubowski, M.H., "Oblivious Hashing: A Stealthy Software Integrity Verification Primitive", in *Information Hiding*, vol. 2578 of Lecture Notes in Computer Science, 2002, Springer-Verlag, pp. 400-414.
- [267] Vigna, G., "Cryptographic Traces for Mobile Agents", in Vigna, G. (ed.), *Mobile Agents and Security*, vol. 1419 of Lecture Notes in Computer Science, 1998, Springer-Verlag.
- [268] Kassab, L. and Voas, J., "Agent Trustworthiness", in *Proc. of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems Secure Internet Mobile Computations*, 1998, Brussels.
- [269] Biehl, I., Meyer, B., and Wetzl, S., "Ensuring the Integrity of Agent-Based Computations by Short Proofs", in Rothermel, K. and Hohl, F. (eds.), *Proc. of the 2nd Int'l Workshop, Mobile Agents 98*, vol. 1477 of Lecture Notes in Computer Science, 1998, Springer-Verlag, pp. 183-194.
- [270] Gertner, Y., Ishail, Y., Kushilevitz, E., and Malkin, T., "Protecting Data Privacy in Private Information Retrieval Schemes", in *Proc. of the 30th Annual ACM Symposium on Theory of Computing (STOC)*, 1998, pp. 151-160.
- [271] Baek, J., Lee, D., and Ramakrishna, R.S., "A Design of Protocol for Detecting an Agent Clone in Mobile Agent Systems and Its Correctness Proof", in *Proc. of the 8th Annual ACM Symposium on Principles of Distributed Computing*, 1999, Atlanta, GA: ACM Press, pp. 269.
- [272] Lam, T. and V., W., "A Mobile Agent Clone Detection System with Itinerary Privacy", in *Proc. of the 11th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'02)*, 2002, Pittsburgh, Pennsylvania, USA, pp. 68.
- [273] Riordan, J. and Schneier, B., "Environmental Key Generation Towards Clueless Agents", in Vigna, G. (ed.), *Mobile Agents and Security*, vol. 1419 of Lecture Notes in Computer Science, 1998, Springer-Verlag, pp. 15-24.
- [274] Grimley, M.J. and Monroe, B.D., "Protecting the Integrity of Agents", *ACM Magazine*, 1999.
- [275] Ng, S. and Cheung, K., "Protecting Mobile Agents against Malicious Hosts by Intention Spreading", in *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, 1999, pp. 725-729.
- [276] Ng, S. and Cheung, K., "A Trust-Level Exchanging Protocol in Mobile Agent Systems for Security and Performance Benefits", in *Proc. of the World Wide Web: Technologies and Applications for the New Millennium*, 2000: CSREA, pp. 27-35.
- [277] Minsky, Y., Renesse, R., Schneider, F.B., and Stoller, S.D., "Cryptographic Support for Fault-Tolerant Distributed Computing", in *Proc. of the 7th ACM SIGOPS European Workshop*, 1996, Connemara, Ireland, pp. 109-114.
- [278] Shamir, A., "How to Share a Secret", *Communications of the ACM*, 1979, vol. 22, no. 11, pp. 612-613.
- [279] Pears, S., Xu, J., and Boldyreff, C., "A Dynamic Shadow Approach for Mobile Agents to Survive Crash Failures", in *Proc. of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, 2003, Hakodate, Hokkaido, Japan, pp. 113-120.
- [280] Roth, V., "Mutual Protection of Co-Operating Agents", in Vitek, J. and Jensen, C. (eds.), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, vol. 1603 of Lecture Notes in Computer Science, 1999, Springer-Verlag, pp. 275-285.
- [281] Reed, M., Syverson, P., and Goldschlag, D., "Anonymous Connections and Onion Routing", *IEEE Journal on Selected Areas in Communication, Special Issue on Copyright and Privacy Protection*, 1998, vol. 16, no. 4, pp. 482-494.
- [282] Wang, Y. and Pang, X., "Security and Robustness Enhanced Route Structures for Mobile Agents", *Mobile Networks and Applications*, 2003, vol. 8, no. 4, pp. 413-423.
- [283] Vijil, E. and Iyer, S., "Identifying Collusions: Co-Operating Malicious Hosts in Mobile Agent Itineraries", in *Proc. of the 2nd International Workshop on Security of Mobile MultiAgent Systems (SEMAS'2002)*, 2002, Bologna, Italy.
- [284] Karnik, N. and Tripathi, A., "A Security Architecture for Mobile Agents in Ajanta", in *Proc. of the 20th International Conference on Distributed Computing Systems*, 2000: IEEE Computer Society Press, pp. 402-409.
- [285] Domingo-Ferrer, J., "Mobile Agent Route Protection through Hash-Based Mechanisms", in Rangan, C.P. and Ding, C. (eds.), *Proc. of INDOCRYPT '01*, vol. 2247 of Lecture Notes in Computer Science, 2001, Springer-Verlag, pp. 17-29.

- [286] Young, A. and Yung, M., "Sliding Encryption: A Cryptographic Tool for Mobile Agents", in *Proc. of the 4th International Workshop on Fast Software Encryption (FSE '97)*, vol. 1267 of Lecture Notes in Computer Science, 1997, Springer-Verlag, pp. 230-241.
- [287] Karjoth, G., "Secure Mobile Agent-Based Merchant Brokering in Distributed Marketplaces", in Kotz, D. and Mattern, F. (eds.), *Proc. of ASA/MA 2000*, vol. 1882 of Lecture Notes in Computer Science, 2000, Springer Verlag, pp. 44-56.
- [288] Tate, S.R. and Xu, K., "Mobile Agent Security through Multi-Agent Cryptographic Protocols", in *Proc. of the 4th International Conference on Internet Computing (IC 2003)*, 2003, pp. 462-468.
- [289] Abadi, M. and Feigenbaum, J., "Secure Circuit Evaluation: A Protocol Based on Hiding Information from an Oracle", *Journal of Cryptology*, 1990, vol. 2, no. 1, pp. 1-12.
- [290] Abadi, M., Feigenbaum, J., and Kilian, J., "On Hiding Information from an Oracle", *Journal of Computer and System Sciences*, 1989, vol. 39, no. 1, pp. 21-50.
- [291] Sander, T. and Tschudin, C., "Mobile Cryptography", in *Proc. of the IEEE Symposium on Security and Privacy*, 1998, pp. 215-224.
- [292] Sander, T., Young, A., and Yung, M., "Non-Interactive Cryptocomputing for NC1", in *Proc. of the 40th IEEE Symposium on Foundations of Computer Science*, 1999, pp. 17-19.
- [293] Yokoo, M. and Suzuki, K., "Secure Multi-Agent Dynamic Programming Based on Homomorphic Encryption and Its Application to Combinatorial Auctions", in *Proc. of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (ICAA'02)*, 2002, Bologna, Italy, pp. 112-119.
- [294] Cartrysse, K. and van der Lubbe, J.C.A., "Privacy in Mobile Agents", in *Proc. of the 1st IEEE Symposium on Multi-Agents Security and Survivability (MASS'04)*, 2004, Philadelphia, PA.
- [295] Cartrysse, K. and van der Lubbe, J.C.A., "Secrecy in Mobile Code", in *Proc. of the 25th Symposium on Information Theory in the Benelux*, 2004, pp. 161-168.
- [296] Loureiro, S. and Molva, R., "Function Hiding Based on Error Correcting Codes", in *Proc. of the International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99)*, 1999: City University of Hong Kong Press.
- [297] Loureiro, S. and Molva, R., "Privacy for Mobile Code", in *Proc. of the Distributed Object Security Wrkshp (OOPSLA'99)*, 1999, Denver, CO, pp. 37-42.
- [298] Zhou, C. and Sun, Y., "SPMH: A Solution to the Problem of Malicious Hosts", *Journal of Computer Science and Technology*, 2002, vol. 17, no. 6, pp. 738 - 748.
- [299] Zhong, S. and Yang, Y.R., "Verifiable Distributed Oblivious Transfer and Mobile Agent Security", in *Proc. of the 2003 Joint Workshop on Foundations of Mobile Computing*, 2003.
- [300] Desmedt, Y., "Society and Group Oriented Cryptography", in Pomerance, C. (ed.), *Advances in Cryptology - CRYPTO'87*, vol. 293 of Lecture Notes in Computer Science, 1988, Springer-Verlag, pp. 120-127.
- [301] Shoup, V., "Practical Threshold Signatures", in Preneel, B. (ed.), *Advances in Cryptology - EUROCRYPT 2000*, vol. 1807 of Lecture Notes in Computer Science, 2000, Springer-Verlag, pp. 207-220.
- [302] Varadharajan, V. and Foster, D., "A Security Architecture for Mobile Agent Based Applications", *World Wide Web*, 2003, vol. 6, no. 1, pp. 93-122.
- [303] Shi, W., Lee, H., Lu, C., and Ghosh, M., "Towards the Issues in Architectural Support for Protection of Software Execution", in *Proc. of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, 2004.
- [304] Cartrysse, K., "Security and Privacy in Mobile Agent Systems", in *Proc. of the Safe-NL Workshop Security: Applications, Formal Aspects, and Environments*, 2002: University of Twente.
- [305] Sobrado, I., "A One-Time Pad Cipher for Data Protection in Distributed Environments", 2000, updated: August 2005. Available: <http://xxx.lanl.gov/abs/cs.CR/0005026>.
- [306] Diaz, J.A.P. and Gutierrez, D.A., "Protecting the Data State of Mobile Agents by Using Bitmaps and Xor Operators", *Informatica, International Journal of Computing and Informatics*, 2002, vol. 26, no. 4.
- [307] Diaz, J.A.P. and Gutierrez, D.A., "A Fast Data Protection Technique for Mobile Agents to Avoid Attacks in Malicious Hosts", *Electronic Notes in Theoretical Computer Science*, 2001, vol. 30, no. 3.
- [308] Corradi, A., Montanari, R., and Stefanelli, C., "Mobile Agents Integrity in E-Commerce Applications", in *Proc. of the 19th IEEE International Conference on Distributed Computing Systems Workshop (ICDCS'99)*, 1999, Austin, Texas: IEEE Computer Society Press, pp. 59-64.
- [309] Cheng, J. and Wei, V., "Defenses against the Truncation of Computation of Free-Roaming Agents", in *Proc. of the 4th International Conference on Information and Communication Security*, vol. 2513 of Lecture Notes in Computer Science, 2002, pp. 1-12.

- [310] Zhou, J., Onieva, J., and Lopez, J., "Analysis of a Free Roaming Agent Result-Truncation Defense Scheme", in *Proc. of the IEEE Conference on Electronic Commerce*, 2004.
- [311] Loureiro, S., Molva, R., and Pannetrat, A., "Secure Data Collection with Updates", *Electronic Commerce Research Journal*, 2001, vol. 1, no. 2, pp. 119-130.
- [312] Suen, A., "Mobile Agent Protection with Data Encapsulation and Execution Tracing, Master's Thesis. Technical Report TR-030402, Department of Computer Science, Florida State University, 2003.
- [313] Park, J., Lee, D., and Lee, H., "One-Time Key Generation System for Agent Data Protection", *IECE Transactions on Information and Systems*, 2000, vol. E83-D, pp. 11.
- [314] El Gamal, T., "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms", *IEEE Transactions on Information Theory*, 1985, vol. 31, no. 4, pp. 469-472.
- [315] Yao, A.C., "How to Generate and Exchange Secrets", in *Proc. of the 27th IEEE Symposium on Foundations of Computer Science*, 1986, pp. 162-167.
- [316] Goldreich, O., Micali, S., and Wigderson, A., "How to Play Any Mental Game", in *Proc. of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, 1987, pp. 218-229.
- [317] Ben-Or, M., Goldwasser, S., and Wigderson, A., "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation", in *Proc. of the Annual ACM Symposium on Theory of Computing '88*, 1988: ACM, pp. 1-10.
- [318] Chaum, D., Crépeau, C., and Damgard, I., "Multiparty Unconditionally Secure Protocols (Extended Abstract)", in *Proc. of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, 1988, Chicago, Illinois, pp. 11-19.
- [319] Kilian, J., "Founding Cryptography on Oblivious Transfer", in *Proc. of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, 1988, pp. 20-31.
- [320] Abadi, M., Feigenbaum, J., and Kilian, J., "On Hiding Information from an Oracle", *Journal of Computer and System Sciences*, 1989, vol. 39, no. 1, pp. 21-50.
- [321] Abadi, M. and Feigenbaum, J., "Secure Circuit Evaluation: A Protocol Based on Hiding Information from an Oracle", *Journal of Cryptology*, 1990, vol. 2, no. 1, pp. 1-12.
- [322] Bellare, M., Micali, S., and Rogaway, P., "The Round Complexity of Secure Protocols", in *Proc. of the 22nd Annual ACM Symposium on Theory of Computing (STOC)*, 1990, Baltimore, Maryland, United States: ACM Press, pp. 503-513.
- [323] Beaver, D., "Foundations of Secure Interactive Computing", in *Proc. of the 11th Annual International Cryptology Conference on Advances in Cryptology*, vol. 576 of Lecture Notes in Computer Science, 1991, Springer-Verlag, pp. 377-391.
- [324] Micali, S. and Rogaway, P., "Secure Computation", in *Advances in Cryptology - CRYPTO'91*, vol. 576 of Lecture Notes in Computer Science, 1992, Springer-Verlag, pp. 392-404.
- [325] Canetti, R., Feige, U., Goldreich, O., and Naor, M., "Adaptively Secure Multi-Party Computation", in *Proc. of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, 1996, pp. 639-648.
- [326] Cramer, R., Damgard, I., Dziembowski, S., Hirt, M., and Rabin, T., "Efficient Multiparty Computations with Dishonest Minority", in Stern, J. (ed.), *Proc. of EUROCRYPT '99*, vol. 1592 of Lecture Notes in Computer Science, 1999, IACR, Springer-Verlag, pp.
- [327] Naor, M., Pinkas, B., and Sumner, R., "Privacy Preserving Auctions and Mechanism Design." in *Proc. of the 1st ACM Conference on Electronic Commerce*, 1999, pp. 129-139.
- [328] Goldreich, O., "Secure Multi-Party Computation. Working Draft, Version 1.2". 2000.
- [329] Hirt, M. and Maurer, U.M., "Robustness for Free in Unconditional Multi-Party Computation", in *Proc. of the 21st Annual International Cryptology Conference on Advances in Cryptology*, vol. 2139 of Lecture Notes in Computer Science, 2001, Springer-Verlag, pp. 101-118.
- [330] Naor, M. and Nisim, K., "Communication Complexity and Secure Function Evaluation", *Electronic Colloquium on Computational Complexity (ECCC)*, 2001, vol. 8, pp. 62.
- [331] Canetti, R., Lindell, Y., Ostrovski, R., and Sahai, A., "Universally Composable Two-Party and Multi-Party Secure Computation", in *Proc. of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, 2002, pp. 494-503.
- [332] Damgard, I. and Nielsen, J., "Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption", in *Advances in Cryptology - CRYPTO'03*, vol. 2729 of Lecture Notes in Computer Science, 2003, pp. 247-264.
- [333] Fitz, M., Garay, J., Mauerer, U., and Ostravsky, R., "Minimal Complete Primitives for Secure Multi-Party Computation", *Journal of Cryptography*, 2005, vol. 18, no. 1, pp. 37-61.
- [334] Du, W. and Atallah, M., "Secure Multi-Party Computation Problems and Their Applications: A Review and Open Problems", in *Proc. of the New Security Paradigms Workshop*, 2001, Cloudcroft, NM, USA, pp. 11-20.

- [335] Malkhi, D., Nisan, D., Pinkas, B., and Sella, Y., "Fairplay-a Secure Two-Party Computation System", in *Proc. of the Usenix Security Symposium '04*, 2004, pp. 287-302.
- [336] Bellare, M. and Micali, S., "Non-Interactive Oblivious Transfer and Applications", in *Proc. of the Advances in Cryptology - CRYPTO'89*, 1990, Santa Barbara, California, USA: Springer-Verlag, pp. 547-559.
- [337] Rabin, T. and Ben-Or, M., "Verifiable Secret Sharing and Multiparty Protocols with Honest Majority." in *Proc. of the 21st Annual ACM Symposium on Theory of Computing*, 1989, Seattle, Washington, USA, pp. 73-85.
- [338] Yao, A.C., "Protocols for Secure Computation", in *Proc. of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, 1982.
- [339] Chaum, D., Damgard, I., and Van De Graaf, J., "Multiparty Computations Ensuring Privacy of Each Party's Input and Correctness of the Result", in Pomerance, C. (ed.), *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, vol. 293 of Lecture Notes in Computer Science, 1988, pp. 87-119.
- [340] Naor, M. and Pinkas, B., "Efficient Oblivious Transfer Protocols", in *Proc. of the SODA 2001 (SIAM Symposium on Discrete Algorithms)*, 2001, Washington, D.C.
- [341] Naor, M. and Pinkas, B., "Distributed Oblivious Transfer", in *Advances in Cryptology - ASIACRYPT'00*, vol. 1976 of Lecture Notes in Computer Science, 2000, Springer-Verlag, pp. 200-219.
- [342] Naor, M. and Pinkas, B., "Oblivious Transfer and Polynomial Evaluation", in *Proc. of the 31st Annual ACM Symposium on Theory of Computer Science (STOC)*, 1999, Atlanta, GA, pp. 245-254.
- [343] Ostravsky, R. and Yung, M., "How to Withstand Mobile Virus Attacks", in *Proc. of the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1991, pp. 51-59.
- [344] Endsuleit, R. and Mie, T., "Secure Multi-Agent Computations", in *Proc. of the International Conference on Security and Management (CSREA)*, 2003, pp. 149-155.
- [345] Tate, S.R. and Xu, K., "On Garbled Circuits and Constant Round Secure Function Evaluation", CoPS Lab Technical Report 2003-02. 2003.
- [346] Gennaro, R., Rabin, M.O., and Rabin, T., "Simplified Vss and Fast-Track Multiparty Computations with Applications to Threshold Cryptography", in *Proc. of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, 1998, pp. 101-111.
- [347] Beaver, D., Feigenbaum, J., Kilian, J., and Rogaway, P., "Security with Low Communication Overhead", in *Proc. of the 10th Annual International Cryptology Conference on Advances in Cryptology*, vol. 537 of Lecture Notes in Computer Science, 1990, Springer-Verlag, pp. 62-76.
- [348] Cramer, R., Damgard, I., and Nielsen, J.B., "Multiparty Computation from Threshold Homomorphic Encryption", in *Advances in Cryptology - EUROCRYPT'01*, vol. 2045 of Lecture Notes in Computer Science, 2001, pp. 280-300.
- [349] Ben-Or, M., Canetti, R., and Goldreich, O., "Asynchronous Secure Communications", in *Proc. of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, 1993: ACM, pp. 52-61.
- [350] Ben-Or, M., Kelmer, B., and Rabin, T., "Asynchronous Secure Computations with Optimal Resilience", in *Proc. of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1994, pp. 183-192.
- [351] Canetti, R., "Security and Composition of Multiparty Cryptographic Protocols", *Journal of Cryptology*, 2000, vol. 13, no. 1, pp. 143-202.
- [352] Canetti, R., "Universally Composable Security: A New Paradigm for Cryptographic Protocols", in *Proc. of the 42nd IEEE Symposium on Foundations of Computer Science*, 2001, pp. 136.
- [353] Endsuleit, R. and Wagner, A., "Possible Attacks on and Countermeasures for Secure Multi-Agent Computation", in *Proc. of the International Conference on Security and Management (SAM)*, 2004, pp. 221-227.
- [354] Rivest, R.L., Adleman, L., and Dertouzos, M.L., "On Data Banks and Privacy Homomorphisms", in *Proc. of the Foundations of Secure Computation*, 1978: Academic Press, pp. 169-177.
- [355] Neven, G., Van Hoeymissen, E., De Decker, B., and Piessens, F., "Enabling Secure Distributed Computations: Semi-Trusted Hosts and Mobile Agents." *Networking and Information Systems Journal*, 2000, vol. 3, pp. 1-18.
- [356] Dadon-Elichai, A., "RDS: Remote Distributed Scheme for Protecting Mobile Agents", in *Proc. of the Autonomous Agents and Multi-Agent Systems Conference (AAMAS'04)*, 2004, New York City, NY: ACM.
- [357] Wang, Y. and Tan, K., "Dispatching Mobile Agents with Secure Routes in Parallel", in Qing, S., Okamoto, T., and Zhou, J. (eds.), *ICICS 2001*, vol. 2229 of Lecture Notes in Computer Science, 2001, Springer-Verlag, pp. 386-397.

- [358] Page, J., Zaslavsky, A., and Indrawan, M., "A Buddy Model of Security for Mobile Agent Communities Operating in Pervasive Scenarios", in *Proc. of the 2nd Australasian Information Security Workshop (AISW2004)*, 2004, Dunedin (New Zealand), pp. 17-25.
- [359] Pleisch, S. and Schiper, A., "Approaches to Fault-Tolerant and Transactional Mobile Agent Execution---an Algorithmic View", *ACM Computing Surveys*, 2004, vol. 36, no. 3, pp. 219-262.
- [360] Claessens, J., Preneel, B., and Vandewalle, J., "(How) Can Mobile Agents Do Secure Electronic Transactions on Untrusted Hosts? A Survey of the Security Issues and the Current Solutions", *ACM Transactions on Internet Technology*, 2003, vol. 3, no. 1, pp. 28-48.
- [361] Guan, S.U. and Yang, Y., "SAFE: Secure-Roaming Agent for E-Commerce", in *Proc. of the 26th International Conference on Computers and Industrial Engineering*, 1999, pp. 33-37.
- [362] Guan, S.U., Yang, Y., and You, J., "POM - a Mobile Agent Security Model against Malicious Hosts." in *Proc. of the 4th International Conference on High-Performance Computing in the Asia-Pacific Region*, 2000, Beijing, China.
- [363] Gambetta, D., "Can We Trust Trust?" *Trust: Making and Breaking Cooperative Relations*, Gambetta, D. (ed.), 1990, Basil Blackwell, Oxford, 213-237.
- [364] Kagal, L., Finin, T., and Peng, Y., "A Delegation Based Model for Distributed Trust", in *Proc. of the IJCAI-01 Workshop on Autonomy, Delegation, and Control*, 2001.
- [365] Cahill, V., Gray, E., Seigneur, J.M., Jensen, C.D., Chen, Y., Shand, B., Dimmock, N., Twigg, A., Bacon, J., English, C., Wagealla, W., Terzis, S., Nixon, P., Serugendo, G.d.M., Bryce, C., Carbone, M., Krukow, K., and Nielsen, M., "Using Trust for Secure Collaboration in Uncertain Environments", *Pervasive Computing Mobile And Ubiquitous Computing*, 2003, vol. 2, no. 3, pp. 52-61.
- [366] Capra, L., "Engineering Human Trust in Mobile System Collaborations", in *Proc. of the 12th International Symposium on the Foundations of Software Engineering (SIGSOFT 2004/FSE-12)*, 2004, Newport Beach, CA.
- [367] Carbone, M., Nielsen, M., and Sassone, V., "A Formal Model for Trust in Dynamic Networks", in *Proc. of the 1st International Conference on Software Engineering and Formal Methods (SEFM'03)*, 2003, Brisbane, Australia, pp. 54-63.
- [368] Kagal, L., Finin, T., and Joshi, A., "Moving from Security to Distributed Trust in Ubiquitous Computing Environments", *IEEE Computer*, 2001.
- [369] Karjoth, D., Lange, D.B., and Oshima, M., "A Security Model for Aglets", *IEEE Internet Computing*, 1997, vol. 1, no. 4, pp. 68-77.
- [370] Jansen, W., "A Privilege Management Scheme for Mobile Agent Systems", in *Proc. of the International Conference on Autonomous Agents*, 2001, Montreal, Canada.
- [371] Hashii, B., Malabarba, S., Pandey, R., and Bishop, M., "Supporting Reconfigurable Security Policies for Mobile Programs", in *Proc. of the 9th International World Wide Web Conference (WWW9)*, 2000, Amsterdam, Netherlands.
- [372] Antonopoulos, N., Koukoumpetsos, K., and Ahmad, K., "A Distributed Access Control Architecture for Mobile Agents", in *Proc. of the International Network Conference*, 2000, Plymouth, UK.
- [373] Robles, S., Borrell, J., Bigham, J., Tokarchuk, L., and Cuthbert, L., "Design of a Trust Model for a Secure Multi-Agent Marketplace", in *Proc. of the 5th International Conference on Autonomous Agents*, 2001, Montreal: ACM Press, pp. 77-78.
- [374] Robles, S., Poslad, S., Borrell, J., and Bigham, J., "A Practical Trust Model for Agent-Oriented Electronic Business Applications", in *Proc. of the 4th International Conference on Electronic Commerce Research*, 2001, Dallas, USA, pp. 397-406.
- [375] Robles, S., Poslad, S., Borrell, J., and Bigham, J., "Adding Security and Privacy to Agents Acting in a Marketplace: A Trust Model", in *Proc. of the 35th Annual IEEE International Carnahan Conference on Security Technology*, 2001, London: IEEE Press, pp. 235-239.
- [376] Navarro, G., Robles, S., and Borrell, J., "An Access Control Method for Mobile Agents in Sea-of-Data Applications", *Upgrade*, 2002, vol. III, pp. 47-51.
- [377] Ametller, J., Robles, S., and Ortega-Ruiz, J.A., "Self-Protected Mobile Agents", in *Proc. of the 3rd International Conference on Autonomous Agents and Multi Agents Systems*, 2004: ACM Press.
- [378] Robles, S., Mir, J., and Borrell, J., "MARISMA-A: An Architecture for Mobile Agents with Recursive Itinerary and Secure Migration", in *Proc. of the 2nd Information Workshop on Security of Mobile Multiagent Systems*, 2002, Bologna, Italy.
- [379] Pfleeger, C. and Pfleeger, S.L., *Security in Computing*, 3rd ed, 2003, Upper Saddle River, NJ: Prentice Hall.
- [380] Davidson, S. and Harlow, J., "Guest Editor's Introduction: Benchmarking for Design and Test," *IEEE Design & Test of Computers*, vol. 17, no. 3, July-Sept 2000, pp. 12-14.

## BIOGRAPHICAL SKETCH

J. Todd McDonald is an active duty Lieutenant Colonel in the United States Air Force and is currently assigned as an Assistant Professor with Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright Patterson Air Force Base, OH. He was born in Atlanta, GA to Wayne McDonald and the late Margie McDonald. He is married to the former Angela Lodge of Calvary, GA and they have two children, Allie and Tucker. He graduated from Cairo High School, Cairo, GA, in 1986. Todd is an associate member of the Institute of Electrical and Electronic Engineers (IEEE) and the Association of Computing Machinery (ACM).

### **Service History**

*Aug 2003 – Dec 2006, PhD Candidate, Dept of Computer Science, Florida St University, Tallahassee, FL.*  
Graduate student and research assistant working towards doctoral degree in Computer Science. Working with Dr. Alec Yasinsac, worked to develop advanced security techniques to protect mobile computations and mobile agent applications.

*Jun 2002 – July 2003, Advisor to the Chief Scientist, AFOTEC, Albuquerque, NM.*  
Staff officer assisting the Technical Advisor to the Commander and the Chief Scientist of AFOTEC in technical analysis, research projects, and coordination of AFOTEC technical direction. Contract manager and technical lead for \$1M study effort that determines 4 year/\$50M spending plan for directed energy test infrastructure funded by OSD.

*Mar 2001 – Jun 2002, Chief, Test Capability Analysis Branch, AFOTEC, Albuquerque, NM.*  
Test manager conducting operational utility analysis of the Joint Modeling and Simulation System... led joint service evaluation of software system coordinating Army and Navy evaluation teams. Led trade-off analysis studies of test infrastructure capabilities (open air range assets, hardware-in-the-loop facilities, man/pilot-in-the-loop facilities, modeling and simulation systems) required for successful operational testing of major weapons systems (CV-22, F-22, Joint Strike Fighter, Space Based IR Systems, F-15, B-1B).

*Apr 2000 – Mar 2001, Chief, Software Plans Branch, AFOTEC, Albuquerque, NM.*  
Performed software analysis on weapon systems in support of operational testing conducted by AFOTEC for C-130J, CV-22, and C-130 modernization programs. Conduct and execute methodologies that determine maintainability, reliability, code quality, maturity, and process (CMM) level implementation.

*Jul 1998 – Apr 2000, Masters Student, AF Institute of Technology, Dayton, OH.*  
Completed 75 course hours with a 3.86 GPA to earn a Master's of Science degree in Computer Engineering. Pursued 3 different elective tracks to include: database-information retrieval (IR), software engineering, and artificial intelligence (AI). Thesis work focused on practical application of object-oriented database technology (OODBMS), agent oriented systems (AOIS) design, and object-oriented data modeling (OOA/OOD) to a real-world AF problem.

*Jun 1996 – Jul 1998, Commander, Information Systems Flight*  
Assigned as manager of an 11-person development team specializing in Graphical User Interface (GUI) software in Ada95 and Open-GL. Planned, organized, and directed upgrade of existing COBOL MIS application suite for base-level applications into 3-tier, RDBMS, Web-enabled architecture.

*Aug 1994 - Jun 1996, Chief, Data Administration Section*  
Part of a software development and support team; took over database administration Managed the Oracle Relational Database Management System (RDBMS) on VAX, IBM, and SGI, and Windows NT platforms. Oversaw the installation, upgrade, and support of database systems that service over 500 users. Lead developer for an Oracle RDBMS based application used by over 150 users.

*Aug 1990 - Aug 1994, Simulation Analyst, AF Wargaming Institute, Montgomery, AL.*  
Developed GUI front-end/RDBMS back-end application and support tools to execute theater-level seminar wargames played by over 700 students annually. Responsible for the installation and execution of wargame software at the Command and General Staff College (Ft. Leavenworth, KS), the Royal Air Forces Staff College (RAF Bracknell, UK), and the Canadian Forces Staff College (Toronto, ON).

*Jun 1986 - Jun 1990, Cadet, US Air Force Academy (USAFA), Colorado Springs, CO.*

### **Degrees Conferred**

*Bachelors of Computer Science (B.S)*, May 1990  
U.S. Air Force Academy, CO

*Masters of Business Administration (M.B.A)*, December 1996  
University of Phoenix, Nellis AFB Campus

*Masters of Science in Computer Engineering (M.S.C.E.)*, March 2000  
AF Institute of Technology, Wright Patterson AFB, OH

*Doctor of Philosophy in Computer Science (Ph.D.)*, Fall 2006  
Florida State University, Tallahassee, FL

### **Instructor Experience**

*2000 – 2002, MIS Faculty Member, National American University*

Rio Rancho, NM Campus / Albuquerque, NM Campus  
Fall 2000, CI2350 Intro to UNIX

Winter 2000, CI1420 Principles of Programming

Winter 2000, CI3520 Programming in C/C++

Winter 2000, CI4070 SQL Server Administration

Spring 2001, CI3520 Programming in C/C++

Spring 2001, CI4520 Advanced C/C++ Programming

Summer 2001, CI1150 Introduction to CIS

Summer 2001, CI1420-A Principles of Programming

Summer 2001, CI1420-B Principles of Programming

Fall 2002, CI1420 Principles of Programming

Fall 2002, CI2490 Structured Query Language

Fall 2002, CI4680 Advanced JAVA Programming

Summer 2002, CI1420 Principles of Programming

Summer 2002, CI3680 JAVA Programming

Winter 2002, CI1420 Principles of Programming

*2002 – Present, MIS Faculty Member, University of Phoenix Online Campus*

2002/07, POS370 Principles of Programming

2002/10, POS370 Principles of Programming

2002/12, POS370 Principles of Programming

2003/08, POS370 Principles of Programming

2003/10, POS370 Principles of Programming

2004/06, POS370 Principles of Programming

2004/07, MTH208 College Mathematics I

2004/12, CSS561 Programming Concepts

2005/11, CSS561 Programming Concepts

2006/11, MTH208 College Mathematics I

*2006 – Present, Assistant Professor, AF Institute of Technology, WPAFB, OH*

2006/10, CSCE593, Introduction to Software Engineering

## **Publications**

J. T. McDonald and A. Yasinsac, "On the Possibility of Perfectly Secure Obfuscation for Bounded Input-Size Programs," submitted to 4th Theory of Cryptography Conference (TCC'07), Feb. 21-24, 2007, Amsterdam, The Netherlands (decision October 2006).

J. T. McDonald and A. Yasinsac, "Program Intent Protection Using Circuit Encryption," to appear, in *Proc. of 8th Int'l Symposium on Systems and Information Security*, Sao Paulo, Brazil, Nov. 8-10, 2006.

A. Yasinsac and J. T. McDonald, "Foundations for Security Aware Software Development Education," in *Proc. of the 39th Annual Hawaii Int'l Conference on System Sciences (HICSS'06)*, 2006, p. 219.

J. T. McDonald and A. Yasinsac, "Of Unicorns and Random Programs," *Proc. of the 3rd IASTED International Conference on Communications and Computer Networks (IASTED/CCN)*, Marina del Rey, CA, October 24-26, 2005.

J. T. McDonald, "Hybrid Approach for Secure Mobile Agent Computations," in *Proc. of the Secure Mobile Ad-hoc Networks and Sensors Workshop (MADNES '05)*, vol. 4074 of Lecture Notes in Computer Science, 2005, Springer-Verlag, pp. 38-53.

J. T. McDonald and A. Yasinsac, "Application Security Models for Mobile Agent Systems," in *Proc. of the 1st Int'l Workshop on Security and Trust Management*, Milan, Italy, 2005, Electronic Notes in Theoretical Computer Science, vol. 157, no. 3, 25 May 2006, pp. 43-59.

J. T. McDonald, A. Yasinsac, W. Thompson, "Mobile Agent Data Integrity Using Multi-agent Architecture," in *Proc. of the International Workshop on Security in Parallel and Distributed Systems (PDCS 2004)*, San Francisco, CA, 14-17 September 2004.

W. Thompson, A. Yasinsac, J. T. McDonald, "Semantic Encryption Transformation Scheme," in *Proceedings of the International Workshop on Security in Parallel and Distributed Systems (PDCS 2004)*, San Francisco, CA, 14-17 September 2004.

J.T. McDonald, M. Talbert, "Agent-Based Architecture for Modeling and Simulation Integration", in *Proceedings of the National Aerospace & Electronics Conference (NAECON 2000)*, Dayton, OH, Oct 2000. (2<sup>nd</sup> Place Best Paper for NAECON 2000 in the Student Award Category)

J.T. McDonald, M. Talbert, and S. Deloach, "Heterogeneous Database Integration Using Agent-Oriented Information Systems", in *Proceedings of the International Conference on Artificial Intelligence (IC-AI'2000)*, Las Vegas, NV, Jun 2000.

J.T. McDonald, "Agent-Based Framework for Collaborative Engineering Model Development", MS Thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, AFIT/GE/ENG/00M-16, March 2000.