FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

# BUILDING TRUSTED COMPUTER SYSTEMS VIA UNIFIED SOFTWARE INTEGRITY PROTECTION

By

JONATHAN JENKINS

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

Degree Awarded:
Summer Semester, 2015

Jonathan Jenkins defended this dissertation on April 20, 2015.
The members of the supervisory committee were:

Mike Burmester

Professor Directing Dissertation

Washington Mio

University Representative

Sonia Haiduc

Committee Member

Xiuwen Liu

Committee Member

Ashok Srinivasan

Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with university requirements.

This work is dedicated to my parents, and to the future. I further thank my advisor for his patience in working with me and my committee members for their suggestions and feedback. Since I have a practice of abundant thought before action, this work will almost certainly never be complete

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF LISTINGS

# ABSTRACT

The task of protecting software integrity can be approached with a two-part strategy that addresses threats to the *static* integrity of memory contents and the *dynamic* integrity of software memory interactions. Although their resultant effects are realized in fundamentally different ways, attacks on either *memory contents* manipulated by instructions or the *operation* of software are both damaging and facilitate further attack. The ability to alter the static memory state (programs, configuration data, etc.) of software opens an attack vector as broad as the capabilities of the attacker. Altering the operation of running programs (e.g. control flow) allows an attacker to divert the results (memory effects) of the target program from those which were intended by the program author.

Neither static nor dynamic analyses of integrity are alone sufficient to completely describe the integrity of trusted system software. Further, there is a characteristic facilitation of vulnerabilities between the two classes of software integrity in that common security violations are decomposed as sequences of static and dynamic integrity violations which have enabling relationships.

In order to capture and provide a *unified* software integrity, this work will analyze software integrity and detail techniques to enable protections to be applied by a coherent, systematic framework directly to the memory and memory interactions which exhibit software integrity rather than tailored to each member of an evolving set of particular threats.

# CHAPTER 1

# INTRODUCTION

A trusted computer system is expected to be able to provide reliable evidence of its ability to satisfy some set of requirements with respect to the integrity of its state and operation in addition to merely demonstrating the ability to produce usable outputs. Put another way, the notion of trust in a system is based on *justified* confidence in the correct behavior of the software and hardware in operation.

The problem of general software integrity protection involves addressing a complex of inter-related yet fundamentally distinct threats. This means that the analysis and the ultimate solution will require a coherent treatment of the related and disparate aspects of computer system software integrity. In the same way that human society's greatest problems demand the elevation of strategies to consider several aspects of their manifestations from corresponding viewpoints, the task of constructing foundations for the general integrity of computer system software can not rely on a single domain procedure or depend on particular attack profiles. Our greatest achievements in the maintenance of integrity within complex systems (and all great achievements) result from the combination of key patterns of thought and the recognition of the different faces the problem presents.

Building trust in the integrity of computer system software is a multi-faceted task. One way to approach this task is by enumerating the distinct manifestations of the integrity of the system as they are exhibited in real systems. Software is statically manifested as program code residing in memory. The integrity state of system memory is readily captured by the compliance of the set of memory contents resident on a system with a reference. Confidence in the *static integrity* of the system is derived from verification that the resident programs and data are in their intended state in memory when subject to unauthorized modification. Conventional methods of verification of this property are based on processes which take the program as input, producing a value which is representative of the literal memory state of the program, yet are more compact than mere reproductions of the program contents. Implementations of such verification take the form of, for example, digests such as Message Authentication Codes (MAC).

The *dynamic integrity* of software, on the other hand, is manifested in the soundness of the run-time execution of program code, which is dependent on yet fundamentally different from its static representation. Unlike inert program text residing on disk, run-time software activity interacts with machine components such as working memory, registers, and CPU. While static program code and binaries merely reside on disk as the basis for execution, running software is characterized by active representations such as processes, which occupy their own working memory space and allocate variables as needed during operation. Ultimately, dynamic aspects of computer system software execution reveal themselves as *interactions*, the nature of which is inferred from changes to system memory. The integrity of dynamic aspects of computer operation can be captured as the compliance of software behavior (as manifested by changes in system memory) with the intent of the software creator(s).

It is possible to measure some aspects of running software (e.g. control flow). However, depending on the measurement of interest, significant overhead or instrumentation of the program code (or alteration of the run-time environment) may be required as described in [1]. In addition, the required adjustments may be unacceptable for certain usage scenarios (e.g. critical infrastructure). Still, it is possible to robustly measure control flow as described in [1], which provides robust protection of program control flow based on static program checking and machine code rewriting. Control flow is a key interaction between memory elements which in part composes dynamic software integrity, and the protection of control flow will be a necessary component of any scheme aimed at the preservation of dynamic software integrity. In any scheme attempting to provide unified integrity, the dynamic manifestations of software represent a fundamentally different challenge from the static, with respect to models and protections.

The spread of untrusted (or *tainted*) data by the system also composes an aspect of dynamic integrity. Tainted data can be described as data that is untrusted by virtue of its source or merely its type. Although the use of tainted data does not imply any defect in the integrity of strictly the platform software itself, the use of tainted data is a threat to the integrity of the system because it may weaken the guarantees that are possible about the operation of the system. Such use can also harm the integrity of the results (outputs) produced by the software. Finally, the dynamic integrity of procedure calls consists of the compliance of the memory state effects associated with calls with the intent of the program creator.

The comparison of the static and dynamic manifestations of software suggests a strategy to protect the overall integrity of software. A synergistic, coherent combination of methods aimed at ensuring the unified integrity of software has the potential to be a comprehensive solution, where the level of security provided is based on the specific measures taken for each type of integrity (static and dynamic).

Trusted computing (TC), popularized by the Trusted Computing Group [73], is a paradigm for increasing the level of trust in computing platforms via a combination of robust (typically hardware-based) *Roots of Trust*, reliable integrity measurement, and reliable reporting of integrity information. TC is an architecture that can be the basis for increased trust in the load-time (static) state of program code. As such, architectures such as TC are a component of the integrity protection scheme in this work.

In this work, the construction of the above described protection scheme will be proposed, and potential benefits and disadvantages shown. As will be revealed, neither static integrity protection methods such as Trusted Computing or dynamic integrity methods are alone sufficient for any form of unified software integrity protection strategy, or for the protection of particular contemporary attacks.

## 1.1 Existing Integrity Models

There are multiple useful yet incomplete existing definitions of software integrity, each appropriate for describing the desired property in the terminology of one aspect of the machine. There are correspondingly several concrete aspects of computer systems that are relevant to integrity, but the methods currently employed to measure integrity cannot be applied across boundaries to other aspects with the expectation of producing a measurement of the same quality. For example, applying a message authentication code (MAC) to verify the intactness of a software program on disk is a valid application of integrity protection in the static domain, but applying a MAC to measure the integrity of run-time working memory presents major obstacles (particularly that useful reference measurements may not be possible to produce). In this section, existing treatments of integrity are discussed and related to the present task of unified software integrity protection.

Existing research advances *partial* and conceptual definitions of integrity that are independent of the *state of action* (either static or run-time) of the software or hardware (many are summarized

in [36]). Jeremy Jacob relates the concept of dependability to integrity by defining the latter as justified confidence in the ability to provide a *service*, which is captured as maintaining freedom from improper alterations [44].

The landmark work of Clark and Wilson defines integrity in terms of *external consistency* [20]. In [20] separation of duty and well-formed transactions are employed to maintain the link between "data" within systems and the "real world." Well formed transactions manipulate data in ways which preserve integrity. If a security sensitive set of tasks is entirely entrusted to a single entity, the equivocation of that entity alone suffices to violate security policies and thus break the link between the "data" in computers about security-sensitive events and the "real world" events that truly occurred. Dividing task responsibility among several entities creates a requirement for *all* entities to collude in order to corrupt the overall process.

Prior to Kenneth Biba's landmark integrity analysis in [4] the problem of addressing the security of a military computer "utility" focused on the *dissemination* of information. In [4], maintaining the *validity* of information is the major goal. Computer subsystems must be caused to adhere to a standard of correct operation.

Threats in the Biba model are classified by source and type. The source of the threat may be external, such as when a remote subsystem attacks a local one, or internal to a subsystem. Type may be direct as in a malicious write into a resource, or indirect as in a corrupting modification to data used by a subsystem. Independent of category, integrity violations are captured as "unpriviledged, intentionally malicious" modification (sabotage).

A crucial and necessary observation of Biba is that the judgement of the propriety of a modification is dependent on the individual protection problem faced. The protection problems addressed by Biba are driven by the context of a secure military computer utility. As a result, considerations of national security and user identity are the basis for the characterization of "proper" modification.

Various integrity policies aimed at addressing these protection problems are presented in [4]. The mandatory policies of Biba are expressed as axioms constraining the ability of subjects to exercise capabilities of observation, modification, and execution. The axioms require that the integrity levels of subjects obey particular relations with respect to the levels of objects and other subjects in order for the given action to be allowed. For example, the low water mark integrity policy allows subjects to modify objects only if the object's integrity level is at most that of the

subject. Further, the integrity level of subjects after accessing an object is set to the minimum of the subject's pre-access integrity level and the level of the object accessed. Thus, the integrity level of a subject can only decrease, causing unfortunate consequences for the construction of secure, practical software systems.

In order to contrast the work of Biba with the present work, note that Biba applies *policies* to subjects and objects to ensure that only proper interactions with data and programs will occur, while the threat model of the present work admits that in real systems integrity violations occur regardless of the authorization or policies applied, even with the assumption of certified software. The present work thus chooses to focus on the various forms of integrity exhibited by computer systems, applying detection and protection in a systematic way to integrity as it is exhibited natively.

## 1.2 Universal Composability

There exists an important formal definition of security which is based on the ability to securely compose multiple protocols running concurrently [10, 51]. The work of Canetti in particular presented an influential and unique paradigm for protocol security: Universal Composability (UC) [10].

In that work, security is built upon the indistinguishability of protocol operation between an ideal model of operation and a real-world model, as observed by an *environment machine*. Composability is a desirable property of a security notion which means that security is preserved when the analyzed protocol is caused to run alongside other protocols (including instances of the same protocol). The UC model and definitions bear similarities to this work in terms of broad goals, but differ in important ways.

The work on Universal Composability defines a security framework including a real world process model for the protocol, an ideal process model to represent the security requirements of the protocol, an adversary, an environment, and the parties running the protocols. The framework in [10] is a basis for the notion of *securely realizing* a task, which is achieved when running the protocol in the *real world model* emulates *the ideal process*. The F-Hybrid model is an extension to the real world model where parties may access *multiple* concurrent instances of the functionality that has been securely realized. The universal composition theorem in [10] provides that running an arbitrary

protocol composed with a protocol securely realizing an ideal functionality is equivalent to running the base protocol in the F-Hybrid model. As mentioned, the theorem is based on indistinguishability of these two runs with respect to an environment.

With its attempts to model and guarantee secure composability, the UC model can be applied to the global task of securing the interoperation of the innumerable set of protocols (programs) running concurrently on typical modern computer systems. Both the UC model and the present work are able to be applied to secure the imposed interactions between executions of software instructions interacting with a common memory resource.

The key distinction is that the present work provides a method to protect the compliance of software with intent by revealing and analyzing the memory interactions *inherent in the instructions and interactions within execution*, while UC can provide a way to formally model how to maintain security in the presence of separate instruction executions.

In the present work the focus is intentionally limited to integrity, and the method of modeling and protecting integrity is fundamentally different. The UC model treats processes as *stand alone*, focusing on their ability to be composed securely. The present work analyzes abstractly the integrity of the functional programs and processes themselves, and all of the components of computer systems from which integrity-relevant phenomena emanate. The present work seeks to protect the integrity-relevant components of systems that programs ("protocols") must interact with as well as the run-time integrity of the operation of the programs. Further, the UC model employs Interactive Turing Machines in order to model the communicating parties involved in the protocol executions, while the present work objectively focuses on the components and behaviors that manifest integrity. Finally, the UC model of security hinges tightly on the ability of an environment machine to feasibly distinguish between protocol runs. This is in contrast to the role of the program static state and the program creator in the present integrity analysis. These two are the ultimate source of the integrity requirements (and by extension the measures taken to check integrity) and the integrity reference measurements, respectively.

Ultimately, the contributions of the present work are the revelation and protection of the integrity native to memory and memory interactions inherent in software. The capture of the interactions *between separate programs* is limited to the procedure call interface presented by software. The present work realizes its benefits at a lower granularity than the UC model, but nevertheless

both strategies can be applied to the secure operation of software, with UC able to more broadly reason about interactions of separate executions.

# CHAPTER 2

# INTEGRITY THREATS

A precise enumeration of the threats to computer software integrity is necessary in motivating the task of analyzing and protecting unified software integrity. Thus, a discussion of the threats faced by software which directs alterations of and interactions between memory is presented here.

Attackers can compromise memory contents on the platform by inducing unauthorized modification, or attack the operation of programs by for example exploiting vulnerabilities in software to violate its intended control flow. In addition, it is possible to abuse control data used by programs in order to effect undesirable states of operation. Often, these threats are interdependent, such as when a hijack of the control flow of the system allows the attacker to execute code which alters memory contents on the system.

The discussion of threats to software integrity here introduces the fundamental decomposition that is foundational to the general integrity protection strategy created in this work. Although only threats are discussed here, the categorization of threats by the state of action of the integrity threatened facilitates the application of protections at the precise locations (possibly multiple) where they are required. Further, the detailed decomposition of integrity to follow in later chapters allows the insertion of protection at advantageous instants during sequences of integrity-relevant events.

## 2.1 Static Integrity Threats

*Property 1. Presuppose that a memory region is not subject to authorized modification in the interval of interest. The contents of the memory region (program and/or data) on the platform at the instant of measurement have not been modified in an unauthorized manner and therefore match a state provided in a literal reference measurement*

Static integrity threats hold the potential for violations of Property 1 via unauthorized modifications of memory region contents with respect to a reference of correct state. Inherent in this definition is a lack of exposure to legitimate modification within the interval of threat exposure. If modification is allowed during the exposure interval, no violation of static integrity is implied when the memory region state undergoes change.

A static integrity threat may appear as a consequence of a localized attack via software interaction on platform, or may be carried out via interaction between network endpoints.

Although strictly the presence of unauthorized memory state on the platform does not imply intentional attack (for example memory corruption or errors caused by software), even a single incorrect bit (with respect to reference) within program code or data is a static integrity violation. Thus, environmental factors or unintentional memory corruption are legitimate static integrity threats, but they are merely one category of such threats.

It is important to draw a distinction between integrity threats that merely violate Property 1 and threats that add additional attack steps such as control flow diversion. This is because although an integrity violation with respect to Property 1 may not identify the type of attack in progress or even represent the ultimate attack goal, defenders of system integrity can take meaningful action based on the recognition of static integrity threats, even if the action is merely to halt operation. In addition, as will be revealed in this work, static integrity violations are frequently members of larger sequences of inter-related integrity violations of various types. A systematic software integrity protection cannot ignore the fundamental danger and predictive power of static integrity violations.

Finally, with the assumption that memory is initially intact (matches its reference), violations of static integrity imply state changes in memory regions. By definition, static integrity violations arise from unauthorized *modifications* of memory state with respect to a literal reference. As will be revealed, transitions in dynamic integrity (or violations) imply no such change in memory state.

## 2.2 Dynamic Integrity Threats

Threats to dynamic integrity alter the executions of computer software in its imposed interaction of system memory regions such that the intent of the program creator is violated. These are threats to the desired sequences of memory effects and actions that compose the programmatic goals of the program. There are several major categories of threats to dynamic integrity, arising from attacks on data and the continuity of execution.

### 2.2.1 Control Flow Threats

In a typical model of computer run-time operation, execution proceeds from instruction to instruction, describing a *control flow* pathway defined by sequential execution order as well as

control instructions such as *if* and *for* in C. The dynamic integrity of control flow is defined by the adherence of abstract links between instructions executed to the paths intended by the program author(s). A sequence of actions or state changes that diverts such links may cause the flow of the execution "state machine" to make a transition to a set of compromised instructions or deviate from the operation intended by the program creator.

Violations of control flow integrity are defined as deviations from correct control flow as expressed by any of several measures such as creator directive, program control flow graph, or call graphs. Control flow graphs for static program code explicitly state all control flow transfers (e.g. jump or branch) present in the program logic (some transfers are defined at run-time), and are thus a suitable and feasible (yet imperfect) reference for program control flow, given the existing capabilities of compilers.

In a typical control flow attack, the attacker causes an execution to initiate a sequence of memory effects which corrupts control data or alters the control effects of a software interaction. Although this action does not imply a violation of program intent (only the alteration of operation), even alterations not perturbing the program creator's intent are a threat because such changes are commonly members of sequences of state changes that collectively implement attacks on control flow. Further, the gap between control flow operations expressed within program code and those intended by the program creator admits the possibility of perturbations in control flow which remain compliant with the creator's directive of intent, but not with the explicit program code control flow or control flow graph which can at best imperfectly express the program creator's intent.

From the integrity protection perspective, flaws in software act as key interfaces to the control flow of target programs. This interface is exploited by attackers via the supply of aberrant data or the invocation of logic in a manner so as to exercise program memory effect sequences which can alter control data without authorization. Examples of attacks demonstrating the threat to control flow integrity are discussed below.

The most notorious method of achieving a control flow diversion has been the buffer overflow [58]. Commonly, the attacker abuses the behavior of existing program code which writes to memory locations. More specifically, unless the developers of the victim code (and those of all attached libraries) have properly checked the length bounds (and data types, etc.) of write operations and ensured that destination memory areas hold sufficient space, the write operation may write past the

bounds of the specified destination location and over-write adjacent memory areas. The overwriting of return addresses on the run-time stack with addresses (pointing to programs) of the attacker's choice completes the hijack of the running process control flow. When the compromised return address is used as a control transfer destination, the integrity of control flow is lost.

With a *heap-based* buffer overflow, the attacker can take advantage of weaknesses in the management routines of functions designed for dynamic memory allocation. This can result in overwriting unintended memory areas of the heap, potentially causing incorrect allocation or deallocation of memory. Integer overflows, where storing values too large for a given data type result in subtle bugs in program code, generally do not provide a way to directly effect a control flow hijack, yet can facilitate them via e.g. bypassing an array bounds check that would normally protect from buffer overflows.

An additional threat is represented by attacks which extend deviations in control flow, rather than initiate them. With the key assumption that control flow has *already* been hijacked by the attacker, techniques such as Return Oriented Programming (ROP) [63] involve assembling generalized attack code from existing program code (from e.g. system libraries) on the platform. This technique relies on locating existing "gadget" instruction byte sequences on the platform that can be chained together using manipulation of control flow addresses on the stack (return addresses). The gadgets can be used to construct arbitrary computation sequences in order to implement attacks. As such, ROP extends dynamic integrity threats to control flow, provided that an initial diversion of control flow is assumed.

### 2.2.2 "Non-Control Data" Attacks

There is great attention given in available research [24] and response efforts [13] to control flow attacks (particularly buffer overflows), and such treatment is justified by the threat of an attacker diverting execution to code of his choice, gaining complete control over the affected machine.

Similarly damaging outcomes can be produced by attacks on "non-control data." Attacks on control flow work by corrupting either the machine's interface to control-relevant functions or the data used by those functions. However, it is not necessary to directly corrupt the targets to effect a comparable compromise. In [17], the authors provide examples of attacks on user identity data, configuration data, decision data, and user input data that represent a realistic threat, with comparable severity to control-data attacks such as buffer overflows.

Clearly, software programs use data produced externally during normal operation. Whether the focus is on configuration data, user identity data, or decision data, an attack on the non-control data used by a program represents a violation of *the requirements of the interface* the program presents to any input that it uses. This interface can be seen to be composed of a set of requirements for proper or "correct" operation given that the input is accepted, with restrictions on data type, quantity, structure, value or semantic meaning, and possibly more attributes. A violation of such an interface is a threat here because it impacts on the integrity of instantiations of the *assumptions* of the program. With key assumptions broken, any expectation of integrity of the outputs or operation may be lost.

In [17], the authors have demonstrated non-control data vulnerabilities in real-world programs, and there are almost certainly many more such vulnerabilities within other software. Here, we focus on the categories of non-control data attack, and how they are realized.

User identity data is used by programs for e.g. access control or authorization. One attack on user identity may be to overwrite this data in order to allow normally unauthorized operations.

Configuration data such as file paths are common across many types of software. Corrupting this data can enable an attacker to control which external software is loaded, or which separate input data is used, clearly opening the way for severe compromise of the platform.

Decision data can be generally described as data that is directly used as input to a decision in relation to the user or his agent. Checks may have already been completed, and the decision data should now hold the "final decision" within memory region available to the software. Corrupting the decision data at this point clearly allows an attacker to change the implied programmatic consequences of the decision.

Finally, in a similar way, the attacker can corrupt user input data in a time of check to time of use (TOCTTOU) attack. Input validation is a standard software engineering practice that attempts to prevent the use of malformed or invalid input by checking aspects of the input such as length, type, or structure. If the attacker can corrupt the user input *after* the validation check has occurred, the program using the input may be vulnerable to other attacks such as control flow. Again, the requirements for correct interaction and results with respect to the program are violated.

### 2.2.3 Environmental Threats

Since the threat model in this work includes any unauthorized modifications to memory contents, changes in memory state or control flow resulting from for example hardware errors or abnormal electrical power events cannot be ignored. Nevertheless, the probability of such events can be assumed to be very low, meaning these events are essentially not practical threats in this work.

## 2.3 Advanced Persistent Threats

The exploitation of computing services used by individuals with access to critical information has led to the definition of the Advanced Persistent Threat (APT) [5, 26]. Rather than directly pursuing the violation of the operation of target systems, this form of attack relies heavily on the acquisition of the ability to leverage legitimate access outward to the true targets holding valuable information. Thus, the APT maximizes the use of permitted actions in order to avoid detection during the exfiltration of information. The use of social engineering or "spearfishing" techniques manipulate human actors toward the allowance of access to their priviledged roles in systems.

Although the APT is not an integrity threat, the use of malware or software exploitation is a component of such attacks. As a result, integrity protections serve to interrupt the threat at instants where the operation or contents of software is violated, but human behavior-based violations must be addressed by policy, monitoring, logging, and analysis.

# CHAPTER 3

# EXISTING SOFTWARE INTEGRITY PROTECTIONS

Before introducing applications which demonstrate the efficacy of systematic static integrity protection and lead to the main task of addressing dynamic integrity, existing techniques which protect computer system software integrity must be surveyed.

Existing software integrity protections (in contrast to the work proposed here) are typically applied to particular security-relevant aspects of the system, and are often tied to specific threats or attacks.

There are targeted defenses against specific critical attacks such as buffer overflows (e.g. memory techniques to indicate overwrite), yet these defenses tend to have either increased cost or significant limitations (or both).

Simple alterations to the software development environment also provide a form of protection. This adjustment effectively embeds protections within the programs produced by leveraging the work of software development environment tools. Programs can be written with "safer" high-level languages that include additional security checks such as Java. Security protections for e.g. memory errors are extended to all programs produced by merely switching the language.

Despite their limitations, existing protections must factor into a discussion of the development of a systematic, attack-independent software integrity protection, and serve to reveal through analysis the properties of a systematic framework to be developed.

## 3.1 Attack-Specific Protections

Stackguard [21] advances the idea of protecting return addresses on the run-time stack with *canary* values, placed adjacent to the data value needing protection. Since overwriting contiguous stack bytes in a sequential fashion is one common way of overwriting control flow data, checking the canary word before returning to calling functions is a valid integrity check.

However, (as the authors elucidate) this form of checking ensures no integrity if attackers can find ways to precisely target the return address without affecting the canary. Further, attackers

may target other control data such as function pointers or longjmp buffers. Similar techniques exist to provide targeted protection to the return addresses and also function pointers [75, 35], e.g. by relocating return addresses to seperate non-vulnerable memory and logically rearranging process memory layout to prevent buffer overflows from reaching adjacent values, respectively. Nevertheless, these techniques are similarly limited in that attackers can precisely target addresses using pointers: a pointer which points to the attack target can be used to directly corrupt the target.

## 3.2 Architectural Protections

Since attackers commonly attack running software by inducing the execution of memory contents provided as data yet holding program instructions, disallowing execution of working memory areas is a relevant technique for ensuring the integrity of execution (one version is [31]).

The damage inflicted by the imposition of unauthorized data in memory is significant to any comprehensive integrity analysis, yet it is in particular the *subsequent execution* of the attacker's code that often produces the intended damage. The attack code, which often sits in a crafted form on the stack, may upon execution spawn shell terminal sessions causing damage limited only by the imagination of the attacker. Disabling execution of data memory areas thwarts any such attack that relies on the execution of attacker-crafted contents in memory.

This style of protection has been implemented in many forms, via support by operating systems and hardware microprocessors. This "never execute" feature is supported by major operating systems and processors, and even required for operation in some cases ([12]). Never execute is often implemented with a bit within virtual memory pages indicating whether the page should be allowed to execute.

The proposal to create an "architecture" for detection of "semantic" integrity violations in [59] is the most similar identified work to the present work in terms of goals, but the work there is limited to operating system kernels, and focuses on the protection of the literal contents of certain critical data structures which undergo change during run-time. The work proposed in [59] therefore protects the static integrity of system data, rather than the memory interactions which compose executions of instructions. Thus, dynamic integrity as modeled in the present work is not addressed.

## 3.3 General Anti-Malware Protections

Conventional security software (i.e. anti-virus or firewall) plays a relevant role in maintaining computer system software integrity by the application of several reliable and well accepted techniques that provide protection against integrity threats. Matching software behavior or static state to known signatures of an attack is one method employed by security software.

The protection provided by such anti-malware architectures covers both data and to a limited extent operational aspects of the platform software, because it is possible to detect both changes to the set of programs resident and unauthorized run-time conditions such as the presence of unauthorized communication channels (e.g. abnormally open ports). Signature matching is an effective and efficient technique for identifying malware because the creation of machine-matchable signatures is feasible for most threats, as is the matching process (the matching process requires an acceptable amount of computation time). Creating a malware signature can be as simple as specifying a regular expression or a digest value derived from analysis of the attack code. Further, signature matching is readily usable for static representations of software since the generation of static signatures is widely implemented and well understood.

On the other hand, if no signature currently exists for a particular attack, this method of detection provides no security. Further, this technique requires implementors to continuously create signatures for all dangerous members of a large, growing software space that is composed of programs from innumerable sources. This is an unavoidable limitation of signature matching due to the evolving nature of current integrity threats such as virii, more particularly the signatures of these threats. If an adversary creates new attack code, a meaningful signature scheme would derive from the new code a unique signature differing from the signatures for existing code from this adversary.

Another approach to maintaining system integrity is based on anomaly detection, where defenders attempt to characterize a *normal* or authorized state of operation that can by comparison be used to identify attacks [14]. The task of the construction of such a normal state of operation amounts to the creation of an integrity reference against which ongoing measurements of state and operation are compared.

The difficulty of forming an accurate characterization of the normal state is a critical factor in the effectiveness of anomaly detection. Since distinguishing normal from abnormal is often an

unclear and imprecise process, there is the risk of excessive false positive malware identifications with this technique.

Beyond the difficulty of establishing a meaningful partition for normality, adversaries may adapt to the assumptions of detection by transforming their imposed system behavior to resemble normal behavior. The characterization of normal operation itself must often "adapt" to the conditions of the application, as in intrusion detection, where the profile of normal network communication traffic may change in time.

Finally, the input data being analyzed may contain extraneous "noise" values that resemble anomalous ones, yet are not meaningful inputs. Part of the task of processing system profile data for anomaly detection is the imprecise process of filtering the non-meaningful noise data, preventing it from being considered during the detection process.

## 3.4 Tracking Untrusted Data

Data used by software on the system can represent an integrity threat by virtue of its source or type, or where it is used. This idea is the basis for *taint tracking* techniques that attempt to identify and track untrusted (tainted) data as it enters the system and is used (e.g. for addressing, arithmetic, logic) [18, 56, 19, 69, 25, 47, 74].

If certain devices, remote (network) data sources or memory are considered tainted, operations propagating or using data from those sources are correspondingly suspect. A salient application of taint tracking is the detection of a corrupted address on the run-time stack, the result of a buffer overflow. This detection technique prevents the bad address, which may point to malicious attacker code, from being used as a return destination.

Since taint tracking must recognize tainted data as it is used in general purpose operation across sequences of instructions, basic data about taint state must be maintained in addition to the data itself. The taint data is typically maintained in a *shadow memory* with a set of bits dedicated to taint attached to conventional memory items.

Tracking taint has been implemented via interpreters such as the Perl programming language, hardware extensions to processors, and instrumentation of source code or binaries such as in [62, 18, 56]. Dynamic data flow tracking [47] is a prominent technique for the implementation of taint tracking, but also sees application outside of security in the analysis of interesting data flow during

program execution. The analysis of taint need not be constrained to single programs. Untrusted data may also propagate between processes and across network links, driving the creation of larger scope taint tracking systems [41, 61, 78].

Each taint tracking method discussed here carries limitations that affect its ability to protect software integrity. The Perl interpreted language supports tracking taint [60], but clearly presents a language-dependent solution and brings the performance cost of interpretation along with security support.

Introducing extensions to processors offers efficient support for taint tracking, yet changing hardware engenders resistance due to cost, and other factors. Processor producers face numerous considerations in determining the set of supported instructions and capabilities, notably some environment properties outside of their control. Still, hardware support for taint tracking shows promise under appropriate circumstances, and will thus be further explored in this work.

Instrumenting existing program code to add checks for taint is an attractive method because the process merely adds an additional phase to the production of the program code. However, the overhead of the tracing process during run-time cannot be ignored, with performance slowing by a factor of between 10 and 30 using the techniques referenced here [18, 56]. The use of virtual machines or emulators is a technique employed for cross-network taint tracking, yet one which results in significant performance impact [41].

In spite of the barriers to adoption of security measures by computer hardware producers, the capabilities of architectural approaches to taint tracking cannot be ignored (indeed, their capabilities for measurement of other integrity forms will be shown to be critical in this work) [19, 69, 25, 74].

The work of Suh et al directs the operating system to tag untrusted data on arrival, while processor extensions track the flow of taint during execution by manipulating and analyzing taint tags [69]. Their technique posits taint transmission events in dependencies based on copy, computation, and load or store operations. The technique is demonstrated to detect a broad set of contemporary attacks which exhibit the use of spurious data, such as format string attacks and buffer overflows.

Computational dependencies (tainted data used during computations) cause significant additional tracking performance overhead relative to other dependencies. Thus, the authors evaluate a security policy with and without such events. Nevertheless, in [69] the simulated worst case

overhead for the protection is 6%, with an average overhead of 0.8% even for the more expensive policy which by tracks computational dependencies in addition to the others.

## 3.5 Constraining Software Execution

Enforcing a software safety property called *control flow integrity* (CFI) is a robust method of protecting the integrity of software control flow [79, 1]. Whereas unauthorized modification of branch destinations is a primary starting point for attack, protecting control flow means that such addresses must be checked before use. If a reference of the correct control flow of a program can be created, the control transfers of the program can be checked to adhere to the correct path dictated by the reference.

In [1], a *control flow graph* is generated with static binary analysis (the implementation uses Vulcan [33]), and machine-code rewriting of binaries is employed to instrument the target program with checks to ensure the safety property is maintained for all indirect branches which involve a jump to a location not explicitly listed in code. The checks inserted into program binaries ensure that control flow destinations are valid addresses, as measured against the control flow graph generated before run-time. Constant destinations can be checked statically in executable code, while addresses computed at run-time must be checked dynamically.

The measured computational overhead associated with enforcing control flow integrity as in [1] is uneven across programs, peaks at 45%, and is below the overhead for competing proposals.

However, any attack that respects the control flow of the program is unaddressed by CFI protection. It is not strictly necessary to divert the control flow of the running software in order for an attacker to take control of the system, as format string vulnerabilities demonstrate [64]. In major programming languages, there exist format functions which convert inputs of various data type between input and output. These functions can be invoked with crafted inputs and parameter parity relationships which expose or corrupt working memory used by the running program.

The compute time required for software-based enforcement (enforcement in [1] is software-based) may not be acceptable for some applications. Thus, hardware implementations are attractive for their ability to pass performance impact to processors, yet it is not realistic to expect processors to add such features presently due to complex requirements, market choices, and cost.

The performance penalty of pure CFI protection has spurred the creation of weaker "coarse-grained" versions which achieve smaller performance overhead than the original work in [1] by reducing the frequency or type of CFI checks that are enforced [27]. These techniques typically employ behavioral heuristics, reduce the frequency of checks, or adjust the type of branches which are checked (e.g. call, jump or return) with the goal of providing low-overhead CFI protection.

Various current approaches for relaxed versions of CFI are surveyed in [27]. Naturally, the relaxation of enforcement opens the way for attacks such as the recent Return Oriented Programming (ROP) technique [27, 11, 37]. Recent work effects such attacks by a) fashioning ROP code sequences which comply with the restrictions imposed by the protection or b) concealing evidence of attack.

Despite the performance consequences of pure control flow integrity protection, and the so far limited success in creation of low-overhead, secure relaxed versions, control flow integrity remains a cornerstone of any discussion of general integrity protection because it illustrates one of the several dominating aspects of the integrity of running software programs. The compliance of instruction to instruction links with program creator intent is a fundamental, distinct aspect of the integrity of software execution. As a result, control flow integrity must factor highly into the remainder of the work presented here toward the general protection of software integrity.

## 3.6 Trusted Computing

Trusted computing (TC), developed by the Trusted Computing Group (TCG), is a collaboratively developed, implementation-agnostic architecture that standardizes *Trusted Platforms* whose integrity is protected from the most basic stage of operation (BIOS) to run-time. The integrity of programs is measured by cryptographic digests to ensure their contents are intact before further stages of boot are allowed.As such, TC is a valid technique that can be applied to protect the static integrity of memory contents within computer systems.

TC is based on the introduction of hardware-based *Roots of Trust* which are engines for key integrity protection services. Although hardware-based implementations of TC technology are dominant, software instantiations are not excluded. There are three mandatory Roots of Trust specified in the TC specifications [73]: The Root of Trust for Measurement (RTM), storage (RTS)

and reporting (RTR). These Roots of Trust supply a protected set of security capabilities including attestation, measurement, and reporting.

In Trusted Platforms there is a transitive trust link between the software/hardware implementing each stage of boot and all later stages. Beginning with the code implementing the Core Root of Trust for Measurement (CRTM), each stage engages the RTM to measure the integrity of the code implementing the next stage before passing control to it. RTM measurements are generally stored within 20 byte, volatile Platform Configuration Registers (PCR). Since all Roots of Trust (RoT) are by assumption completely trusted, the code for all stages of operation (measurement of application code is also possible) can be reliably measured with TCG structures and placed within a *Trust Boundary* before being allowed to run. All Roots of Trust are specified by TCG as abstract *engines* whose implementation is left to vendors/implementors of TCG technology in hardware or software. Further, all TCG architecture components are intended to be similarly vendor-neutral and implementation-agnostic.

The use of cryptographic digests of program code for integrity protection has little value without some form of reference indicating the "correct" value of the measurement. Implementors of software requiring integrity protection are given responsibility for maintaining the reference measurements used for comparison to generated digests. Computing digests (e.g. MAC) is only one of various cryptographic operations supported by the Trusted Platform Module (TPM) [70], the basic hardware device housing the canonical TC capabilities of the Roots of Trust. The TPM specification also includes encryption and signing routines, and a monotonic counter whose values can be linked by bounding to an external time value.

The Root of Trust for Storage is a TC engine charged with managing the secure storage of data within the TPM. Cryptographic keys are the chief type of resident data within the secure area of the module. Two keys permanently embedded within the TPM are the Endorsement Key (EK) and the Storage Root Key (SRK). The asymmetric EK keypair serves to identify the platform that is *bound* to the TPM (which is bound to the platform by virtue of holding the EK). The link between the platform and the TPM is further strengthened by requiring tamper-resistance measures on the location of the TPM. The SRK is an encryption key used to protect (encrypt) other keys, and "seal" data stored outside the TPM.

In addition to reliable measurement and secure storage, a TPM is required to host an engine, the Root of Trust for Reporting, capable of reliably reporting platform integrity evidence to interested parties. The RTR must manage the exposure of the platform PCR digest values, as well as the process of *attestation*, in which the platform provides a cryptographically signed report of platform integrity measurements to a remote entity.

A TPM can hold a number of Attestation Identity Keys (AIK) for the purpose of signing (the EK is never directly used to sign or encrypt) for example integrity reports. The AIKs are derived from the EK, and are certified by a remote Certification Authority (CA). Since the TPM and its keys, the surrounding platform, and many devices and software may also require credible evidence of adherence to specifications and relevant properties, various credentials are called for in the TCG specifications. These credentials are either issued to particular platforms or TPMs to demonstrate compliance or properties, or issued to provide reference measurements illustrating "proper" operation of hardware or software whose integrity must be protected. Credentials are created by entities qualified or ideally positioned to provide them (e.g. software vendor can provide a reference measurement digest indicating a "safe" binary).

Although in the TCG model the Roots of Trust are completely trusted to carry out their operations, practical implementations of e.g the TPM and other technologies will realize levels of trust driven by the practices of the implementors and the amount and quality of review they receive. The TCG specifications also require the TPM to be protected by physical tamper-resistance measures in order to strengthen the link between the TPM and the physical computing platform it is to be bound to. Software-based implementations of TPM will necessarily employ a different type of tamper-resistance, with the only requirements vis-a-vis TCG standards being uniqueness and a convincing case that the measures are equivalent to those used for hardware versions.

Since Trusted Platforms will typically be expected to be members of computer networks, The Trusted Network Connect Working Group (TNC-WG) within the TCG actively develops standards for an architecture enabling integrity information about network endpoints to be exchanged within the network in order to inform network infrastructure access control decisions The TNC specification [71] describes a set of abstract roles that exist within the TNC model and a set of interfaces between entities and protocol or function endpoints. The Access Requestor (AR) makes its request for access to the Policy Decision Point (PDP), which will evaluate the security posture of the AR against the

active security policy. A Policy Enforcement Point (PEP) will enforce the network access decision of the PDP. There are also optional roles for Metadata Access Points (MAP) and MAP Client (MAPC). The MAP is a storehouse for various integrity and authorization information about AR hosts, while MAP Clients will interact with the MAP to publish or consume integrity information about other AR hosts. Considered at large, the TNC architecture enables entities (e.g. routers) that are faced with network access decisions to rely on robust integrity information about requestors, bolstered by the presence of TPM technology on hosts.

By virtue of its employment of reliable methods of static memory measurement, robust trust anchors and reporting functionalities, Trusted Computing is an appropriate architecture for protecting the static integrity of computer system software. The structures and functionalities of TC allow any unauthorized changes to static system memory (program or data) to be detected. Further, TC structures can secure vectors for remote attack by requiring remote hosts to demonstrate their integrity to the local endpoint before connection is allowed, in a manner standardized by TC. Any evidence of unauthorized modification can be used to trigger a defensive response, or remediation. Therefore, the capabilities of TC match closely with the task of protecting the static memory of a computer system.

The work in [28] seeks to extend the TC architecture to apply "dynamic integrity measurement" toward protection against Return-Oriented Programming (ROP) [63] attacks. The authors propose to employ tracing and tracking techniques via program re-writing to attempt to detect ROP attacks. Although the authors correctly point out that ROP attacks exploit existing program instructions on the target computer system (meaning that TC techniques cannot detect a static integrity violation linked to this exploitation), ROP attacks require as a pre-requisite a violation of control flow. Further, the fact that programs are certified as statically intact provides no guarantee about proper operation under a scenario where unavoidable flaws are exploited.

Similarly, the work in [48] attempts to extend TC attestation by proposing to measure "dynamic system properties" of programs in order to provide a dynamic form of attestation. The work in [48] proposes static analysis to create the constraints for the "structural integrity" and "global data integrity" that are chosen to represent the integrity of running programs, meaning that dynamically formed links between addresses and control data are not analyzed. Further, the focus of Kil et al. on

simple data invariants and incomplete control linkage information means that any form of dynamic integrity protection provided is incomplete, as the authors note.

There have also been attempts to provide attestation based on *properties* of systems, rather than literal configurations, such as in [16]. Although such works extend Trusted Computing and provide a more portable manner of attestation which is agnostic of particular implementations, verifiers may in some circumstances require specific configurations to be included in attestations. Works such as [16] differ from the present work in their chosen representations of integrity (including their level of detail), and strategy proposed to protect it.

## 3.7 Randomization

Program address space randomization (e.g. Address Space Layout Randomization, ASLR) has been employed for the prevention of security attacks in many contemporary systems [72, 52]. This technique randomizes the locations of key parts of a running program's address space, preventing attackers from targeting or directing control flow to predictable memory addresses. Each time the protected program is loaded into memory, at least the program base address is randomized. Since attackers commonly construct exploitations which require the diversion of execution to reach the location of crafted attack code, randomization of the locations of program addresses in memory interferes with such attacks.

Several methods have been identified to bypass the protection of ASLR. With 32 bit architectures, brute force attacks are able to identify the key addresses despite the randomization [65]. Other attacks exploit the fact that non-randomized programs are loaded by protected programs, or tailor the overwriting of stack data in order to allow the desired addresses to be reconstructed during the attack.

In [40] Hiser et al. propose to randomize the executable instruction address space for a program, confounding attacks which assume knowledge of instruction layout, such as Return-Oriented Programming (ROP) [63]. The work of Hiser et al. expands the scope of randomization relative to the earlier Address Space Layout Randomization ([72]), violating prospective attackers' assumptions of source code instruction locations. Return-oriented programming attacks require the *predictable location* and re-use of instruction sequences in existing executable code. ROP attacks divert execution to these existing sequences in order to carry out operations (e.g. adding two registers) which

collectively implement the attack. Thus, the randomization of instruction locations defeats such attacks.

Instruction layout randomization (ILR) achieves its aim by using an execution model where each instruction has an associated explicit destination specified in a *fallthrough map*, allowing the full process instruction space (full address space of 32 bits for x86 architectures, 64 for newer architectures) to be randomized. The execution model is implemented with a process virtual machine (PVM) which interprets a set of *rewrite rules* to ensure that randomized execution control flow is followed. The authors report an average overhead of 13 percent.

ASLR and ILR do not seek to model or protect integrity, and they do not address attacks which do not rely on precise knowledge of target program instruction locations. However, both techniques serve to protect the integrity of the execution of programs in the presence of attacks which hijack and direct the target program control flow. These techniques (to different degrees) prevent attackers from reliably reaching instructions of their choice within existing programs resident in memory, thus interrupting control flow integrity violations. Ultimately, the randomization protections discussed here succeed by imposing a cost for an attacker's access to instruction location information that is normally visible via analysis of static program code.

In general, randomization is fruitfully applied to increasing the cost of access to sequences of application-dependent information. Since it typically aims to increase the disorder of sequence data, the process of randomization runs counter to the goal of ensuring the literal adherence of data to a reference.

Randomization techniques do not directly protect the static integrity of memory contents, yet the corruption of memory will commonly be included in the sequence of memory effects of attacks which are dependent on instruction location. The integrity of memory interactions composed by software executions are not directly protected either, yet attack sequences resulting in violations of the compliance of execution can be interrupted by barriers to information erected by randomization. Thus, the protection provided by randomization techniques such as ILR to integrity is not systematic or general, but increases the cost of a key step in common attacks upon several aspects of the correct operation of software.

## 3.8 Summary

The defensive techniques we have analyzed here provide some degree of protection to a particular manifestation of the integrity of computer systems. The limitations of these techniques have been elucidated with respect to the type of threats they address. Aside from a lack of generality, a critical underlying factor in the failure of most available protections is their inability to extend to both the contents and operation of software. Without the flexibility to protect both memory and executions in a unified manner, security schemes will remain vulnerable to attacks that target the unaddressed categories, or attacks which exploit multiple aspects of the system (perhaps sequentially) in order to achieve a larger aggregate violation of integrity. The inter-dependencies between the forms of computer system software integrity revealed in this work impose a requirement for the creation of a realistic general software integrity protection: the overall strategy must transfer beyond the scope of individual attacks, and must be based on a concise, unified model of software integrity which does not exclude the integrity of any type of memory or any type of memory usage during execution.

# CHAPTER 4

# TRUSTED GROUP KEY MANAGEMENT FOR CRITICAL INFRASTRUCTURE

National electric power grids are a challenging and crucial setting for the creation of cyber-physical systems with security in terms of availability, confidentiality, and integrity. In this chapter, an application of Trusted Computing (TC) integrity protection is discussed in which a group key management framework for critical infrastructures is constructed using the static integrity protection of TC along with the authentication service of Kerberos. For the full published work the reader is referred to [46].

Trusted Computing is an architecture for establishing trust in computer systems via the inclusion of robust trust anchors that host protected secure capabilities [39]. When implemented, the *Trusted Platforms* specified by Trusted Computing benefit from secure storage, secure and reliable integrity measurement, and the ability to attest to the integrity of the on-board software in a reliable manner. The Trusted Platform Module (TPM) is the core asset of TC, providing secure storage and protected cryptographic capabilities. These capabilities are the basis for the increased trust placed in Trusted Platforms. The measurement capabilities hosted by the mandatory Root of Trust for Measurement (RTM) engine are employed to statically measure the integrity of software on disk, by comparison with reference digests. Trusted Computing assets are deployed to computing resources, gaining the ability to

1. Make robust assumptions of trust concerning the software run by computer system infrastructure.
2. Prevent unintended system execution via the measurement capabilities standardized by TC. Computers can be shut down if measurements are not satisfactory.
3. Apply attestation and secure interoperability standards to prevent the propagation of damage from compromised machines in computer networks.
4. Derive domain-specific security advantages from their use of TC assets or structures.

In [46], Jenkins et al. apply the Trusted Computing architecture within a strategy to meet the stringent integrity and availability requirements of critical infrastructure devices. The context

of the application is the provision of secure multicast group key management to the devices, as demonstrated within the electrical power grid. Computer systems are assumed to host the Trusted Platform Module (TPM).

In addition, the Trusted Network Connect (TNC) [71] architecture is employed for secure interoperation of infrastructure devices. This allows network access decisions (e.g. connections) to be made based on robust integrity information provided by Trusted Computing components. Connection requests can be denied if the remote device is not reliably identified or has not reported proper integrity information originating from the secure storage within its TPM.

The strategy proposed in [46] is evaluated by the emulation of critical portions of the group key management functionality on a modern testbed using a profiler (see Section C within the appendix). The precise timing measurements for the required group key management cryptographic operations such as encryption, digest computation and TPM key seal/unseal comply with the demanding timing requirements of critical infrastructure scenarios such as the electrical power grid, where mandatory response times are potentially within 4ms.

The construction of a secure, scalable, efficient group key management protocol using Trusted Computing structures and extension of an existing authentication protocol in [46] together represent an application of static integrity protection strategies towards the group key management problem within critical infrastructures. The established network authentication capabilities of Kerberos and the robust trust anchors of Trusted Computing fulfill two major security requirements of the protocol: trustworthy identities and the secure use of cryptographic material. Within each infrastructure device, Trusted Computing is applied in a unified manner to the *static integrity* of the device, where it enforces the load-time operational software requirements of critical infrastructures: any deviations in the static software state on the platform result in an appropriate response (often shutdown).

Although robust protection of *run-time* device integrity is an open problem, communication-based vulnerabilities are addressed by the Trusted Network Connect inter-operability architecture, and proposals are made for the next steps in approaching the problem. The work of Jenkins et al. in [46] is detailed below.

In [46], the authors propose a novel trusted real-time group key management framework for critical infrastructures. The authors leverage the widely deployed Kerberos authentication service and the Trusted Platform Module (TPM) interface as building blocks.

In particular, the Key Distribution Center (KDC) of Kerberos authenticates the principals (users or components) of a cyber-physical system, allowing the establishment of authentication channels between the KDC and any principal. On this foundation, the KDC is extended with the ability to create multicast channels for efficient group session key distribution, supported by TPM storage of long-term keys.

To demonstrate that the proposed group key management protocol is robust in real-time its performance and sufficiency were analyzed by using workstations that emulate IEC 61850-90-5-compliant protocols for substation automation systems.

The framework is built on the Trusted Computing (TC) paradigm (in particular the TPM) that provides built-in and non-migratable trust, and the Kerberos network authentication service.

## 4.1 The Trusted Platform Module

The Trusted Computing Group (TCG) [73] describes an architecture in which *trusted engines*, called *roots of trust* (RoT), are used to establish trust in the expected behavior of system software.

In this group key management framework the RTS capability of the TPM is utilized (key *sealing*) to enhance the security of long-term keys. Figure 4.1 illustrates a code segment that uses TPM to seal data. For further details of Trusted Computing, the reader is referred to Chapter 3.

## 4.2 Kerberos

Kerberos 5 [55] is a single-sign-on authentication service for open (untrusted) networks, and is based on the Needham-Schroeder authentication protocol [54]. It serves networks exhibiting dynamic trust graphs with short lived authentication links. For further details on Kerberos the reader is referred to [55].

## 4.3 Multicast Group Authentication

Kerberos is a single-sign-on authentication service for client-server applications. It is not designed for multicast applications where a client wants authenticated multicast access to several

```
//Create context, connect to TPM
Tspi_Context_Create(&hContext);
Tspi_Context_Connect(hContext,NULL);//NULL: local, or TSS_UNICODE str for remote
//Get the TPM handle
tsr=Tspi_Context_GetTpmObject(hContext, &hTPM);
//Create objects.. (all objects bound to the context)
//Load SRK from TPM (create object insufficient)
tsr = Tspi_Context_LoadKeyByUUID(hContext, TSS_PS_TYPE_SYSTEM, srku, &hSRK);
tsr = Tspi_GetPolicyObject(hSRK, TSS_POLICY_USAGE, &srkpol);
tsr = Tspi_Policy_SetSecret(srkpol, TSS_SECRET_MODE_SHA1, 20,
tsr = Tspi_Context_CreateObject(hContext, TSS_OBJECT_TYPE_RSAKEY,
                TSS_KEY_TYPE_STORAGE |
                TSS_KEY_VOLATILE | TSS_KEY_SIZE_2048 |
                TSS_KEY_NO_AUTHORIZATION |
                TSS_KEY_NOT_MIGRATABLE,&hSealKey);
tsr = Tspi_Context_CreateObject(hContext,TSS_OBJECT_TYPE_POLICY,TSS_POLICY_USAGE,&hSealkPol);
tsr = Tspi_Policy_SetSecret(hSealkPol, TSS_SECRET_MODE_SHA1, 20, mylamesecret);
tsr = Tspi_Policy_AssignToObject(hSealkPol,hSealKey);
tsr = Tspi_Context_CreateObject(hContext,TSS_OBJECT_TYPE_ENCDATA,TSS_ENCDATA_SEAL,&hSealData);
tsr = Tspi_Context_GetDefaultPolicy(hContext, &hPolicy);
tsr = Tspi_Policy_SetSecret(hPolicy, TSS_SECRET_MODE_SHA1, 20, mylamesecret);
tsr = Tspi_Data_Seal(hSealData,hSRK/*hSealKey*/,64,rawData,hPCRs);
```

Figure 4.1: Code Segment to Seal Data to a TPM

principals, a common requirement with cyber-physical systems. Furthermore it does not address denial of service threats or insider attacks, which can incapacitate power grids and more generally critical infrastructures.

In this application Kerberos is extended to capture both requirements: efficient authenticated group multicast as well as security against denial of service threats and insider attacks. The extension is described in terms of the RFC 4120 [RFC 4120] specification. For security the TPM interface is used (Section 4.1). For the details of the protocol, see [46].

## 4.4 Trusted Group Authentication

To secure the Kerberos multicast authentication protocol a TPM architecture is used. It is assumed that the principals of the Kerberos system and the KDC Servers are TPM-compliant. This guarantees that the private keys of the principals in the Kerberos database are in shielded locations and only become available when needed by Kerberos. Furthermore, prior to establishing an authentication link, remote attestation will ensure that there are trust paths that link the KDC to the client and the principals (via shared secret keys). This will protect the system from external integrity and confidentiality threats, as well as insider threats that target the private long term keys of principals.

The Trusted Network Connect (TNC) [71] is a TC platform interoperability architecture for trusted access control that is based on the the integrity measurements stored in the TPM. It allows

for network access control based on trustworthy host integrity information. For this Kerberos application this means that, an authentication channel between a client and Server is established only if:

1. The identities of the client and server are trusted. For the first and second rounds of the Kerberos protocol, the Server is a trusted entity (either AS or the TGS). If the client is pre-authenticated, then its identity is confirmed via a secret shared key.

2. The client is allowed access to the Server. Authorization can be implemented either with Kerberos 5, or by using an appropriate Access Control System with credentials.

3. The identity of the client and server is authenticated. This is achieved by encrypting data with a shared secret key (either the secret key of the client, or a key generated by the KDC). This requires a RoT engine to be invoked on the TPM of both sides to release the required encryption key. The TPM will only release keys if the current configuration of the OS of both principals allows for this.

The Kerberos 5 specifications [55] also allow for a public key setting. For such settings public key authentication channels are used to establishing a symmetric key authorization service (in this case the KDC only knows the certified public keys of the clients). One of the advantages of such an approach is that Kerberos is not required to store long term secret information.

## 4.5 Reliable Group Authentication

In [46] several measures were taken to address the scalability issues of applying Kerberos authentication to the critical infrastructure group communication context, including redundant Kerberos key distribution centers, load-balancing, and the lightweight directory access protocol (LDAP).

The approach to secure the Kerberos multicast framework is based on the following requirements:

1. *Reliability.* The Kerberos network system has sufficient redundancy so that the probability of system failure caused by Nature is less than a small threshold $\varepsilon$.[1] This requires that we: $(i)$ specify the set of natural fault events $\mathcal{F}$, $(ii)$ estimate the probability $\varepsilon$ that such events occur, and $(iii)$ replicate system services sufficiently (and independently) to enable it to respond with probability $\geq 1 - \varepsilon$ to at least $r$ requests per unit time.

[1] For critical infrastructure protection, typically $\varepsilon = 2^{-30}$.

2. *Integrity and confidentiality.* The Kerberos system is TPM-compliant. This prevents integrity and confidentiality attacks.

3. *Availability.* We assume that the supported rate $r$ is sufficient to cope with high priority requests. This requires a bound on the number of high priority requests, particularly on $(i)$ the number of requests per KDC Server and $(ii)$ the number of clients per Server. To protect against DoS attacks caused by flooding bogus requests, an intranet is used and the high priority designation of a request is authenticated (*e.g.,* included in *padata*)—in particular, only high priority requests get forwarded when the traffic level is high.

The level of security provided by adopting these requirements is that afforded by the TPM interface (in particular how well it is implemented), provided that faults caused by Nature are restricted to the event space $\mathcal{F}$.

## 4.6 Trusted Substation Automation

To validate the group key management protocol and establish real-time efficiency, the protocol was integrated with an open source IEC 61850-90-5 emulation program recently released by SISCO [67] for trusted power system automation applications.

### 4.6.1 System Setup

The experimental software is based on the open source profiler for the IEC 61850-90-5 Technical Report released by SISCO [67], which was enhanced using the techniques in Section 3 from [46]. The modifications to the source code include porting the code to Linux, adding our group key management protocol, a trusted platform module, cryptographic support for encryption and authentication, and a time-keeping functionality for data generation.

A testbed consisting of seventeen local machines connected by a Cisco Catalyst 3560G series PoE 48 switch was used. All machines were running Ubuntu Linux 12.04 64 bit OS; eight had an Intel Xeon 5120 1.86 GHz x2 CPU with 2GB of memory and nine had an Intel Xeon E5506 2.13GHz x4 CPU and 6 GB of memory. A dedicated machine running the krb5 API acted as our Kerberos 5 release 1.10 server, which interfaced directly with the Kerberos API.

Experiments were carried out to measure timings for AES encryption/decryption, HMAC Authentication, Random Number Generation, and TPM sealing. Additionally, experiments were

completed to estimate the time required for initial group key registration, group session key distribution, and transmission of system event packets. The critical estimated total time required for transmission of the event packets was within the time bounds specified by the IEC standards studied.

## 4.7 Conclusion

In [46] Jenkins et al. proposed and developed a framework that enables secure multicast communication via efficient and scalable group key management. Leveraging the widely used Kerberos authentication service and TPM modules for secure storage, the system provides distinctive security advantages. The sufficiency of the proposed framework was demonstrated by *implementing* a fully integrated and functioning system based on a power utility automation profile emulator. Emulation results showed that the system can meet the stringent real-time requirements of the IEC 61850-90-5 Technical Report while providing security guarantees.

The proposed framework and implementation were a first step toward securing cyber-physical systems. While the experiments involved a relatively small number of nodes (due to practical limitations), the results can be readily generalized to much larger systems.

The work here has demonstrated the efficacy of static integrity protection via the application of Trusted Computing to the security of group communication within critical infrastructures. The application of Trusted Computing engines including the TPM provide robust measurement of the integrity of the programs run and the key information used. Further, the establishment of network connections is linked to integrity reporting requirements based on the available integrity information.

However, the application of Trusted Computing integrity protections are limited to the contents of memory, particularly the static integrity of program code that will be executed on platforms which must be trusted. The operation of program code is not similarly protected. As Chapters 6 and later will reveal, static integrity protection must be extended by the addition of techniques for the protection of the dynamic integrity of execution in order for unified software integrity to be realized. Beyond the content of programs, the operation of those programs must be trusted.

# CHAPTER 5

# REAL-TIME AND RUN-TIME TRUST FOR CRITICAL INFRASTRUCTURES

In this chapter an application of static integrity protection to real-time and run-time threats to critical infrastructure device is discussed. For the full published work the reader is referred to [45].

A strategy is advanced by Jenkins et al. in [45] for protecting critical infrastructure devices from real-time and run-time threats by combining the static integrity protection of the Trusted Computing architecture, access control structures and dynamic software integrity monitoring. Critical infrastructure devices must face threats of equipment failure and communication delay, but also intentional attack via malicious software. Strategies are necessary to address these threats, given the dire consequences of infrastructure failure to society with respect to life, cost, and stability.

In this work, the Trusted Platform Module (TPM) and Trusted Network Connect (TNC) standards of Trusted Computing (TC) provide robust static software integrity protection, secure storage and measurement-based interoperation, while access control techniques are applied to meet the stringent timing demands of real-time critical infrastructures. Secure interoperation mitigates one vector for run-time threats, since remote attackers may exploit vulnerabilities in software which makes use of network communication. The foundation of the benefits provided by TC to the critical infrastructures studied is a service for the measurement of the integrity of the static contents of programs which reside in memory.

Availability is a key requirement of critical infrastructures such as the electrical power grid, where the transmission of high-priority information must be guaranteed. The technique proposed here is to apply an attribute-based access control structure such that when congestion is present on network channels, high-priority data is given exclusive access to the channel by gateway devices. The access control is implemented by associating attributes for congestion levels with the traffic and channel. Given an assumed rated data rate for the infrastructure network, the use of the channels can be guaranteed for critical traffic in severe circumstances when secondary traffic must be dropped.

In order to address *generally* the problem of dynamic, run-time system integrity protection it is proposed that dynamic integrity monitoring shows promise, when combined with appropriate

constraints. The technique of monitoring running software for the purposes of performance measurements or optimizations is well-understood, and tools such as profilers exist for this purpose.

Finally, a profiler for a trusted electrical power grid substation automation system (SAS) was implemented in [45] (see Appendix C) to validate the strategies applied to protect the integrity of critical infrastructure devices. The trusted SAS employs Trusted Computing engines (i.e. the TPM) to ensure that the system program contents are uncorrupted and that unauthorized modifications result in appropriate security policy enforcement responses (even in the case of insider attacks). The cryptographic operations and key management routines necessary for securing infrastructure devices were timed using a testbed with modern machines. This emulation illustrates the feasibility of securing critical infrastructure systems against the studied threats. The work of Jenkins et al. in [45] is detailed below.

## 5.1 Introduction

Trusted Computing (TC) as defined by the Trusted Computing Group [73] is a technology that: (*i*) supports application platforms in securing distributed systems by using trust engines that attest to the integrity of the system, (*ii*) provides sealed storage and (*iii*) enforces the expected behavior of a system. This prevents the execution of untrusted (potentially malicious) software and also addresses insider threats which include the running of programs whose contents are in an unauthorized (uncertified) state. However, TC architectures offer only static (load-time) protection to the contents of programs, so one still has to address the possibility of run-time (execution) attacks on the operation of software.

It is vitally important for the security of a critical infrastructure that controllers can access system state information in real-time: correct messages delivered at the wrong time may lead to erroneous responses and system failure. Communication must therefore be subject to real-time constraints with availability guaranteed within strict time bounds, a fact which renders most commonly adopted security paradigms for cyber systems inapplicable. The three primary goals for cyber security are confidentiality, integrity and availability. However for critical infrastructure systems, real-time availability with integrity is a primary goal, with privacy often only a secondary goal. This leads to new paradigms for securing critical infrastructures in which robustness in real-

time is a primary goal. This has an impact on implementation aspects, since real-time behavior cannot be captured by traditional security paradigms.

It is impossible in general to provide fully effective protection to critical infrastructure systems with currently available operating systems, at least with regard to real-time availability and run-time integrity. What is needed is to identify the key vulnerabilities of critical infrastructure systems and then use a TC architecture that prevents the exploitation of such vulnerabilities.

In this work a TC architecture is combined with a real-time access control infrastructure to get protection against both real-time availability and malware threats. Ultimately, to thwart such attacks in the general case, network devices of TC-compliant systems will have to: ($i$) operate with a limited, enumerated set of functions and, ($ii$) employ a form of dynamic integrity monitoring that is time-bound by the real-time availability constraints of the system.

## 5.2 Trusted Computing for Critical Infrastructure Protection

The Trusted Computing Group (TCG) [73] has published specifications for architectures and interfaces for several computing implementations. Platforms based on these specifications are expected to meet functional and reliability requirements of computer systems that allow increased assurance of trust. As such, they are well suited to support and protect critical infrastructures. The Trusted Platform Module [70] (TPM) and the Trusted Network Connect (TNC) [71] are two TCG components that address the security threats of critical infrastructures. The reader is referred to Chapter 3 for full details of Trusted Computing.

## 5.3 A Threat Model and Security Framework

By securely housing integrity measurements of software, the TPM facilitates the prevention of compromised components from executing. As a result, if we exclude run-time threats, malicious (Byzantine) threats are reduced to Denial of Service (DoS) threats. This is a radical departure from the traditional threat model for computer systems because, as opposed to Byzantine faults, DoS faults are overt: they are self-revealing and hence detectable.

To protect a critical infrastructure from such overt threats one may use reliability mechanisms such as replication and redundancy. It is well-known that tolerating $f$ Byzantine faults in a

distributed system requires $(2f + 1)$ redundancy (with reliable broadcast) [32]. For a DoS threat model the faults are overt. Tolerating $f$ DoS faults requires only $(f + 1)$ redundancy.

There are two kinds of faults that may affect a TC-compliant critical infrastructure: natural (which includes accidents) and adversarial (intentional/malicious/insider) DoS faults (excluding run-time attacks). Natural faults can be predicted, in the sense that an upper bound on the probability of such events can be estimated. Redundancy can then be used to reduce this probability to below an acceptable threshold. Malicious DoS faults cannot be predicted.

However, such DoS faults are overt and because of the TPM and TNC integrity verification, must be physical (*e.g.*, involve tampering with the TPM chip). So, there is a cost involved, and one way to thwart them is to make the cost high enough to prevent them. There are several security models that use economics and risk analysis based on replication and redundancy [50] that are appropriate for threat models with overt faults. These assume a bound on adversarial resources and an architecture with sufficient redundancy to make DoS attacks prohibitively expensive. In this approach for critical infrastructure protection an architecture with sufficient redundancy is assumed such that: $(i)$ the probability that the system will fail due to a natural fault is negligible (*e.g.,* less than $2^{-20}$) and, $(ii)$ the cost of a successful DoS attack by compromising system components is prohibitive for a resource bounded adversary. The security framework for a critical infrastructure $\mathcal{C}$ consists of:

1. A real-time model that captures its functionality including a faults distribution $\mathcal{F}$ on components.
2. A set $\mathcal{S}$ of specifications, policies, constraints and security requirements that identifies the *vulnerabilities* $\mathcal{V}$ that need to be protected.
3. An $\mathcal{S}$-profiler that emulates the behavior of $\mathcal{C}$ in real-time subject to the specifications $\mathcal{S}$. The profiler is defined by a software program that runs in real-time.
4. A proof that the $\mathcal{S}$-profiler adheres to the security requirements specified by $\mathcal{S}$ in real-time in the presence of faults/disasters, accidents and malicious behavior.

**Remark.** This framework captures current state-of-the-art for integrity and confidentiality and addresses availability in real-time. The $\mathcal{S}$-profiler is used to measure the actual time required to protect system resources. Note that traditional models based on formal methods or security models do not capture real-time availability.

### 5.3.1 A Security Framework for Critical Infrastructures

Let $\mathcal{C}$ be a real-time $\mathcal{S}$-compliant critical infrastructure with faults distribution $\mathcal{F}$, specifications $\mathcal{S}$ and vulnerabilities $\mathcal{V}$. We model $\mathcal{C}$ by a finite hybrid real-time automaton with faults [9]: $\mathcal{C} = (\tau, A, Q, q_0, D, \mathcal{F})$, with $\tau : t_1, t_2 \ldots$ a time schedule, $A$ a finite set of actions that includes a special symbol "$\perp$", $Q \neq \emptyset$ a finite set of states that is partitioned into *safe* states $Q_s$, *critical* states $Q_c$, and *terminal* states $Q_t$, $q_0 \in Q_s$ an initial state, and $D \subset Q \times Q \times A$ a transition function that is time triggered. $D$ is deterministic when $a \in A \backslash \{\perp\}$ and probabilistic when $a = \perp$; in this case, the posteriori state is selected by Nature according to $\mathcal{F}$.

The parties of a critical infrastructure are those specified by $\mathcal{S}$: *e.g.,* intelligent electronic devices, operators, the adversary and Nature (the Environment).

Nature controls the temporal and location aspects of all events and schedules state transitions in a timely manner according to $\tau$, using the distribution $\mathcal{F}$ to select from among her strategies for component failure, and resolves concurrency issues by linking events to their actual start time. For the application in Section 5.6 the transition frequency $\delta = t_{i+1} - t_i$ is $4\,ms$—the latency specified by the Technical Report IEC 61850-90-5. It is important that the protection mechanisms of a critical infrastructure adhere to the time-frame $\tau$: command/control information that arrives late may result in the system transitioning to a critical/terminal state.

A semantic security threat model is extended to capture real-time events. The threat model restricts the adversary to exploiting the system vulnerabilities $\mathcal{V}$ specified in $\mathcal{S}$, in particular: those identified by the policies of the system, vulnerability assessments, and gray-box penetration testing [30]. The vulnerabilities involve the components of the system such as the control systems, the embedded systems and the communication channels.

### 5.3.2 The Good, the Bad and the Ugly

Since the inclusion of TC results in a threat model with both significant benefits and unique limitations, an analysis of the effects of TC on the model is presented here.

**The Good.** The TPM is an interface that protects system components from behaving in an unexpected way during a malicious attack. This is achieved by having an integrity protected boot process and using protected capabilities to access shielded locations for: ($a$) PCRs to verify integrity and, ($b$) cryptographic keys for integrity/confidentiality services. In particular, prior to

the execution of any authorized program an integrity check of its state (against a stored PCR configuration) is required. Consequently, if the program is compromised it will not be executed.

**The Bad.** The TPM allows only authorized software programs to execute. Therefore the integrity of system software (which includes the OS) is a fundamental requirement in order to ensure trust in the computing infrastructure. The system software must be well designed, and not have security holes, backdoors or other vulnerabilities that could be exploited by an adversary.

A vulnerability in the system software may allow the adversary to bypass the protection offered by the TPM. There are many reasons why software programs for critical infrastructures may have a faulty design. A major reason is the complexity and architectural constraints of the execution environment (the OS and CPU hardware). Additional reasons are poor software development practices and lack of software security in applications. In conventional cyber systems, a reactive approach is typically used in which program flaws are addressed with patches, which is not suitable for critical infrastructures.

**The Ugly.** The TPM provides integrity verification of the static contents of software, and does not fully address run-time attacks. By exploiting a vulnerability in system software an adversary may be able to change the execution flow of a program, *e.g.,* by using a buffer overflow attack. There are several run-time attacks [2] that use metamorphic malware such as the self-camouflaging Frankenstein [53] or more generally, return oriented programming (ROP) [63]. With such attacks the adversary must be able to control the flow of execution on the stack, but there are ways to prevent this [29]. However, this does not exclude the possibility of other attacks that exploit system vulnerabilities to execute run-time attacks.

## 5.4 Real-Time Availability Threats for Critical Infrastructures

Real-time availability threats to critical infrastructures exploit the time required to deliver system resources. For example, in an electrical grid controllers must have access to state transition information in real-time in order to avoid cascading failures. For substation automation systems, synchrophasor data streams are sent to controllers via local networks. Protecting such streams from threats involving deletion (availability), corruption (integrity) and privacy is crucial.

Typically the synchrophasor reporting latency should be less than $10\,ms$ (to prevent cascading faults [43]). Real-time availability faults occur if the time taken to deliver such information is

longer than the specified latency. Real-time faults may be natural—*e.g.* critical state information may get dropped or not arrive in time to be processed, or malicious—*e.g.* an insider may corrupt synchrophasor data. Note that Quality of Service (QoS) is not a protection mechanism for critical infrastructures: an average delay less than the latency requirement may still result in real-time failure if the actual delay exceeds the requirement.

To address real-time access control, a real-time attribute based access control mechanism was proposed which extends the Integrated Services and Differentiated Services architectures for network bandwidth allocation [6, 66]. By employing redundant capacity to deliver traffic at a specified rate and using a queuing algorithm based on real-time traffic attributes, high priority traffic is guaranteed delivery under faults. For the details of the access control measures, see [45].

## 5.5 Run-Time Attacks

Run-time attacks involve the imposition of unintended behavior on a software target that is executing. In a return oriented programming (ROP) attack [63] there is no need to inject new malicious code: code in local libraries may be reused. ROP overwrites the stack with addresses that point to existing code in a library (*e.g.*, the standard C library libc). Instead of calling a function, it sets return addresses on the stack to possibly unaligned instruction sequences that typically end with a return instruction. After executing the instructions, the return takes the next address from the stack where execution continues, and increments the stack pointer. The stack pointer in effect determines the program control flow and acts as an instruction pointer. For more details the reader is referred to [63, 29].

Attacks of this kind use instruction sequences (e.g. from libc) instead of whole functions. Clearly the adversary must be able to control the stack flow for such an attack, and there are ways to prevent this [29]. However, there may be run-time attacks that exploit other vulnerabilities using a minimal footprint. There is therefore a need to guarantee that during run-time the execution of a program cannot deviate from the prescribed flow path.

### 5.5.1 Run-Time Threats for TC Systems

One critical characteristic of the TC model is its reliance on digests of static execution targets (software) as the key measurement of the platform's integrity. In order to materially impact the

integrity of the trusted platform, the adversary must attack the platform software without requiring the presence of malicious software that would be detected with TC structures. ROP attacks [63] do not require additional malicious code to be injected onto the platform, but assume that the attacker has already diverted control of execution. The most salient way to achieve this initial diversion under our constraints is with a network-based attack, for which there are many examples in the literature [3]. If TPM hosts are successfully exploited by run-time, network-based attacks, it will be possible to alter their behavior in a manner that is not detected by the TC methodologies. However, the TNC architecture enables network endpoints to decide whether to accept connections using exchanged integrity information. Consequently, devices implementing TNC structures and protocols have the capability to restrict network associations to allow only trusted remote entities, thereby limiting the adversary's ability to launch network-based attacks.

## 5.6 Secure Communication for the Electricity Grid

To demonstrate the effectiveness of this approach it is applied to protect the communication of an IEC61850-90-5-compliant substation automation system (SAS) of an electricity grid against real-time availability and run-time integrity threats. IEC 61850 [42] is a Technical Report of the International Electrotechnical Commission (IEC) that offers advanced object oriented semantics for information exchange in power system automation applications, supervisory control and data acquisition (SCADA), system protection, substation automation and distribution automation. IEC 61850-90-5 [43] extends the previous standard by specifying the use of the IP transport protocol with data encapsulated in IP packets and formulated so that they can be distributed in Wide Area Network (WAN) environments. This allows for low cost Wide Area Monitoring, Protection and Control. Both IEC 61850 and its extension do not address security issues.

In this application it is shown how to achieve robustness by using a TC architecture with TPMs. This involves integrating the SAS infrastructure with: (*i*) a TPM interface (*ii*) a Kerberos multicast authentication service [8], (*iii*) a real-time attribute-based access control system. and (*iv*) cryptographic services for AES and HMAC (SHA2). Protection against run-time attacks is provided if we assume that all trusted software is well designed and has no flaws or other vulnerabilities (Section 5.3.2). This assumption is reasonable for critical infrastructures with minimal system software.

### 5.6.1 A Trusted IEC 61850-90-5 Profiler

SISCO recently released an open source software package [67] which includes a sample profiler that emulates IEC 61850-90-5 systems. This does not support any security services. This profiler is the basis for the demonstration of real-time security for this application.

### 5.6.2 The Testbed

A testbed with 17 workstations connected via a Cisco Catalyst 3560G series PoE 48 switch and other switches was established. The machines were TPM enabled and ran on both Ubuntu Linux and Windows: 8 an Intel Xeon 5120 1.86 GHz x2 CPU with 2GB memory and 9 had an Intel Xeon E5506 2.13GHz x4 CPU and 6 GB memory. A dedicated machine running the krb5 API acted as our Kerberos 5 release 1.10 server, which interfaced directly with the Kerberos API.

### 5.6.3 Trusted Substation Automation

A substation automation system (SAS) consists of multiple substations connected via an intranet network. To establish a trusted SAS, each component in this zone should be TC-compliant. For trusted interoperability, the TNC architecture is used (Section 5.2). This addresses passive attacks, but also active attacks including insider attacks. The TPM interface prevents authorized components that get compromised from behaving maliciously, in particular, from contributing to a DoS. Furthermore, TNC network access control prevents external DoS attacks. By using a real-time access control service to manage data feeds (Section 5.4), real-time availability is guaranteed for high priority packets provided the SAS network has sufficient redundancy to be resilient when only such packets are sent.

A set of experiments were conducted with the test environment for authentication (HMAC SHA), encryption (AES), random number generation, and TPM seal/unseal. The results demonstrated that the security services are able to operate within the strict timing and static software integrity requirements of real critical infrastructures. For the full details of results see [45].

## 5.7 Conclusion

The addition of TC capabilities to critical infrastructure devices addresses threats to static software (load-time) on Trusted Platforms. Real-time availability can be provided with appropriate

access control techniques and redundancy. If run-time threats are excluded, TC systems enable a conversion of general (Byzantine) threats to Denial of Service attacks.

An attack on software execution (run-time) can not be as readily revealed by static integrity measurements. Thus, separate techniques are needed to address such threats. To mitigate run-time threats for critical infrastructures, any successful approach will have to both limit the openings for exploitation on platform software and employ methods to detect run-time compromise.

To achieve a smaller attack surface, devices will need to abide by constraints on functionality and resource usage, operating with a structured, well-defined, enumerated set of duties. Clearly the presence of software flaws is related to the complexity and size of software.

There is substantial existing work on techniques for dynamically detecting software faults (dynamic integrity monitoring and taint analysis) [15, 18, 62]. Although most existing proposals carry significant computational overhead, critical infrastructure systems can justifiably be expected to bear the requisite computational resources. Dynamic integrity monitoring is a technique that shows promise for the task of protecting infrastructure systems from run-time failures that can result in catastrophic consequences. However, the time required to address run-time threats may increase the likelihood of a real-time availability attack.

This work has demonstrated the application of the static integrity engines of Trusted Computing [73] and state of the art access control techniques to the building of critical infrastructures with real-time availability and integrity. However, fully protecting the integrity of software means employing measures to ensure the compliance of software execution with the intent of program developers in addition to the soundness of program code. In Chapter 6, static integrity protection is extended to address dynamic integrity, the integrity of the execution of software. The introduction of dynamic integrity protection constitutes comprehensive coverage of software integrity. Finally, the subsequent creation of a systematic protection framework for the application of these protections completes the formation of a unified, general software integrity protection.

# CHAPTER 6

# DYNAMIC SOFTWARE INTEGRITY

## 6.1 Applying Trusted Computing to Software Integrity Protection

Trusted Computing (TC) is an architecture that enables the prevention of system components from behaving in an unauthorized manner during a malicious attack. This is achieved by an integrity-protected boot process and the exclusive use of protected capabilities to access shielded locations for: ($a$) Platform Configuration Registers (PCRs) holding integrity measurements and ($b$) cryptographic keys, for integrity/confidentiality services.

In particular, prior to the execution of any authorized program, an integrity measurement of the program static state (to be compared against a stored PCR configuration) can be required. If the program is compromised in a manner reflected by the measurement methods in place, it will not be executed by the system.

### 6.1.1 Limitations of Static Protections

Although Trusted Computing enables confidence in the integrity of computer systems via the use of trust anchors, the TC architecture is not a complete software integrity protection. Some obstacles to the successful application of TC originate outside of the architecture itself, while others are a product of both the limitations of available methodologies and the employment of those methods within TC specifications. Some limitations of Trusted Computing for integrity purposes are presented here.

The Trusted Platform Module (TPM) is the core (typically hardware) Trusted Computing technology that resides within computer systems as the protected base of capabilities described in the specifications [70]. The measurement capabilities housed by the TPM allow only authorized software programs to execute on its host *Trusted Platform*. Therefore, system software (including the operating system) must meet a high standard in order to ensure trust in the computing infrastructure. This is in fact a requirement of the integrity of the *process of software production*, which is beyond the scope of this work. The system software must be well designed and free from security holes, backdoors or other vulnerabilities that could be exploited by an adversary. An authorized

program *with flaws* (which are in truth an inherent part of the production of software), though its measured integrity state in memory matches a reference, may allow an exploitation that opens the way for generalized integrity violations and system compromise.

A vulnerability in the operating system software may allow the adversary to bypass the protection offered by the TPM. Software programs are inevitably flawed, in part due to the complexity and architectural constraints of the execution environment (the operating system and CPU). Poor software development practices and absent attention to security exacerbate the issue. Conventionally, a dynamic approach is typically used in which program flaws are addressed with patches.

Ultimately, the most critical limitation of the application of TC with respect to integrity protection is that the TC architecture provides integrity verification of static software in memory, not executions of software.

Protecting the contents of software is insufficient for general software integrity protection because there is no implicative relationship between the static integrity of program contents in memory and the resulting program execution behavior: it is possible to interact with the protected program via input or call such that the consequences of the program execution are counter to what was intended by the presumed non-malicious author. The details of such interactions and the conclusions drawn regarding the requirements for the construction of a systematic software integrity protection remain to be explored in this Chapter. Selected examples are briefly mentioned here.

By exploiting a run-time vulnerability in the operating system an adversary may be able to change the execution flow of a program, e.g., by using a buffer overflow attack. There are several run-time attacks [2] that use metamorphic malware such as the self-camouflaging Frankenstein [53] or more generally, return oriented programming (ROP) [63]. With such attacks the adversary must be able to control the flow of execution on the stack, and there are ways to prevent this [29]. However, this does not exclude the possibility of other attacks that exploit system vulnerabilities to execute run-time attacks.

The capabilities of Trusted Computing components such as the TPM enable secure storage, and reliable measurement and reporting of integrity. This means that TC is well suited to the protection of static integrity via reference comparison. In addition, a static integrity protection based on TC benefits from the standardization and interoperability that is inherent in the Trusted

Computing architecture. However, the reliance of TC protections on static integrity measurements means that additional measures are needed to address the executions of software.

## 6.2 Addressing Dynamic Integrity

### 6.2.1 Threat Model

The adversary envisioned in this work is able to adversely influence the execution of software within computer systems via providing input and inducing call interactions. The software creator's intent is assumed to be non-malicious. The protected software programs, although certified and trusted, will necessarily contain flaws which may be exploited. The typical vector of attack for this adversary is the exercise of his influence on the inputs to protected programs. This is not to exclude other undiscovered methods of achieving the objective stated above. This threat model reflects the real-world threat posed by attackers of conventional software who target software vulnerabilities by exploiting inferences about internal program behavior and use of input (e.g. data).

### 6.2.2 Extending Static Integrity

In order to begin the task of detailing the form of the extension to static integrity which is necessary for unified software integrity, we begin with a set of starting conditions under which new protections will act.

Presuppose a set of programs to undergo integrity protection.

1. Programs are certified to have a static memory state intended by the author(s) and provided with a literal static integrity reference for direct comparison. Programs contain design flaws.
2. Static integrity of memory-occupying objects for which reference measurements can be produced and threat of unauthorized modification exists is enforced by integrity checks with respect to reference, at a frequency sufficient to ensure integrity prior to use.

Under the stated conditions, the memory effects and action of program code *instructions* is locked. However, the compliance of memory interactions occurring due to software with program creator intent is not guaranteed. An adversary is capable of influencing execution of a protected program via software means (as opposed to attacking the software environment etc.) by causing the execution of protected code memory interactions using input he can influence.

The exercise of such a capability to cause memory effects which are counter to the intent of the program creator constitutes exploitation and the openness of the protected program to such exploitation can be considered a flaw. A key example of this action is the inducement of procedure calls within protected code using parameters crafted by the adversary.

The integrity of data which is provided to a protected program cannot be measured statically due to the violation of the basic requirements for the construction of static integrity measurements (see Section 6.5.1). Firstly, the reference for a static measurement of the provided data lies with the potentially adversarial data source. Additionally, the input data is assumed not exposed to unauthorized modification during the interval between its reception and use.

Integrity-relevant memory interactions are expressed in sequences of program instructions and employed to implement the intent of the program creator, which is represented by a set of *effects on memory.*

*The capability of adversaries to direct sets of memory effects along with the program creator implies that the integrity of the operation of programs with respect to the creator's intent can be violated despite the intactness of program code contents, allowing further integrity and security violations.*

Thus, in order to provide unified integrity to software such that its contents and action is in compliance with intent, the dynamic integrity of memory interactions influenced by adversaries must be protected, in addition to the contents of memory locations occupied by data and programs (static integrity protection).

## 6.3 Foundations of a Strategy for Unified Integrity

With an appreciation for the threats to computer system integrity and the limitations of static integrity protections, it now remains to construct a strategy which can provide robust, unified integrity to computer system software. First however, a case must be made for the feasibility of such a strategy as demonstrated by available building block techniques and their amenability to the task. The prospects for the creation of a unified software integrity protection are confirmed by the existence of several well-understood techniques serving as foundations that can be built upon.

When the various manifestations of system integrity are considered individually, existing techniques provide some level of protection. The static memory of the system can be protected with

scans for malicious modification. This is to include checking of memory contents with respect to reference *during run-time*. When applied to static memory integrity protection, the Trusted Computing architecture [73] enables reference-based, trustworthy measurement and reporting of integrity information.

Conventional security software (anti-malware) provides a limited form of integrity protection to execution, in that it may filter the use of undesirable inputs. For example, security suites may proactively block the user from opening or accessing a remote content resource based on an a priori assessment of its threat. We have seen in Chapter 3 that there exist targeted defenses for specific attacks on the integrity of software execution.

Although the problem of dynamic integrity protection of executions is challenging and open to much research, there do exist methods of measuring certain properties of dynamic execution. Software *profilers* are able to measure aspects of run-time operation that are relevant to performance or optimization, such as counting important procedure, calls, timing measurements or tracing the results of branch operations. There are numerous tools for monitoring targets from single programs to full operating systems. Implementations of software profiling typically rely on program instrumentation [77]. Further, the measurements involved can be taken with reasonable computational overhead [68]. Provided that the profiling principle of minimal interference with program operation is observed, software instrumentation is an unobtrusive and efficient method of profiling execution of programs [38, 68]. The instrumentation of source code in particular allows access to symbolic, syntactic, and contextual information during instrumentation which is typically not available in the binary stage of software.

Whether or not program instrumentation has the required access and capability to measure the dynamic integrity of software (which is investigated in this work), it is a basis for higher level analysis of software integrity that is of interest to broad categories of application.

Software programs are defined by their contents in program code residing in storage, as well as the interactions between memory areas their instructions impose during execution (which are to a limited degree visible in static source code). Therefore, the analysis of the integrity of software must be unified to capture the compliance of both aspects of software programs with references. In order to provide unified software integrity, it is necessary to specify a unified, coherent definition and model of such integrity. Having done so, the task moves to addressing the multi-faceted

problem of integrity protection with an appropriately general strategy. Our goal in such a strategy is to *systematize* software integrity protection, defending the integrity of the contents and action of software natively rather than defending against particular attacks.

Trusted Computing as an architecture has been discussed as a critical component of the proposed integrity protection strategy presented here, because it provides robust protection against *static* (sometimes referred to as "load-time") threats and mitigates certain dynamic threats related to interoperation. In this work, it remains to detail the rest of the components of a software integrity protection strategy, in particular how to address the protection of dynamic software integrity, which is a major research challenge. It is the contention of this work that a form of remote dynamic integrity protection (involving e.g. monitoring) is the remaining needed component of a complete software integrity protection strategy.

Control flow integrity and taint tracking techniques have been discussed in detail in Chapter 3. Both techniques have demonstrated effectiveness against attacks on control flow and propagation of untrusted data which leads to further attack. For a unified software integrity protection, both control flow integrity and taint tracking earn unique positions as foundations by applying their protections to interactions between memory objects which are characteristic of sound execution. Taint tracking can be carried out with reasonable performance impact, yet fine-grained (full) control flow integrity protection via software may cause a more substantial impact depending on the application. The fact that control flow integrity and taint tracking techniques address the integrity of the fundamental manner of interaction of memory-occupying objects distinguishes them from static techniques based on analysis of values, and identifies a new category of software behavior which can be threatened. As a result, both techniques must factor into the construction of a general software integrity protection.

## 6.4 A Unified Software Integrity Model

In order to model static and dynamic computer system software integrity, we define a set of elements as follows:

1. A set of object-associated finite memory regions $M$ with some finite size, $k$.
2. an interaction $I$ which is defined as a tuple $\{M_i, M_j, op\}$.

$M_i$ and $M_j$ $(i, j \leq k)$ are memory locations within $M$ and *op* is an abstract or concrete memory-altering or using operation using $M_i$ and $M_j$ as parameters in order to implement program logic. The interaction $I$ is sufficiently general to construct arbitrarily complex interactions between sets of memory locations. *op* may result in modification to the contents of $M_i$ and $M_j$, and therefore to the static integrity of these two memory objects. The use (e.g. read) of $M_i$ and $M_j$ in *op* may result in changes in the dynamic integrity of programs implementing the interaction $I$. Note that even a non-memory altering interaction such as *read* can impact on the integrity of executions of software because as will be detailed, the use of data may alter the compliance of the memory interactions which compose execution with creator intent.

It is necessary to identify a small set of key interactions that are critical for the modeling of integrity. A *read* interaction (*op* is read) describes the reading of the contents of a memory location in order to make those contents available for use (no use is implied, though it may of course occur). The *write* (*op* is write) interaction alters the contents of a memory location, placing certain values into the location. Finally, an abstract *use* interaction describes a set of read and write interactions with the property that a desired relation on a set of memory locations holds after the use.

Note that a *use* does not imply intent. Uses as described here model the active transitions in memory state which are effects of programs on memory. Programs are rarely expressed by developers in a manner such that memory effects are explicitly defined and intended. Thus, program logic is implemented by the direction (through a translation process) of memory state changes such that a certain relation or condition is satisfied with respect to a set of memory locations. In fact, a use may be directed by an attacker, the program author, or an environment program author. Uses directed by distinct entities can co-occur in time and memory space, during the execution of one program.

Static integrity of data undergoes direct change when the contents of a memory area are changed by an interaction (a write). The integrity reference for such an interaction is a literal value (or an approximation for efficiency purposes) which can be used for direct comparison to the data whose integrity is being measured. Violations of static integrity occur upon the execution of an interaction after which the contents in a memory location, say $M_i$, no longer matches a literal reference value.

Dynamic integrity of programs undergoes change when an interaction (or a set of interactions) occurs in which the reading of memory or writing to memory alters the compliance of execution

with the intent of the program creator. The compliance is with respect to a dynamic integrity reference such as a set of conditions on the data supplied to the interaction, or a control flow graph, etc. Each interaction between memory areas is a point of departure or adherence with intent by virtue of the transfer of compliance requirements from a subset of the memory inputs to the affected subset.

Dynamic integrity violations occur when as a result of the read, write or use interaction, the intent reflected in reference is deviated from, such as when a condition on two of the memory values in an interaction does not hold. Conditions on memory locations that are useful for dynamic integrity references are based on integrity primitives such as length, structure, presence/parity, linkage, and sequence .

As has been mentioned previously, flaws are a persistent and inevitable characteristic of the creation of software programs. For integrity purposes, flaws permit interactions in which memory state changes occur which deviate from the intent of the implementing program creator, which is assumed to be non-malicious. The adversary has the capability to exercise such interactions by providing input or influencing the interactions imposed on memory objects. In such a threat model, memory region state changes resulting from interactions must be decomposed. Creator-intended state changes and conditions define the envelope of proper program operation.

The inherent imprecision of the transformation of intent to executable program code ultimately expressed as explicit memory interactions, an "intent to effect" gap, along with deficiencies in the creation of programs (there may be additional sources of variation) mean that memory state changes and memory interactions that are *not explicitly intended by the creator* can co-occur with the execution of instructions if a triggering set of interactions are imposed. Some of these effects may violate integrity.

In addition, all memory state changes which are derived from program execution are *directed* by either the program creator or an entity able to influence execution: they may be directed by the interactions expressed by the creator, or by for example an adversary arranging for a procedure call in which the memory interactions imposed have dangerous effect. The combination of influences of both entities directs the entire set of memory state changes which ultimately occur and which exhibit the integrity of executions.

### 6.4.1 Protecting the Integrity of Interactions

*For the purposes of dynamic integrity, the execution of instructions is condensed to sequences of memory interactions which produce a set of effects in memory that fulfill a programmatic goal. Such sequences direct alterations and uses of the contents of memory regions in order to produce a set of stored results in designated memory locations which satisfy a relation with the input provided.*

The set of memory interactions directed by an instruction, call or program will generally not be explicitly specified by the program creator. There is an inherent intent-to-interaction or intent-to-effect gap between execution-bound integrity-relevant interactions of memory and the intent expressed by program creators due to the software-creation process (excluding atypical cases where software instructions are explicitly arranged by developers). These gaps are exacerbated by design errors in software production. However, note that the intent gaps mentioned can be exploited to produce subsets of creator-unintended memory interactions due to **insufficiently specific expression** of the intended memory interactions, while errors involve the **incorrect expression** of the intended behavior.

In the presence of an attacker or flaw, the directed memory interactions may not respect a requirement for compliance with intent across any of several dimensions or *primitives* of integrity such as length, structure, parity, etc. which are inherent in the use of memory locations.

Clearly, concrete instructions such as ‘add r1,r2,r3‘ impose interactions between memory areas (here by placing the sum of the value in two registers into a destination register). Procedure calls direct a set of memory interactions implemented via read and write memory interactions upon the data provided to the call, as well as data accessible within the scope of the procedure called.

*The mechanism by which call, control flow and taint memory interactions reveal dynamic integrity is the transfer of requirements for compliance with intent. This transfer of requirements occurs along dimensions defined by the primitive integrity characteristic of memory objects including length, parity, structure, value.*

The transfer of requirements exhibits integrity as follows: given the current static state of the interacting memory areas, a subset of the memory areas has an additional requirement imposed upon its properties (integrity-relevant properties mentioned above) in order for author intent compliance to be maintained by the effects of the interaction. For example, given that control flow has arrived at address $a$, sequence and linkage requirements are imposed on the succeeding control flow

destination in order for the intent reflected in a program control flow graph to be maintained: it must obey the destination prescribed by the control flow graph for $a$.

An example of integrity requirement transfer for procedure calls is found in a familiar memory copying function which copies contents from a source to a destination memory location. The continuance of compliance with intent demands that a requirement on the length property is imposed upon the destination memory region. Violation of this requirement includes executions in which memory is written past the boundary of the destination memory region, a memory effect which is not included in any legitimate non-malicious program intent expression, and which is a violation of fundamental requirements for proper operation of program execution environments.

Taint exhibits interaction in its interaction of presence and marking (marked as untrusted) properties across data-holding memory areas, while control flow exhibits sequence and linkage interactions between instructions. One distinction of taint is that the spread of untrusted data is typically an *effect* of (or and indicator of) violations of the remaining forms of dynamic integrity rather than an attack vector. Thus, there is a directed, enabling link between call and control flow integrity violations to taint.

The call interaction presents a stable interface to callers which allows the inducement of *author-directed* abstract or concrete memory interactions as part of the operation of the call program code, yet admits the influence of calling entities via input and the resultant primitive memory interactions. Thus, procedure calls are a larger scale, lower resolution application of the interaction of memory areas which contributes to the intent gaps discussed earlier. Requirement transfer with procedure calls is across the dimensions of integrity such as length, presence/parity, sequence, or structure.

The call-based set of memory interactions collectively directed by the author and non-author parties provides an opportunity for the direction of memory effects whose *combination* violates the transfer: the requirements for compliance with intent differ between the author-directed and non-author-directed sets, and their combination imposes a new set of requirements unsatisfied.

Consequences of adversarial influence on procedure calls with respect to the integrity model in use are:

1. The call interaction and input are the only ways under the stated conditions for adversaries to direct memory interactions along with those intended by the program author.

2. Under the conditions given, violations of dynamic integrity via software can only occur when a combination of memory effects is directed which counters the program author intent. Further, security violations can occur only via the direction of dynamic integrity violations.

Thus, under the model the protection of dynamic software integrity requires the following measures:

1. Create a process which produces from a memory data input integrity information primitives about its integrity-relevant properties (e.g. length, presence, structure)
2. Create dynamic integrity conditions which must hold between the primitives of memory data inputs to memory interactions
3. Enforce adherence to dynamic integrity conditions by software. This is equivalent to enforcing the transfer of requirements for compliance with intent.
4. Apply appropriate responses to violations

By combining the software integrity model detailed here with observations on capturing the intent of executing software programs, it is possible to show that the integrity of executing software can be measured and evaluated by the application of conditions on numerical memory-based properties of memory-bound objects caused to interact by program instructions. The steps leading to such a conclusion are listed below.

1. Let $M$ be a set of memory regions $m_x$ for use by programs.
2. An interaction $I = (m_a, m_b, op)$ is an operation using and altering memory regions within $M$.
3. Memory interactions alter memory regions, causing *effects*. For software integrity protection, the act of direction of memory interactions is the exclusive studied method of producing memory effects for the implementation of programmatic goals.
4. A developer creates source code program $p$, from which the executable program $P$ is produced. Let executing program $P = I = I_i, I_{i+1}, ...I_m$, where $I$ is a sequence of memory interactions.
5. Let the *intent* of $P$ be a sequence of interactions $I' = I_j, I_{j+1}, ..., I_n$ (not explicitly expressed by the developer) such that after $P$ is executed $M1 = m_t, m_{t+1}, ..., m_{t+p}$ $R$ $M2 = m_u, m_{u+1}, ..., m_{u+q}$ where $M1, M2 \subset M$ and $R$ is a relation on $2^M$.
6. Each interaction $I$ carries intent by imposing a set of intent compliance requirement transfers $T = (M_i, M_j, d)$ where $d$ is a primitive integrity dimension (length, parity, structure, linkage, sequence) and $M_i, M_j \subset M$.

7. The primitive integrity dimensions length, parity, structure, linkage, and sequence of memory regions in $M$ are represented numerically.

8. Then, for any $P$, interaction $I$ and compliance transfer $T$, there exists a relational expression $c$ which can be applied to all $T$ for $I$ in order to evaluate the intent of $P$ with respect to $I$, and the full intent of $P$ can be evaluated.

## 6.5 A Decomposition of Software Integrity

### 6.5.1 Static Integrity

Static integrity is demonstrated by the matching of the contents of memory with its reference measurement, with the condition that modification is unauthorized at the instant of measurement.

Thus, two prerequisites for the effective use of static integrity measurement are

1. a usable reference measurement

2. the memory undergoing measurement must be subject to unauthorized modification

The reference measurement is a value (or an acceptable approximation for efficiency reasons) whose contents are compared (possibly via the action of a function) to the memory region whose static integrity is being measured. Dynamic integrity is captured by the on-going compliance of software memory interactions with a set of requirements reflecting the intent of the software creator (not merely the extension of comparison tests to a time interval).

A major distinction between static and dynamic integrity which results in a need for unique dynamic integrity measurement methods is the property that the static integrity of a value can be measured independently of the software operational requirement. There is no requirement to query an external "oracle" process for the compliance of a data value with the reference because the reference measurement is a substitute for such a query. The matching of a data value's contents with its reference measurement implies that the value complies with the standard by which it is being evaluated. With the precondition that modification is unauthorized at the instant of measure, mismatches imply a static integrity violation.

### 6.5.2 Dynamic Integrity

*Dynamic* software integrity is the compliance of the memory interactions imposed by software with the intent of program creators. The concept of an interaction is applied to great effect in

this work to capture the nature of the dynamic integrity of execution, whether referring to the interaction imposed by a single instruction (e.g. with registers such as a binary add instruction) or the complex interaction composed by a procedure call.

The contribution to integrity of the action of software instructions beyond the contents of their operands is constructed from the particular memory state interactions (abstract functions which take memory areas as input, and may modify memory areas) which are charged to collectively carry out the program creator's intent. In fact, modern software programs are correctly revealed as high-level expressions of action which are imprecisely transformed into executable instructions which implement the creator's intent with acceptable accuracy.

Properly modeling dynamic software integrity is best approached by examining software behavior at run-time, yet in some cases integrity information is visible in static program code. The distinction is that static code analysis reveals state and action which is relevant and will be relevant during execution, but does not exhibit the integrity of true execution using machine components such as memory. Both objective value measurements *and* measurements of compliance must be employed in such analysis. Without consideration of operational integrity requirements, software whose control transfers disobey the intent of its creator are wrongly considered to exhibit sound operation. Without objective value comparisons to characterize system static integrity, any evaluation of the integrity of a computer system has no interface to its target, and so has little practical value. Ultimately, static and dynamic integrity are closely interrelated: The measurement of dynamic integrity may require the measurement of the static integrity of certain values in order to show that the program being analyzed exhibits dynamic integrity. See for example the compliance of program addresses with the control flow graph.

Dynamic integrity may be decomposed into control flow integrity, taint, and call integrity. Control flow integrity is the compliance of the execution-bound program control flow decisions with the intent of the program creator. Control flow is the selection of instructions by sequential execution or by the output of control instructions (e.g. conditionals). The inherent flow and the control instructions present in executable program code are a mechanical, transformed translation of the intent of the program creator with respect to the sequence of goal-serving operations performed. In the presence of attack, the sequences (paths) of instructions executed may deviate from the allowable set, thereby violating the intent of the program creator. The allowed control flow links

may be expressed in for example a control flow graph produced via analysis of the program code. In the context of the integrity model in use, control flow integrity is the intent compliance of the linkage and sequence interaction requirement transfers occuring during execution.

Call integrity is the compliance of procedure invocations with intent, specifically to what degree the impending memory interactions of the invocation consist of or enable integrity violations. Compliance is lost due to the combination of the call's abstract and concrete memory interactions and the properties of the input data. Protecting dynamic call integrity requires measures beyond reference comparison because the safety of a call is dependent on compliance with an operational specification rather than with a literal reference measurement.

Calls may be unsafe due to the violation of a run-time relationship along an integrity-relevant dimension between parameter memory primitive properties, which is clearly only visible at the instant of the call. In order to demonstrate the distinction between static and dynamic integrity, it is helpful to note that the literal values used as static integrity references cannot directly specify program creator intent during the execution of software. Useful dynamic integrity requirement specifications provide references for *relations or conditions* that must be respected by the memory interactions of the program in order for the intent of the program creator to be executed without deviation. Dynamic integrity conditions are thus non-functional requirements. Finally, call integrity is also expressed as the intent compliance of the integrity primitive requirement transfers between parameter memory objects which are composed within interactions by the parties to the call. Common call-based interactions are with respect to dimensions of integrity such as length, structure, parity.

Taint is the spread of untrustworthy data during software execution. For integrity purposes, the critical phenomenon is the interaction between subsets of memory regions with other subsets marked as undesirable for a given reason. This spreading interaction (via copying, etc.) exhibits the diffusion of the influence of such marked data. Although not universally threatening in itself, the spread of the influence of undesirable data is an indicator of other dynamic integrity violations and also alters the set of guarantees which can be provided by software executions. In the operative integrity model, taint exhibits as the intent compliance of requirement transfers between memory region subsets with respect to a marking or tagging dimension.

A critical final example solidifies the distinction between static and dynamic integrity. While a static integrity violation implies the corruption of memory contents, a dynamic integrity violation implies no alteration at all, but merely a use of data which causes a deviation from program creator intent. Static integrity violations cannot by definition occur without the modification of data, as opposed to dynamic integrity violations.

Dynamic integrity violations due to software (not due to external sabotage, etc.) are captured as undesirable memory effects resulting from interactions between memory objects. It is the need to measure the integrity of the *interactions* between memory items (as opposed to expressed interactions within program source code instructions) that motivates the model and methods proposed in this work to measure dynamic integrity. These interactions must be verified to be in compliance with the intent of the program creator, which is safe by assumption.

The task of dynamic integrity evaluation is to determine whether the memory interactions violate the relations or conditions in a requirement specification of intent, rather than to check whether specific values in memory match a reference. For example, attacks such as buffer overflows directly rely not on the attacker's use of a particular value, but instead on the imposition of an unsafe *relation* between the lengths of data objects between which data transfer occurs.

Dynamic call integrity can be measured by applying *dynamic integrity conditions* to procedure calls, if a set of requirements can be created which evaluates the desired call integrity property. As this work will reveal, sensible requirement specifications can be created without enumerating individual call instantiations (of which there are of course a large number), and specifications can be applied across different categories of procedure calls as default integrity checks. The creation of specifications for dynamic integrity measurement can be inserted as an additional software development step which can be partially automated with proper assistive tools.

## 6.6 Requirements for Unified Software Integrity Protection

A suitable starting point for a unified software integrity protection scheme can be found in an example from the software engineering field: a set of requirements. In forming the requirements, it will be necessary to delineate the computer system components we will protect, the scope of the components that will be protected, and the level of focus of the protection. A unified software integrity protection scheme must satisfy a set of requirements as follows:

1. Protect memory contents from unauthorized modification for their individual required lifetimes.
2. Ensure that the memory interactions of programs comply with dynamic integrity conditions provided by the program creator or his agent, with respect to
   (a) their respective explicitly defined control flow (e.g. in a control flow graph)
   (b) the integrity of procedure calls.
   (c) the propagation of untrusted data as required by the application.
3. Discharge an appropriate response to each violation detected.
4. All integrity protection engines serving these requirements must be isolated from each other with respect to resource dependence (e.g. memory), and functionally constrained.

## 6.7 Instrumentation for Integrity Measurement and Exposure

Software instrumentation is a technique which reveals valuable information about the run-time operation of software [76, 49]. In general, instrumentation proceeds by inserting instructions into the (possibly running) software in order to extract data about its performance, statistical properties, or other aspects.

In keeping with the development of software in stages based on the types of transformations being performed on it, instrumentation can be applied to software at any of these stages while realizing different benefits and limitations for each.

Instructions for instrumentation can be inserted by the developer during the source code development stage of software. This practice is common in the software development discipline at all levels such as for example the run-time writing of data to files for analysis or printing key values. Source code provides access to information that is key to integrity analyses of uses of memory which underly and eventually represent instructions. Information exposing syntax, memory linked symbols, and other contextual information which informs memory interactions is accessible to analysis implemented in source code.

However, there are inherent limitations of code stage instrumentation. Instruction address level information is generally not available at the "high level" of source code the developer faces. Other aspects unique to the assembly or object level instructions are of course unavailable.

It is also possible to employ instrumentation at the compiler stage, as demonstrated by projects such as gprof [38]. The compiler typically translates source code into intermediate representations, performs transformations such as optimizations, and finally produces object code. Instrumentation at this stage involves modifications to the compiler, which may be an undesirable requirement for the implementor of software instrumentation, especially if the objective is domain-dependent. Compilers typically target a single high level programming language, meaning that the instrumented functionality would need to be added to the compiler for each language of interest for the project. Again, certain address information and low-level data are unavailable at this stage of software production.

A linker program combines object and archive files, resolves symbols, and builds the final executable. The linker is also able to instrument software, though in this role it similarly faces limitations. Instrumenting with the linker adds the benefit of access to the libraries linked to the target program (not possible earlier in the software production process) and access to relocation information since the linker must process all code and libraries as objects and bind addresses [49]. The linker, unlike the compiler, generally operates on final generated code (the compiler largely uses intermediate code representations) meaning that instrumentation code enjoys access to finalized location information.

On the other hand, note that the source code for precompiled libraries will not be available to either the compiler or linker, preventing modification for instrumentation. Most code executed by contemporary programs is employed from libraries, and significant portions of libraries are loaded post-execution [80]. Link stage instrumentation requires making modifications to the linker, whose source code itself may not be available. Further, programs may not be available as object files.

The latest stage and product of the production of software is the working binary executable file. Addresses are generally bound within executables, and any referenced libraries are processed for linkage to the program. Instrumentation of executables as demonstrated by Larus in [49] notably does not require any modification of compiler or linker, and does not require access to the source code. Instructions can be inserted into executables as is, provided that the executable format and associated architecture (e.g. ELF, x86) are respected.

### 6.7.1 Instrumenting Software for Integrity Purposes

The choice of the stages at which to implement instrumentation for dynamic integrity measurement must take into account the effect of existing software practices on the availability of source code etc., but must for the sake of the prospect of a true unified software integrity framework include source code development *and* later stages such as executable editing.

Instrumentation inserts instructions into the program that is to be the subject of measurement. For source code, the developer may directly input code for measurement purposes without any change to the conventional process of software development. The distinction with instrumented instructions is their ability to monitor and analyze the actions of the software within which they reside.

Instrumenting via the compiler or linker or via binary instrumentation departs from the traditional software development process by inserting instructions at a phase within which *new* special-purpose instructions (that are not intentionally tied to existing instructions by the program's developer) would normally not be introduced. The benefit of this departure, though, is the revelation of addressing and symbol information, and other register level details not visible from the source.

"Late" instrumentation (instrumentation of later stage software such as binaries) can reveal values such as memory addresses which are critical to integrity in a straightforward manner, yet source code instrumentation provides access to key information linking source-level objects to execution memory regions.

The external evaluation of integrity information required for systematic, robust software integrity protection must interface with running software, meaning that executable code and data in memory must be targeted for analysis, but such analysis requires the context of information derivable from source code. As a result, instrumentation must be applied to the exposure of integrity related information to external evaluators in addition to measurement.

Software instrumentation offers unique benefits for the exposure of integrity information in addition to measurement. Since the software production stages in general exhibit a sequence of regular linked patterns of code transformation, the manifestations and results of instrumentation can be reliably carried over to later stages during the process. Source code instrumentation induces compilers to produce object code bearing the influence of the instrumented logic. Similarly, instrumented binary executable code preserves instrumentation logic and structures in the output of the

linking process in the same manner as nominal instructions, except that instruction set architecture (ISA) instructions with well-defined locations now implement the programmatic task.

Instrumentation lends itself to measurement and exposure of the full breadth of information accessible in the several phases of software development, and further the contributions at the various stages can be leveraged coherently together to achieve the relocation of dense program integrity information to an external evaluator. This practice weaves a thread of information to the evaluator, facilitating structured integrity protection architectures.

Instructions can be inserted into source code to create a context attached to existing logic which exposes to compilers and integrity evaluation engines information about memory locations used and mappings between source level objects and the memory structures which are employed to represent them by compilers and linkers. This information is necessary because the intent of software is largely expressed at the human-readable level of source code rather than the declarative, relatively explicit directives of executable code or memory manipulation.

Systematic integrity evaluation engines can feasibly analyze the information in the memory with the benefit of the context provided via source code instrumentation. If an entity with access to working memory during execution is assumed, only the proper context to coherently analyze exposed integrity measurements is needed.

With the prerequisite that integrity information must be extracted using measures taken within the various stages of software creation (especially source), the use of software instrumentation for exposure imposes little interference with program operation. The degree of additional memory space and execution time due to instrumentation is in line with the complexity of analysis attempted.

Further, while instrumentation requires only the insertion of additional instructions to an existing set all of which are processed with the same existing tools, the creation of a separate *software-based* architecture for exposure of integrity information held within program instructions would at minimum require entirely new structures which may use system resources. In addition, the creation of a software entity for measurement and exposure of integrity information would require information sharing between software processes, violating the requirements of independence and isolation of all integrity measurement proposed in this work.

Whereas the capability of software instrumentation for the efficient and unobtrusive addition of analysis functions within programs and its access to rich integrity information at both source and

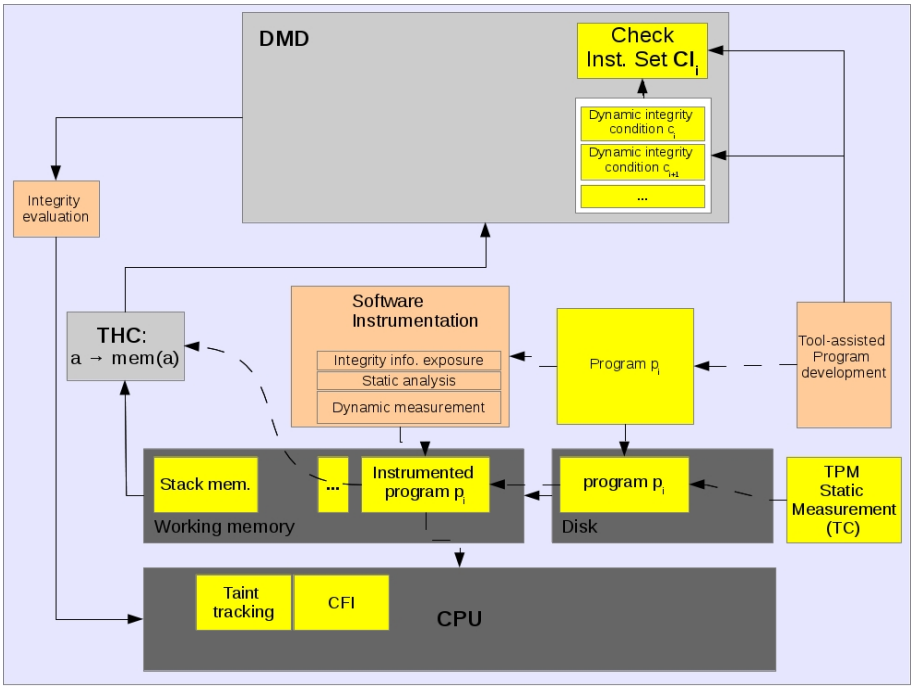


Figure 6.1: A Trusted Framework for Unified Software Integrity

binary stages of software have been detailed, software instrumentation is the ideal methodology for the measurement and exposure of integrity information for a general software integrity protection architecture.

## 6.8 A Trusted Framework for Unified Software Integrity

In order to protect the integrity of both contents and action of software, a structured, implementation agnostic framework with key engines is necessary, as illustrated in figure 6.1.

This framework adds a set of processes and objects to the software development process as well as the architectural execution environment in order to enable the static and dynamic integrity of software to be measured and exposed to external evaluation in a manner such that the measurement functionality itself is protected from attack.

The generality of this framework is provided by a set of isolated, distributed, independent, functionally constrained engines which analyze software integrity natively rather than co-occuring security violations or action profiles. Isolation of functionality is required for the components of a

general software integrity protection, since as we have noted, flaws must be assumed in a threat model which analyzes the vulnerability of both contents of programs and the effects on memory of the interactions those programs impose.

Techniques are in place to prevent the spread of violations across executions, scopes and memory locations. For example, static integrity violations of stored code are detectable and not permitted to facilitate further violations of operation. Attacks on a given individual protected program are forced to target independent, distributed integrity protection functionalities, many of which are based in hardware and thus difficult to subject to unauthorized modification.

Taint tracking and control flow integrity protection have been discussed in this work as integrity protections by virtue of their measurement of the soundness of two important types of interaction between memory-bound objects (firstly marking, and secondly sequence and linkage). However, for systematic software integrity protection these techniques are necessary but not sufficient *as they have been applied*. To achieve functionally constrained measurement of comprehensive dynamic memory interactions without exposing the measurement facility to attack, techniques for analyzing both the spread of data influence and control flow must be merged to form a hardware-based dynamic monitoring extension relying on access to the working memory.

Quite separately from broader threats, even isolated violations of either taint or control flow can facilitate further attack, but also indicate or predict attack. It presents little challenge to consider an attack which requires a particular degree of spread of data influence in order for its damage to be inflicted. Still further, program code propagation of data influence may be employed as a targeting mechanism which selects resources or subsystems for attack: as the spread reaches the desired pattern or degree, the targeted systems are affected, perhaps ignoring others with a higher requirement for exploitation. An attack on purely the control flow of a program seeks only to cause instructions not intended by the program at a given state to be executed in the context of the program's execution. Nevertheless, software execution may be tied to arbitrary external functionality, even real-world physical components. Such services or components can impose requirements on the instructions executed in terms of resource consumption or run time. As a result, depending on the constraints imposed by linked functionalities upon the program of interest, even minor pertubations in control flow may cause violations of tolerances and damage.

Thus, the necessity of taint tracking and control flow integrity protection for a unified software integrity protection is clear, yet both techniques must be adapted to fit within the broader integrity protection architecture revealed here in order to ensure sound execution. The form taken by such an adapted suite of techniques is a hardware dynamic integrity monitoring engine with access to system memory.

The framework for unified software integrity is as follows:

1. Employ static integrity measurement of memory contents supplied by elements including but not limited to Trusted Computing architecture elements including the Trusted Platform Module. This links execution to satisfactory evaluation of static integrity information and locks the memory interactions of dynamically sound execution to those intended by program creators. Static integrity references must incorporate any instrumentation applied.

2. Call integrity is to be measured with program instrumentation in order to produce integrity information primitives for the creation of dynamic integrity conditions. Instrumentation is also to be employed to expose the integrity information to working memory for the benefit of the dynamic integrity monitoring engine.

3. A dynamic integrity monitoring engine as an extension to the Trusted Computing architecture can be constructed by associating with the platform the following set of components (some of which are per-program) with appropriate trust levels based on their implementation:

   (a) Processor extensions for taint tracking, and shadow storage for taint data. This includes instructions for retrieving, storing, and setting taint data.

   (b) Processor extensions for control flow integrity measurement.

   (c) A set of check instructions CI (per program) whose function is strictly to implement the evaluation of a set of dynamic integrity conditions or relations which reflect the desired integrity properties. These conditions are constructed from expressions on the primitive integrity dimensions of memory-occupying objects within interactions composed by the program under measurement, such as length, parity, structure, marking, linkage, and sequence. The instructions in CI for a given program will necessarily vary, just as the programs and dynamic integrity conditions themselves vary. *The entity best positioned to create the set CI is the tool-assisted program creator, however some general conditions can be applied to categories of programs without the aid of the creator.* This is because the software creator holds resources with the ability to capture integrity conditions in a relocate-able way, facilitating remote measurement. In fact, there may be integrity-relevant dynamic conditions that are impossible for entities other than the creator to

create a check instruction set for because domain-specific knowledge is required to specify the condition to be checked. It is ideal for the creator of the program to supply the program itself, any static integrity metric for the program, and the set CI because the creator is well-positioned with the knowledge base to specify the dynamic integrity conditions to be evaluated by check instructions in a manner which captures with the desired scope of integrity protection. Program creators are uniquely capable of characterizing the envelope of safe memory interactions within the set composed by their code, and making performance vs. integrity tradeoff decisions during the deployment of protections. However, a remote, capable and trusted entity could be designated to produce static integrity metrics, or to produce the remaining elements with assistance.

(d) A trusted hardware channel (THC) which provides access to integrity information in memory which was exposed by framework processes. Direct memory access (DMA) is the most readily employed and appropriate hardware access method. Because the function of the THC is limited to providing a raw interface and is entirely in hardware, its operation is easily verifiable and difficult to compromise (with appropriate tamper protections). As a result, any trusted hardware source entity is well-positioned to create the THC.

(e) A trusted hardware dynamic monitoring device DMD whose function is limited to executing the instructions in CI using the THC interface for access to the computer system memory. Because the function of the DMD is intentionally constrained to executing a set of externally produced instructions, any trusted source entity (likely hardware producer) is well-positioned to produce the DMD. If a check instruction detects an integrity violation, the DMD can (for example) send an interrupt (hardware) to the processor, and the appropriate defined response can be carried out (e.g. shutdown).

By design, the components described here are distinct from the platform software being measured and the operational computer system resources being used by the platform. Isolation is an important property of a dynamic integrity measurement scheme that must be trusted and not easily attacked. Further, the functionality of these components is constrained and is intended to be standardized. This facilitates diversity of production and verification, in addition to limiting the attack surface exposed by flaws to adversaries. The integrity protection engines and programs are not exempted from the presence of flaws assumed by the threat model. Thus, functional constraint is necessary for all integrity measurement, exposure, evaluation, and response mechanisms to reduce the workspace for adversaries seeking to exercise dangerous effects on memory inherent in the execution of realistic protected software programs. The use of a software instrumentation

engine for integrity measurement and exposure represents a further functional constraint. Instrumentation inherently avoids the introduction of a separate software program context and hence any interactions which would be necessary if one were employed.

The distribution of integrity protection functionality to a set of engines for measurement, exposure, evaluation, and policy enforcement external to the protected program facilitates the practice of independence between engines. This step reduces the degree of trust necessary to place in a given set of functionalities, since composing even identical functionalities (e.g. protocol sessions) enables security vulnerabilities via resource sharing and concurrency.

Truly general integrity protection constrains the number of protected system functionalities which can be simultaneously attacked by adversaries via distribution and functional restraint. Such measures also confine violations to components rather than systems.

There is an inherent processing impact of measurement for integrity purposes. In order to prevent this impact from facilitating vulnerability, the employment of hardware structures in this framework protects program resources from vulnerability to denial of service or other attacks on availability by passing the processing burden to the hardware component.

Measurement is a core functionality of Trusted Computing structures such as the Trusted Platform Module [70]. All measurement described within the Trusted Computing specifications is via static measurement using for example cryptographic digests (such as SHA). This is expected, since static measurement of memory is well understood, and there are available, reliable methods of measurement. The task of measuring the integrity of program executions is not equally addressed by Trusted Computing, yet measuring selected aspects of running software is both feasible and often applied, as demonstrated by work on profiling software for various purposes.

However, rather than general support for analyzing the contents of or statistical aspects of run-time memory (readily available), the issue in an integrity-focused analysis is the requirement for a reliable *procedure for evaluating the compliance of memory interactions imposed by programs*, as provided in the framework presented here. General support for analyzing run-time software is not sufficient for the protection of run-time integrity. This is firstly because dynamic integrity presents as a set of *properties* which must be inferred from the uses of memory by program code. Dynamic integrity is evaluated by *non-functional* requirements. Secondly, the dynamic integrity of computer systems is in part derived from individual application-based conditions and requirements. Some

requirements for the integrity of software may be based on semantic interpretation or patterns known only within specific problem domains.

The dynamic integrity framework presented here depends on a particular critical property for its success: It must be possible to create a representation of the integrity of a running system that can be remotely measured by evaluating the check instructions CI against the target machine memory. The remote measurement of the integrity of computer system memory represents a departure from conventional security monitoring in the existing literature, but as mentioned above, it is important for the integrity measurement facility to be independent of the control of the target system in order to avoid the attacks conventional software experiences as well as attacks targeting the integrity of the measurements themselves to further a broader security violation. In addition, run-time measurement introduces significant performance penalties when applied locally on the target system (vice remotely) [1].

Software instrumentation is an efficient and effective way to facilitate remote evaluation of software integrity via the instrumentation of the target machine software to expose integrity primitive information to the measuring device. Software instrumentation is the method proposed in this work for exposing the set of manifestations of dynamic software integrity to a properly prepared measuring entity.

Control flow integrity (CFI) enforcement via software has been described in detail in section 3.5 including the considerable performance impact of local, fine grained CFI. Performing such enforcement using hardware has not undergone similar exploration chiefly due to the presumed difficulty of the adoption of modifications to processor architectures. However, as detailed in [7], the implementation of control flow integrity protection in hardware is both feasible and straightforward. Simulations of course-grained hardware CFI enforcement in that work were measured to have average run-time overhead of 3.75%, with a maximum of 7.12%. The benefit of general CFI protection to the dynamic integrity of software justifies the use of hardware approaches, and justifies its inclusion within the framework proposed here for unified software integrity.

The ultimate achievement of the framework specified here is its construction of a set of structures for the systematic measurement, exposure and evaluation of software integrity whose security is based on engines, properties, and processes rather than profiles of particular attacks. This framework rests on a foundation including a unified model of software integrity and a process for the

relocation of integrity information for software programs. A unified software integrity model reveals the components of execution which must be measured and exposed in order to ensure the compliance of running software with intent. Finally, software instrumentation enables the previously unaddressed aspects of software integrity to be analyzed and exported to systematic measurement.

## 6.9 Completing the Framework: Call Integrity Protection

As both a crucial component of the software integrity protection framework discussed, and a critical application to contemporary integrity threats, a technique for the protection of the integrity of procedure calls is presented here. In combination with the techniques already proposed to address the spread of untrusted data and the protection of control flow, the addition of call integrity protection constitutes comprehensive protection for the integrity of memory interactions composed by software execution.

This technique is based on the revelation of contemporary security attacks as sequences of integrity violations of both static and dynamic nature. These sequences commonly rely on the exploitation of vulnerable procedure calls, the integrity of which can be measured with appropriate techniques.

### 6.9.1 Goals

Since the memory interactions composed by data spread and control flow are addressed with existing techniques, the goal of this Section is to advance a novel analysis which reveals a model for procedure call integrity and its violation, along with how such integrity can be measured. This work will advance and demonstrate a method for the detection of dynamic *call integrity* violations at run-time, which are a major trigger of run-time security violations. This technique is a novel and crucial addition to the state of integrity knowledge, and to any true general software integrity protection system.

### 6.9.2 Procedure Call Integrity

In classic work, violations of computer system integrity have typically been associated with unauthorized modification of data [4, 20]. However, a form of integrity can be found in the particular memory interactions imposed by procedure calls, and significant benefits may be realized by the application of this integrity model to working programs. Integrity is best summarized as “adherence

to a standard." Fittingly, the integrity-relevant aspects of run-time calls can be evaluated against an appropriate specification so that integrity violations can be detected.

Violations of call integrity are precursors to major contemporary security attacks with far-reaching effects including complete system compromise. From the perspective of an adversary who has the ability to induce invocations of procedures with data he influences, the set of possible invocations (each of which sets an actual value for each parameter) of a procedure includes a subset of invocations which further an attack. By influencing the series of data parameters to the call, the attacker exercises existing (directed, unintended) call-bound memory object interactions such that the procedure will violate its intent or the intent of the enclosing software (in which the call is placed) to effect further damage.

Procedure calls are a fundamental, necessary unit of construction in software development and an interface between distinct software scopes with their own data sources and logic. This interface is the site of an *interaction* between the two different software scopes, in some cases due to the provision of data which is known only at run-time. Although normally some aspects of the integrity of this interaction are measurable at the level of source code where the call is described, the integrity of the call interaction can in some cases be measured *only* during the execution of the software.

Below the level of the interaction between software scopes, however, the call interaction composes a set of integrity-relevant *interactions between memory areas* placed to implement the goal of the procedure called. These interactions are enclosed by an abstract action presented to prospective calling programs, yet nevertheless the interactions impose effects on memory areas with respect to primitive memory-bound dimensions of integrity such as length, parity, or structure. It is these memory effects which may result in violations of the integrity of memory contents or the execution of software in their failure to respect intent compliance requirements transferred between memory objects.

In major modern computing architectures, data exhibiting call integrity during execution can be found on the run-time call stack, organized at the end of the stack in a manner so as to be available for the approaching call. This work proposes to measure the integrity of the procedure call by exposing the interactions implied by this memory-bound presentation of parameters, and evaluating its integrity with respect to a specification of call integrity conditions.

Beyond violations of call integrity, the techniques revealed here detect to-be-discovered attacks which require unsafe calls of procedures as prerequisites or corequisites, a common scenario. Even the relatively recent Return Oriented Programming style of attack requires an initial hijack of control flow, which is most often achieved via unsafe invocations of vulnerable procedure calls [63].

### 6.9.3 Modeling Call Integrity

Computer system software integrity manifests in several ways, critically including some manners which are tied to interactions between memory areas rather than the instantaneous validity of a data value with respect to a reference. To facilitate the protection of the operational security of computer systems, this work proposes to detect violations of integrity that are visible in system memory upon reaching procedure calls during execution. Since all memory state changes are potentially integrity-relevant, the term memory used here is global, and hence includes cache, working instruction and data memory, registers, etc. However, a key location for analysis of call integrity is the program stack in working memory. We begin with the following basic assumptions:

1. execution of procedure calls will complete.
2. consider only integrity-relevant memory uses arising from the program(s) being executed or the execution environment

Each procedure call $c$ imposes a set of memory interactions (with impending *effects*) $C_c$ which can be partitioned into a set of interactions which are *directed* by either the procedure call *program author*, $C_c(author)$, or the *caller(s)*, $C_c(callers)$. Recall that memory interactions are the condensed, integrity-relevant representation of sequences of instructions in this work, and that such memory interactions impose transfers of requirements for compliance with intent. Dynamic integrity is preserved when these requirements are respected. Note that although the protected codebase contains the procedure call definition, an attacker is here considered a party to the calling of the procedure because he may influence its inputs and some aspects of the call (e.g. timing). Each interaction is directed, but may or may not be *intended* by one of the parties to the call. In the present work, a subset of the interactions directed by calling parties is assumed to be potentially malicious.

Concrete memory interactions alter the contents of well-defined, usable memory locations toward the programmatic goal, as in the statement 'add r1,r2,r3' which would place the sum of the

values held in two registers into a destination register. Corresponding to the concrete memory interactions bound to a call, there are *abstract* memory interactions such as one intended by the call author. These abstract interactions implement qualitative changes lacking specific address or length parameters such as "copy the value in the first parameter to the second parameter," and are inherent in the interface presented by a procedure call to its callers. This interface is presented to calling entities who induce the call with the expectation that the abstract interactions will occur as promised.

With these assumptions, violations of call integrity arise when, as a result of the particular memory interactions imposed by a procedure call instantiation, $C_c(author) \cup C_c(callers)$ includes a subset of memory interactions that either

1. are not in compliance with the intent of the call author(s),
2. cause effects which are static or dynamic integrity violations.

It is within the set of call-bound interactions that dynamic integrity manifests and can be measured in order to protect call integrity. Note that the call-related memory state interactions may include effects implemented by execution environment software (operating system), and such interactions are nevertheless directed by the user program or the adversary.

### 6.9.4 Security Violations as Sequences of Integrity Violations

Common contemporary security analyses categorize attacks or vulnerabilities by mechanism, locus of action, or perhaps by the effects of the attack, e.g. arbitrary code execution. While suitable for purposes of unique identification, this level of analysis does not reveal the degree to which integrity is exhibited in these attacks. This work reveals that current major security vulnerabilities coincide with sequences of integrity violations, each of which are either static or dynamic in nature. Techniques which identify and interrupt such sequences can thus be applied to attacks in order to provide protection.

Procedure calls are both a fundamental interaction construct within software production and an instant at which far-reaching integrity violations can be triggered. The run-time violation of the integrity of procedure calls by causing invocations which instantiate a dangerous memory interaction set is a key tactic that has led to countless broader security violations.

Memory error (e.g. buffer overflow) vulnerabilities have been the basis for innumerable security attacks [57] and remain an ongoing threat, despite numerous efforts to counter them (see for example [2, 23, 75, 22]). One infamous, ubiquitous, and critical type of attack which relies on unsafe invocation is the calling of a data copying function that writes the source data to the destination location without regard to the length of destination object memory available. This type of call enables memory corruption, diversions of control flow, and/or complete computer system compromise. The vulnerability exposed by missing bounds-checks for memory-writing operations has been a starting point for innumerable critical security vulnerabilities in major software [57, 13].

An additional appropriate example is the invocation of format conversion functions (such as printf in C) in a manner that exploits disparities between the format string and the input parameters (the parity primitive). This attack enables information exposure or memory corruption.

Fortunately, these attacks can be revealed as sequences of interdependent static and dynamic integrity violations, and thus addressed by applying protections to interrupt the sequence. For example, a common type of buffer overflow attack targets a stack memory location holding an address (critically the procedure call return address). The attacker exercises a software vulnerability with a procedure call whose directed memory interactions include a write to a memory location.

A safe call to a data copying procedure must obey a length compliance requirement transfer between parameters in which the destination memory length is not exceeded. If the transfer is not respected, a chain of integrity violations can be triggered such that the attacker gains complete control of execution.

In Figure 6.2, a memory write interaction imposed by a procedure call overwrites a return address in working memory, thus compromising the static integrity of the contents of the address. The corrupted address is supplied to a use interaction as a control flow jump destination, an action which amounts to a violation of dynamic integrity as control flow diverts from the intent directed in the protected program housing the vulnerable procedure call interface. Beyond influencing the content of an address that will be used for control flow, an attacker is commonly able to determine the address content to his advantage (e.g. shellcode). This means that arbitrary instructions can be jumped to, and an unbounded attacker-influenced "program" is executed whose memory interactions are not expressed in any explicit stored program code.

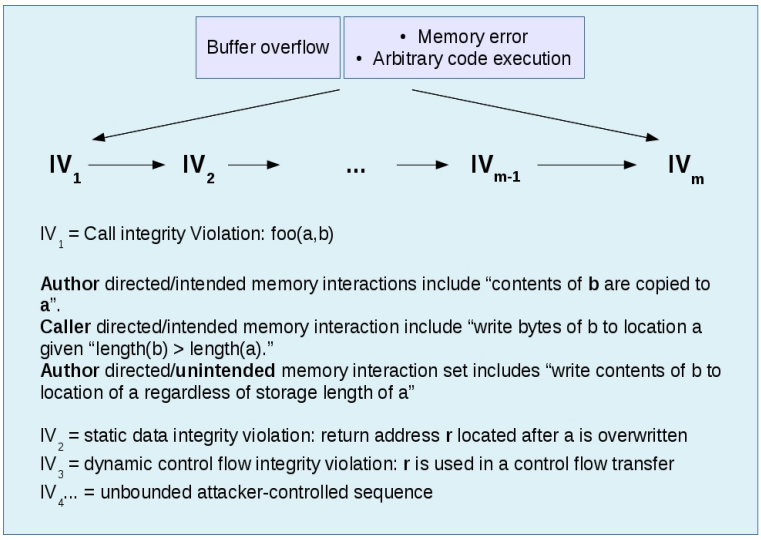


Figure 6.2: A Buffer Overflow Attack as a Sequence of Integrity Violations

To protect the dynamic integrity of the call from memory interactions which are dangerous with respect to the length integrity dimension, the destination parameter length in memory can be measured on the stack frame arranged for the impending call, prior to the execution of the call instruction. When the techniques in this work are applied, the lengths of the source and destination parameters are exposed for measurement during execution, allowing detection of unsafe invocations with respect to a specification.

Format functions are certain C functions which accept a format string in addition to conventional data parameters. These functions transform data into output strings or extract data from strings. The format string is a character string containing a special conversion character sequence (for various data types and operations) for each data parameter that will be part of the conversion process between data and output strings. Format string attacks [64] exploit an attacker's ability to manipulate the correspondence between format strings and the remaining input parameters within format function calls, such as printf.

Given that format functions must accept arbitrary numbers of parameters and the format string allows arbitrary string content, an opening for attack exists where there is an intentional mismatch between the parameters passed and the conversion sequences within the format string (see Figure 6.3). This exploitation is of a "channeling" vulnerability because the call enables the

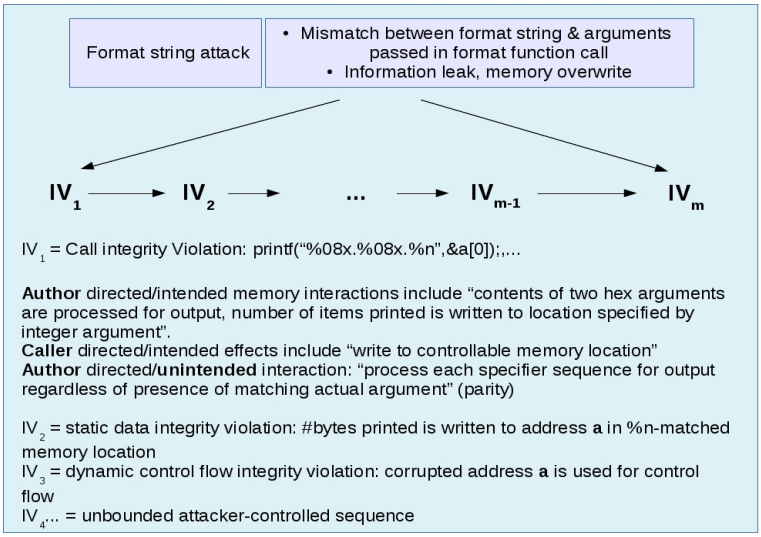


Figure 6.3: A Format String Attack as a Sequence of Integrity Violations

transferring of distinct information types in a merged channel such that switching between the control and data requires special input. Attackers who induce the calling of format functions on data they influence can initiate security violations of various types (information exposure, memory corruption) depending on the type of conversion specifiers used in the format string. In terms of the call integrity model in use, format string attacks exploit the direction of memory interactions which impose requirement transfers across the parity dimension.

For example, when a format function processes a format string conversion sequence with no corresponding parameter, leaks of program data from working memory can result. The format function maintains memory access to parameters in working memory with pointers. Disparities between format string read conversion sequences and actual parameters can cause the format function call to read working memory which was not indicated by a true parameter. Certain specifier codes which write diagnostic information to supplied memory locations can be abused to corrupt arbitrary memory locations, resulting in program crashes or even undetected memory corruption (if the write is within a mapped area). One example is the use of a special conversion sequence which writes the number of characters printed to a supplied memory address (pointer).

In Figure 6.3, the printf format function is called with a attacker-influenced sequence of parameters including a format string. The diagnostic format string sequence '%n' is used to cause

the write of arbitrary bytes to a chosen location in memory, and this process can be extended to overwrite full memory words or addresses. This static integrity violation is succeeded by a dynamic integrity violation when the overwritten address is used for control flow. Attacker-controlled control flow jumps enable unbounded attack sequences.

Return-Oriented Programming (ROP) [63] attacks the operation of programs during run-time by *chaining* together the execution of sequences of executable instructions found within existing programs or libraries. These sequences or "gadgets" can be combined to execute arbitrary computations. ROP presents a significant challenge to existing integrity protections because it does not rely on the presence of any external attack code for the gadgets it uses [11]. However, techniques have been proposed that increase the burden of the potential perpetrator of ROP attacks. In [40], Hiser et al. randomize the locations of executable instructions at run-time, preventing ROP attackers from reliably finding sequences of instructions needed to form attacks.

One key prerequisite of conventional ROP attacks is the requirement to initially hijack a program's control flow in order to direct execution to one of the gadgets the attacker has found in the available code on the system [63]. Outside of such attacks, execution would normally not follow these creatively chosen instruction sequences that are mere parts of working programs. Thus, techniques such as buffer overflows can be used to divert control flow to gadgets. This dependence on a violation of control flow creates an opportunity for detection of attacks as call integrity violations: The abuse of the memory interactions of procedure calls is a major category of technique used for the diversion of control flow.

Beyond the major examples mentioned, call integrity protection can be applied to software to protect against undiscovered attacks in which attackers induce procedure calls to impose unsafe memory interactions. The integrity of these interactions is evaluated with respect to conditions on the call-imposed relations between dynamic integrity dimensions (e.g. length) of the memory objects used within the interaction. In addition, with appropriate dynamic integrity specifications as shown in Table 6.1, call integrity protection is applicable to existing documented vulnerabilities.

In the remaining sections, we will reveal that a combination of regular manipulation of program source and executable code along with specification enforcement can be used to protect procedure calls from exploitation. This work demonstrates that the integrity of procedure calls can be measured and exposed, thus enabling the detection of violations. Since the integrity of the relations

Table 6.1: Call Integrity Specifications for Existing Attacks

| Name(s), Description | Software | Summary | Call Integrity Violation | Call Integrity Specification |
|---|---|---|---|---|
| Heartbleed, CVE-2014-0160 | OpenSSL | Missing bounds check in server side TLS extension packets | Data read parameter value exceeds safe server limit | Read parameter must obey server limit (tool assignable) |
| CVE-2006-3459 | Adobe Acroread 9.3.0 using libtiff vj3.8.2 | Stack buffer overflows in tiff image parse code, arbitrary code execution, denial of service. e.g. tdir_count large value | Parameter value exceeds application limit | Parameter length condition for tdir_count, similar for other overflows |
| CVE-2009-4035 | Xpdf PDF viewer library | Malicious PDF input. Buffer overflow, integer overflow, null pointer dereference, infinite loop | 1. Input processing routines accept too-long input 2. Input routines accept PDF input with crafted type 1 font, triggering code execution | 1. Parameter length condition 2. Application dependent input structure condition |
| CVE-2010-5081 | RM-MP3 Converter 3.1.2.1 | Stack buffer overflow to arbitrary code execution | Playlist file over-length URL argument accepted | Parameter length condition |
| CVE-2007-1195 | XM Easy Personal FTP Server 5.3.0 | Multiple buffer overflow, format string vulnerabilities | 1.Over-length parameter passed, or 2.format string/argument list mismatch | 1. Parameter length condition 2. Run-time parity condition |
| CVE-2008-3408 | Coolplayer v2.18, v... | Stack buffer overflow allows arbitrary code execution | Crafted m3u input file contents exceed app. based boundary | App. dependent parameter length or structure condition |

imposed by procedure calls on length, parity, etc. of parameters can be exposed and measured during execution by the methods revealed here, this protection transfers to any attack scenario which is dependent on the exploitation of the procedure call interface during software execution, if an appropriate specification is applied.

### 6.9.5 Realizing Call Integrity Protection

The effort to realize call integrity protection in this work can be summarized in three processes:

1. Instrumentation
2. Instruction analysis or *introspection*
3. Specification creation and enforcement

Instrumentation involves inserting instructions into the target program in order to enable logic or analysis distinct from that of the target application. Typical applications of instrumentation include profiling and debugging of software. Profiling applications typically employ instrumentation to cause the target program to expose some analysis information, for example performance-related

measurements such as run-time statistics for procedure calls. Instrumentation for call integrity protection inserts source code or binary instructions for the measurement and analysis of integrity primitive properties, the production of integrity information, and the exposure of information to evaluation.

Instruction analysis is the inspection of the contents of target program instructions (operands, opcode) by an analysis program. In the present work, this analysis is needed as a run-time interface to the signaling and marking data exposed by the instrumentation process for the required external evaluator of integrity. This is a level of access provided by for example a process virtual machine environment, debugger, or specialized software introspection library. At the level of instruction introspection, aggregate analysis and detection of integrity relevant events can be performed across instructions (e.g. analyzing relationships between all parameters to a run-time call).

Instruction analysis can be implemented by applying binary instrumentation to executables. Binary instrumentation provides the capability to analyze the integrity-relevant information which resides in the memory interactions composed by executable instructions. Although the instructions themselves can only manipulate true memory contents during execution, their operands and op-codes expose usable addressing label information, memory locations, integrity-relevant memory use patterns and sometimes integrity data. Binary instrumentation can be carried out during execution with libraries such as DynamoRIO [34].

Finally, a specification of dynamic call integrity conditions must be created and enforced. The creation of call integrity specifications requires the input of the program creator for application-dependent requirements, but the remaining requirements can be fashioned from generalized conditions on the integrity primitive properties (dimensions) of memory objects (e.g. length, parity or presence, structure) which are applicable to entire categories of instantiations of calls.

Enforcement may be implemented as part of any of the several call integrity processes mentioned, with varying levels of difficulty. Since integrity information is in some cases not accessible until execution, enforcement may require delay to later processes in sequence.

The processes employed for call integrity protection are applied in three stages:

1. Measurement and exposure of call integrity information
2. Analysis of integrity information for detection of call integrity violations
3. Policy-based response

Before the integrity of function calls can be efficiently measured on the working memory run-time stack, methods are needed to impart two important properties to the stack data for the call: isolation and delineation. In order to impose isolation on the memory storing the parameters of function calls during execution, the source code call can be instrumented to insert a *preamble* of single parameter "split calls" and other call integrity-related instructions. For each parameter listed in the original call which is subject to integrity measurement, a split call and associated measurement instructions must be inserted.

When the instrumented source code containing a call is compiled, each split call results in a corresponding set of executable instructions for call site processing setup where the single analyzed parameter is written to the end of the stack prior to the split call's stack frame being allocated. In this way, the split calls isolate individual function parameters on the stack for measurement such that there is little interference with normal program operation, *prior to the original call.*

To delineate parameters on the stack for measurement, the program is also instrumented to generate information required for the enumeration of parameter memory. All memory-enumerating operations for integrity measurement require methods of distinguishing between integrity-relevant memory-occupying objects and other content. Other basic analysis can also be completed during the instrumentation process, e.g. extraction of integrity information which is visible in source code, such as the number of regular parameters.

The techniques discussed expose the memory regions bound to function call parameters in an orderly, delineated manner on the stack. This ensures that the primitive integrity dimensions of the memory regions which are caused to interact are accessible for analysis prior to the impending call. The remaining task is to apply monitoring to these data to detect call integrity violations and then take appropriate action. There are various options for the manner of detection.

### 6.9.6 Detection of Integrity Violations

One method of detecting call integrity violations is to simply instrument the target program with additional instructions into the source code or executable code to detect unsafe calls. However, the framework proposed in this work requires dynamic integrity information to be measured by a remote engine as a fundamental condition for software integrity. The detection of integrity violation events is not an exception to this requirement. Thus, detection must be carried out by

an engine independent from the execution scope of the target program and pure instrumentation is not sufficient for detection.

Should instrumentation be employed to expose integrity information for external evaluation, the instrumented program instructions are amenable to analysis by appropriate analysis techniques. Source and executable program instructions do not normally possess the capability to perform meta-analysis: analysis of their own contents and operands.

In fact, practical exposure of certain call integrity information during execution requires access to the individual components (operands) of instructions and persistent, locateable storage of analysis information. This is a manner of access provided by for example, a process virtual machine or a run-time introspection environment. The fact that source instrumentation does not provide such access means that more measures are needed to perform the required analysis for this work.

To achieve the necessary access for integrity monitoring, software instrumentation is augmented with a software entity bearing the ability to perform run-time analysis of executable instructions. One such entity is a process virtual machine (PVM), which occupies the same working memory as the target program and acts as an intermediate layer of control between the execution environment software (e.g. operating system) and the target program. Since the PVM has access to the working memory of the target program as well as the run-time stream of target program instructions, it can be implemented to analyze primitive integrity information which has been exposed and is accessible via analysis of executable instructions. The PVM allows the analysis of instruction components at run-time and persistent storage for cross-instruction analysis in a regular fashion.

The use of a PVM-like call integrity analysis engine lends itself to remote measurement of integrity as an extension to architectures such as that introduced by the Trusted Computing Group [73]. A dynamic integrity engine is a fitting addition to the Trusted Computing (TC) architecture in order to address the integrity of memory interactions, the remaining aspect of software integrity not analyzed by TC techniques. The interception and analysis of integrity information by such an engine facilitates the development of standardized measurement entities which check the integrity of software during execution via analysis of working memory. For the benefit of isolation and robust measurement, the natural next step is the development of a specialized hardware evaluation device which would reduce the performance impact of run-time call integrity measurements (since the device relieves the local platform software of the burden of measurement) and prevent the violation

of the integrity measurement process itself. Fundamental to sound software integrity monitoring (and all monitoring) is the employment of measures to ensure that the measurement process itself has integrity (meta-integrity).

The final stage of call integrity protection is the discharge of policy responses to integrity-relevant events. The most readily available response is to abort operation, which prevents progagation of integrity violations and avoids the issue of computational dependencies between procedure call effects and other code. However, it is the contention of this work that it is possible to exert a less disruptive response to integrity violations that preserves the existing call's execution to some degree in a way that is transparent to the application. This type of intermediate response to call integrity violations alters the intent compliance guarantee that can be provided by the protection scheme concerning the protected program, a fact which must be made evident to users of the program. The development of such a response is left to future work.

### 6.9.7 Dynamic Integrity Conditions as References for Call Integrity

In order to protect procedure call integrity, a specification for evaluation of run-time call integrity is needed. Rather than a specification of the functionality of a procedure, what is needed is a non-functional specification of dynamic integrity conditions required for the integrity of the call with respect to the intent of the program author. Such a non-functional specification provides conditions for the evaluation of correct operation rather than a declarative enumeration of proper behavior. Two major examples of dynamic integrity conditions have been suggested in this work:

1. the number of regular procedure parameters present in memory at run-time must match the number implied by the format string parameter in the call.
2. The length of parameters in memory prior to the call must satisfy a given relation

Although an exhaustive set of dynamic conditions on memory primitives for all call applications may be large and difficult to produce, it is possible to partition the set of possible run-time calls into first a set of calls which exhibit the desired level of integrity and a remaining set of calls which require some response. It is also possible to produce useful generalized conditions which require no intervention by software developers, or can be assigned with minimal effort with assistive development tools.

Clearly, for the famous run-time buffer overflow attack in which source data object memory is copied past the destination memory boundary, a sensible default condition (and one which does not requiring developer input) is one in which the destination length primitive cannot be exceeded by any potential data write interaction. In addition, a trivial specification which can be used to detect run-time format string attacks is a parity condition in which the number of non-format string parameters must match the number of parameters implied by the format string accessible in the source code call.

However, ultimately the proper approach is the inclusion of both such specifications into a larger strategy of building a specification of dynamic call integrity conditions per program which relies on sensible tools to assist developers in partitioning the execution call space for the desired level of integrity and choosing appropriate universal conditions as defaults. Finally, some call integrity specifications may need to be based on domain-specific requirements which are not purely relational in nature, requiring the program creator to intervene.

The example specifications suggested are part of a large group of potential non-functional dynamic integrity conditions for the integrity of calls. These conditions are equivalent to an application of mathematical relations in which the relation is the subset of instantiations of call-imposed sets of memory interactions which obey an expression on the primitive integrity dimensions of (e.g. memory length, structure, pairing) parameters and thus prevent non-compliant memory effects. The goal of the protection scheme is therefore to restrict the executions of running procedure calls to those whose interactions are part of the relation and thus guarantee that only compliant memory interaction effects can occur.

Presuppose that the integrity of the call has been exposed by the techniques in this work. In a buffer overflow attack on a data-copying function, an executing call with any instantiation for the source and destination parameters in which the length of the source parameter is at most the length of the destination parameter satisfies an "at most" relation which can be applied in a dynamic call integrity specification to detect the impending overflow before the call is executed. A dynamic call integrity specification which detects format string attacks is a relation where a "matches" relationship holds for each conversion specifier-parameter pair. This relation is the subset of executing calls in which each conversion specifier in the format string parameter has a

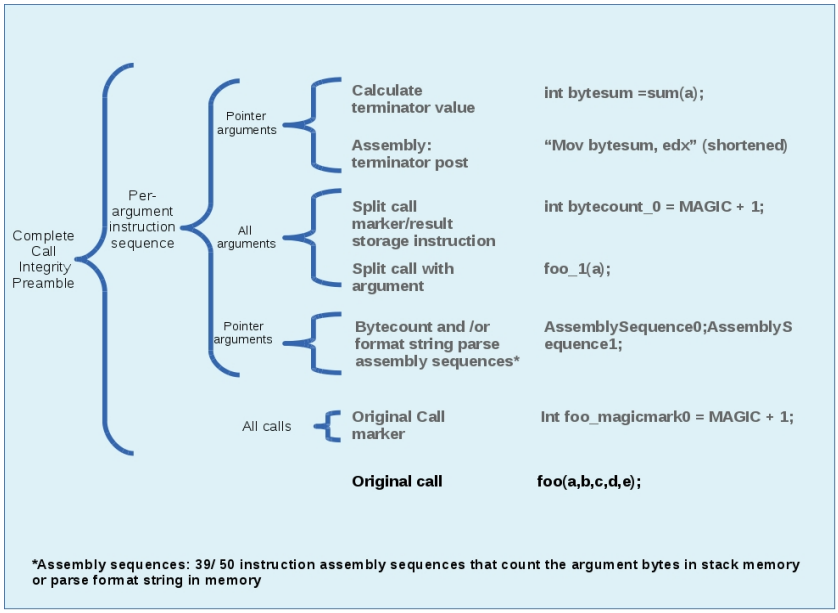


Figure 6.4: C Source Code Procedure Call Instrumentation

matching parameter in the call at run-time. Both relations specified here can be easily translated into instructions for detection of violations.

For a given call, a set of relations of the above type can be created which are used to evaluate the integrity of the running call using the available exposed integrity primitive information as input.

## 6.10 Demonstrating Call Integrity Protection with Source and Binary Instrumentation

In keeping with the distributed, constrained nature of the framework proposed in this work for the protection of overall integrity, dynamic integrity must be measured, exposed to a measurement entity, evaluated, and appropriate policies enforced. In this section, the methods necessary to implement the measurement, exposure and evaluation are detailed. Since control flow integrity and taint can be measured with existing methods (in hardware), the focus of the demonstration here is procedure call integrity. A prototype implementation of the techniques here including performance evaluation is discussed in Chapter 7.

In order to measure parameter primitive integrity information in working memory during execution, within the instrumented call preamble special assembly instruction sequences are inserted

succeeding the split call for the parameter. These instructions implement byte-enumerating measurements of integrity primitives including length and parity. Structure is an additional integrity-relevant primitive property for memory-occupying objects, yet conditions on structure are typically application-dependent and are producible from conditions on length and presence.

The byte-enumerating measurements introduced here rely on memory-bounding conditions produced by instrumented source instructions. These instructions store values which enable later measurement code to reliably restrict its analysis of working memory to the region of the parameter. For the instrumented condition-generating instructions and the full layout of the preamble of instrumentation instructions inserted, see Figure 6.4. The split calls inserted for each parameter ensure that the parameter is isolated in a predictable stack location in memory, and that its bytes can thus be examined in working memory before the byte measurement sequence is reached in the program source code. Upon the completion of the assembly sequence, the bytes of the parameter are exposed to measurement at run-time for integrity purposes to both the source code level and (more importantly) the instruction introspection layer. This technique is the basis for all dynamic call integrity conditions which require the enumeration of parameter bytes.

Binary instrumentation was performed with DynamoRIO, a run-time software-based instruction manipulation library which inserts instructions *at run-time* into the target program *as it executes* [34]. After instructions are inserted and analysis carried out, program basic blocks are directly passed to execution.

DynamoRIO is implemented so as to provide access to a set of key events related to executable basic block execution such as basic block arrival, system call, and thread initialization. These events can be accessed via callback functions that are accessible to client applications which jointly analyze the target program along with DynamoRIO code. The basic block related event provides access to the full basic block of instructions for analysis or modification, including control flow instructions, critically *call* instructions and the instructions which precede them.

During its interval of access to executable instructions, DynamoRIO provides full access to the components (operands) of the instructions for analysis or modification. This instruction introspection capability (a consequence of DynamoRIO's process virtual machine implementation) is employed here to intercept integrity information exposed by the source instrumentation process (see Figure 6.4).

Introspection is also used to implement a heuristic for detecting the number of run-time procedure call parameters present in working stack memory for the approaching call without source code access. The heuristic is based on the identification of a set of pre-call executable instructions which correspond to the preparation of each call parameter for access within working memory. The count of parameters is critical (for example) for the detection of attacks which exploit differences between the number of parameters expected and the number present at run-time, namely format string attacks. The detected number can be compared to the number of parameters referenced in format strings in order to identify these attacks. The demonstration of this heuristic is made possible by the DynamoRIO instruction access described above, and does not require access to or modification of source code.

In this demonstration, the deduced run-time parameter count can be compared to counts derived from instrumentation for parameters exerting parity requirements (e.g. format strings), even if these counts are unknown until execution. The instrumentation-derived counts are exposed via cross-layer signaling instructions to instruction introspection entities such as DynamoRIO. In the case of format functions, the format string is parsed in source or during execution to determine the number of implied parameters requiring a matching regular parameter. This number is compared to the number of remaining parameters in order to form a dynamic integrity presence condition which can be applied to the detection of format string attacks.

The additional space and time required for the implementation of call integrity protection (per call) as detailed here can be derived from several factors:

1. The number of parameters in the call
2. The length of the parameters.
3. The number of conditions evaluated for the call
4. The number of primitive measurements per condition

Since the number of parameters to procedure calls is in general not expected to be large or of unmanageable size, it is reasonable to expect that the overhead as a function of parameter count can be bounded. As a result, dynamic integrity conditions based on parity or counts do not represent a significant performance burden. Analysis of the primitive memory length dimension will require enumeration in memory, thus tying performance to the length occupied by parameters in working

memory. The overall call instrumentation augments the program with a set of instructions *per analyzed parameter*, including source and assembly instructions for determination and exposure of byte-enumerating conditions, exposure instructions, and finally integrity primitive measurement instructions.

The computational overhead of dynamic integrity specification evaluation is related to the number of conditions included in the specification and their complexity. Details of the implementation including performance evaluation are left to chapter 7.

### 6.10.1 Pointer-type Parameter lengths

One initial critical component of call integrity is the set of relationships between the lengths of variable length parameters (e.g. C pointer), which is in some cases known only during execution. For true integrity measurement, the working lengths of parameters must be measured via *direct analysis of the working memory* regions holding those parameters.

There are two necessary abstract capabilities for the measurement of parameter lengths in memory. Firstly, it must be possible to locate the memory area dedicated to the particular parameter, as distinguished from other parameters or distinct memory contents. Secondly, a reliable method for determining the boundary of the memory holding the parameter is required, a fundamental requirement for any process which enumerates the contents of raw memory representing a given object.

The instrumentation of the protected program source code call with an appropriate set of instructions allows the precise measurement of procedure call parameters in working memory. To impose order on the layout of working memory in order to locate each parameter, a *split call* technique is applied. For each parameter requiring the raw, in-memory measurement of its length, an instruction to perform a nominal call on that parameter is inserted. The call has no effect other than to cause the working memory stack to be arranged in an advantageous way to isolate the parameter memory at the end of the run-time stack, while leaving the logic of the surrounding program code unchanged. The necessary definitions for all inserted instructions can be carried out along with instrumentation in a programmatic fashion by an automated script.

In order to make available a terminal condition for the byte-wise count of the parameter memory during run-time, instructions are inserted to calculate the integer sum of the individual bytes of the

parameter memory. This sum value is then placed into a register by a special sequence of inlined assembly instructions so as to be available for the coming counting process.

Finally, the remaining task is to count the units of memory dedicated to the call parameter residing in working memory during execution. For this, a special inlined assembly instruction sequence is inserted which counts the running sum of the bytes encountered in the parameter's contiguous, working memory on the stack. During the count, the running sum exhibits the following property:

$$running\_sum \leq bytesum(arg) \rightarrow number\_of\_bytes\_encountered \leq bytelength(arg)$$

Together with the fact that each pointer-type parameter has exactly one byte sum, this allows the length of parameters in bytes to be measured during execution in working memory, represented by the $number\_of\_bytes\_encountered$ value above. Note that although more than one parameter (which is composed of a set of byte values we interpret as integers) may have an identical bytesum, the terminal condition derived here for the bytecount is valid for any of the set of parameters having the identical sum.

One advantage of this method of counting the bytes of an parameter is that it does not require static source code analysis for the provision of the boundary condition sum. Additionally, the use of a byte sum means that this method can determine the actual *bytes used* within parameter memory, rather than the number of *bytes allocated* for the parameter. This demonstration method fulfills the key goal of measuring the true number of bytes used by parameters in live working memory during execution.

Since a small number of source code instructions are inserted per parameter and no complex computation is involved, the performance impact of this type of integrity measurement is expected to be minimal. However, the fact that a byte enumerating process is employed means that the execution overhead will be related to the length of parameter memory contents.

Without access to source code, several significant obstacles prevent the reliable measurement of parameter byte lengths in memory (see Table 6.2). Such a lack of access is a realistic concern due to factors including the proprietary, closed source nature of some programs and dynamically generated code (e.g. just in time compilation). Assuming only access to a program binary which is in a state prepared for execution, the analysis lacks general purpose access to the contents of the

Table 6.2: Call Instrumentation Measurement Options

| Integrity Measurement | Source code | Argument Length | Call Argument Count | Format String Count Extraction |
|---|---|---|---|---|
| Source Code instrumentation | Available | Compile or Run-time | Static Analysis | Compile or Run-time Format String |
| | No | No | No | No |
| Source Code + Binary Instrumentation | Available | Compile or Run-time | Static Analysis or Introspection | Compile or run-time Format String |
| | No | No | Static Binary Analysis or Run-time Introspection (heuristic) | No |
| Binary Instrumentation | | No | Static Binary Analysis or Run-time Introspection (heuristic) | No Method Found |

parameter, preventing the derivation of a terminating condition for a byte-level count. Although in general instructions for writing parameters to working memory can be located by analysis of the binary program, no accurate mapping between a particular parameter and the set of instructions which are determinative of its final pre-call value can be guaranteed to be found. The ultimate source of this problem is a lack of context information normally available in source code, resulting in an inability to distinguish separate variable-bound working memory areas from each other or from memory used for other purposes. Therefore, without unrealistic restrictions on the use of variable-bound memory for programs in C, these experiments demonstrate that with only binary executable access there are severe obstacles to the measurement of pointer-type parameter lengths in working memory which may prove insurmountable.

### 6.10.2 The Relation of Parameter Count to Format String

A second critical aspect of procedure calls which holds valuable integrity information is the parity between parameters, some of which impose presence requirements on others. The number of parameters referenced by *format string parameters* is an important value which is necessary for the detection of a major category of attacks.

Recall that format functions are a type of conversion function typically tasked with conversion of formatted input to or from character strings. Examples of C language format functions are *scanf* and *printf*. Format functions accept a format string, a special string parameter which must

accommodate an arbitrary number of conversion specifier codes for various data types, each of which must be matched by a regular parameter to be converted using the specifier. The number of specifiers in the string may be known at run-time only in some cases when it is derived dynamically. The abuse of the relationship between the parameters supplied and the format string given has been the source of an entire category of security attacks [64].

For the measurement of dynamic call integrity related to parameter parity, source code instrumentation is applied in this work for the analysis of format strings and procedure call lines during the scripted source code instrumentation process. If the format string is visible in source code, it is parsed to determine the number of format specifiers. The number of non-format string parameters is extracted from source code analysis during the instrumentation pass. For demonstration, these two values are compared *during execution* via the insertion of instructions for simple numeric relational comparison, a mismatch indicative of a call integrity violation and a potentially dangerous call. The necessary exposure to the external evaluator for ultimate evaluation is carried out by exposure methods detailed below. If the format string is known only at run-time, the methods for isolation and bounding of parameters used for parameter length measurement are reapplied here to allow the format string to be analyzed in working memory. Inlined assembly instructions are inserted to parse the format string in memory to determine the number of format specifiers, each of which are identified by a '%' in C language. In order to use the integrity information deduced with this method, this count is compared as before with the number of non-format string parameters by inserting source code instructions for a relational comparison. The result of this method is a value indicative of the dynamic integrity of conversion type function calls during execution.

### 6.10.3 Deriving the Parameter Count via Binary Instruction Analysis

Although there are significant obstacles to the run-time measurement of certain integrity primitives without access to source code, experiments conducted here demonstrate that with a run-time instruction manipulation/introspection system (DynamoRIO [34]) it is possible to measure parameter count prior to a procedure call during execution without source code analysis.

Common register memory architectures such as Intel x86 allow the programmer to perform operations on memory locations and registers. In the experiments completed here, analysis performed on the binary executable code of C procedure calls reveals that instructions are transparently inserted to prepare the working memory stack for the approaching call. This set of instructions

contains a subset dedicated to placing call parameter values into memory in a generally contiguous fashion from the "end" of the stack, the address of which is held in the ESP register in x86.

The above observations suggest a heuristic for identifying the number of parameters passed to the procedure call: count the number of instructions which move values (or their locations in nearby memory) to contiguous memory as measured from the end of the run-time stack.

As mentioned, DynamoRIO allows run-time instruction manipulation of executable programs. Arbitrary transformations can be applied to instructions within any basic block prior to its release to execution. The system also provides a callback facility to run general analysis instructions when key events occur during execution of the target. It is this capability that enables the implementation of the parameter counting heuristic, and in general access to integrity information when it is exposed by appropriate methods. In order to demonstrate the heuristic, the event signaling a basic block arrival is used as an attachment point to begin analysis code. When a basic block arrives for processing by DynamoRIO, we impose a check for procedure call instructions (which will necessarily end basic blocks, due to restrictions of the library). If the block ending instruction is a call, we are able to analyze and transform the call instructions and all previous ones within the block.

To implement the heuristic, it suffices to scan the instructions from the terminal call in a reverse order, analyzing the operands to detect instructions which write (e.g. mov in x86) to contiguous locations relative to the stack pointer address (base + displacement).

### 6.10.4 Exposure of Integrity Primitive Information

In order to demonstrate the integrity of the measurement process itself within the software integrity protection framework proposed in this work, the measured integrity information must be exposed so as to be accessible to an external entity. This distribution of functionality facilitates isolation, restraint of function, standardization and diversity of real-world production, all of which are necessary to a well-adopted, efficient, self-secure general software integrity protection architecture.

In these experiments, the exposure of integrity information is carried out via the addition of source code instructions which perform *initializing assignments with marking identifier values*. The goal of such assignments is to expose to the integrity evaluating entity the location of integrity information (stack memory offset), integrity measurements (e.g. the memory length of a parameter), or meta-information about instrumented target program code which facilitates protection (e.g. to signal to the evaluation layer the occurrence of a split call). The power of these marking, initializing

assignments for integrity exposure lies in their ability to cause compilers to emit binary instructions which reveal data and information which can be linked to unique memory-occupying source code-level constructs.

These assignments are transformed during compilation into binary instructions which *write controllable values to working memory*, thus enabling recognition by DynamoRIO or other process virtual machine (PVM) level entities which have full access to instructions during run-time. Marker values can be assigned by application type or defined by the integrity evaluating entity (or even arbitrary "magic" numbers), but must serve to identify key memory locations or expose integrity information.

A key observation of the exposure process demonstrated here is that *the exposure instructions constitute an in-memory signaling and marking communication channel to engines with instruction analysis capability: the components of instructions can signal events, mark instructions for analysis purposes, or transfer integrity information.* This capability enables the relocation of integrity information to a remote, isolated evaluator, demonstrating a critical required process for the construction of a systematic software integrity protection framework.

A most critical example of the use of an initializing assignment for exposure here is the application of an instrumented integer assignment with a special marker value which identifies a split call (recall that split calls were used to arrange working memory in a manner advantageous to call integrity measurement) to DynamoRIO. Since DynamoRIO has complete access to the operands of instructions at run-time, the write of a recognized marker value to stack memory is clearly visible, allows the trigger of appropriate instructions for action, and also exposes to some degree the location at which the value was written.

Note that although we successfully expose integrity information to DynamoRIO here for evaluation against integrity specifications, it is also possible to statically instrument binary executables directly for the purpose of exposing integrity information to a yet more distant evaluator entity. However, such a process is not investigated in this work. The richness of the integrity information available via processing of source code means that source code is given priority as a point of exposure in this work.

### 6.10.5 Evaluation of Integrity Information

The final process required for demonstration of call integrity protection is the evaluation of integrity information with respect to a set of dynamic call integrity conditions. Appropriate enforcement of policy can be carried out based on the result of the evaluation. In terms of dynamic call integrity, the applicable specifications of conditions were

1. The relationships between the lengths of parameters in memory must comply with a set of integrity conditions. A useful default example condition is "no length may exceed another"
2. The number of format string specifiers must match the number of non-format string parameters

As mentioned during the discussion of format function dynamic integrity measurement, the evaluation was carried out via the insertion of instructions to implement simple comparisons to check compliance with the integrity conditions above. In C source code, this is achievable with conditional code such as *if*.

Evaluation was also demonstrated at the level of the binary instrumentation, which is carried out by DynamoRIO. This is done via standard exposure techniques detailed above, leaving DynamoRIO to recognize the measurements during run-time, apply comparisons, and enforce the applicable policy. The experiments carried out here involved a process of determining the ideal work-sharing relationship between the source code instrumentation and binary manipulation processes for evaluation of dynamic integrity in a manner which complies with the structural requirements of the software integrity framework defined in this work. Since crucial call integrity information has been shown here to require access to source code, applying evaluation as part of source instrumentation is in general acceptable and minimizes performance impact. However, ultimately the evaluation of must be passed to a standardized, isolated, functionally constrained engine in keeping with the general integrity protection framework revealed in this work.

Process virtual machine software entities such as DynamoRIO illustrate the type of component (and level of access) which is necessary to analyze and evaluate integrity information exposed by other techniques (namely, assisted source code instrumentation), and also demonstrate the validity of the relocate-ability of integrity information. While on-platform DynamoRIO binary instrumentation serves this role in our demonstration, the role is ultimately to be filled by the external hardware evaluating engine defined in our protection framework.

### 6.10.6 Conclusion

By employing a novel analysis of the integrity of running computer systems, this work has identified the launch of procedure calls during software execution as a key instant at which integrity can be measured and, if necessary, protective action taken. Integrity is evaluated with respect to a specification of conditions for the integrity of the types of procedure calls expected.

A demonstration scheme was implemented using both source code and binary instrumentation with a software-based instruction introspection engine, illustrating that the integrity of running procedure calls can be measured, exposed, and analyzed externally to the protected program execution to detect meaningful violations. Finally, although the number of instantiations of run-time procedure calls is large, specifications for the protection of *categories* of procedure calls can be created with the help of developer-assistive tools.

There are innumerable examples of the exploitation of vulnerable procedure calls in records of contemporary run-time attacks. A fundamental reason for this is the role of procedure calls as key interfaces to programs, importantly between software scopes driven by separate entities. The ability of an attacker to provide crafted or perturbed input to programs being defended is an unavoidable consequence of software production practices adopted to enhance the practicality of the construction of complex software: the ability to transfer functionality to a modular, callable procedure provides an interface to accept input, enhances abstraction and reuse, and eases the implementation of logic which may be arbitrarily complex.

To address the inevitable, continual profusion of vulnerable software, a general protection strategy is needed, rather than a patchwork of individual techniques. Run-time call integrity protection allows for the inclusion of specifications for a broad number of general attacks or application-dependent violations. Further, this protection prevents some violations purely by interruption of *sequences* of violations at a critical step.

Protecting the integrity of procedure calls is a key step in addressing realistic attacks in a general manner because procedure calls are one type of integrity-relevant interface through which a complex of memory interactions can be directed by non-author entities despite an assumed static integrity of programs. Ultimately, running programs can be revealed as large scale streams of memory state interactions, some of which may violate the explicit intent of the program creator when execution is subject to attack. It is the role of procedure calls as imprecise directing agents

of sets of such memory state interactions that makes them a valuable point at which to implement integrity protection.

# CHAPTER 7

# A PROTOTYPE CALL INTEGRITY ENGINE

In order to demonstrate the validity of the measurement, exposure and protection of program dynamic call integrity, a set of experiments with software instrumentation were performed and a prototype created for the measurement, exposure and evaluation of call integrity (for the full instrumentation code see Appendix B). The implementation of the prototype is detailed here.

## 7.1 Setup

The prototype was implemented on Linux with C source code, using ELF (executable and linkable format) binaries and the x86 instruction set. Source code instrumentation was implemented with a script which inserts the needed preamble of integrity instructions per call. DynamoRIO was used to implement run-time binary instrumentation by creating an instrumentation client [34]. The prototype was run on a laptop with an Intel Core i5-2520m processor (dual-core, 2.5Ghz, 3MB cache).

## 7.2 Implementation

The evaluation code can be adapted to other environments which use a working memory stack and registers, and additionally allow software instrumentation and run-time instruction analysis. The goal and scope of evaluation here is to demonstrate the principles and techniques detailed in Chapter 6: that the dynamic integrity of a procedure call can be measured and integrity violations detected by combining several forms of instrumentation with instruction introspection without requiring the original protected program to be modified by the developer.

Rewriting binary executable files as in [49], though feasible for measuring some aspects of program behavior, is not attempted here for integrity purposes due to a lack of access to integrity-relevant information held within source code. One fundamental requirement of the external evaluation of program dynamic integrity is the access to information necessary to map memory-using objects in source code to true memory structures linked to execution.

As has been mentioned, compilers are another option to implement integrity measurement and exposure. However, this work does not attempt to modify existing compilers for several reasons.

Compiler software shares system resources including (critically) memory with programs to be protected, a violation of the requirements of the protection framework introduced in this work. Such association exposes a compiler-based integrity measurement functionality to attack, and even extended threats such as Advanced Persistent Threat [26].

Further, adding integrity functionality to existing compilers does not serve functional constraint, due to the intentional mingling of distinct functionalities in the design of the compiler program. The use of a collective suite of software instrumentation for the capture and exposure of integrity information is a relative functional constraint owing to the unidirectionally oblivious and minimal addition of an efficient set of integrity analysis instructions.

Source instrumentation was performed automatically using a script which processes C source code calls, adding a *preamble* of instrumentation instructions (source code and assembly code) prior to all procedure calls residing within a range of code specified by a special start and end tag comment.

The Perl instrumentation script (Listing B.1 in Appendix B) implements a one pass instrumentation of source code, inserting either source or inlined assembly instructions as needed along with all necessary declarations in logically named separate header files.

```
#CALL. pre: all vars in vardecllist or global vardecllist
elsif ($_ =~ /(\S+)\((((\S|\s)+,|\S+)+)\);/){ #exclude void calls
      my @callArgs = split ',',$2;
      my $argCount = $#callArgs+1;
      my $isPointer = 0; my $varType = "";
      my $isConstant = 0;
```

Listing 7.1: Instrumentation Script Regular Expression

The instrumentation script leverages regular expressions to recognize the key source code constructs which are the focus of integrity instrumentation: calls (an example is shown in Listing 7.1). Since parameter parity is a critical dimension of dynamic integrity, format function calls are recognized as a call subset of interest, and given special analysis as detailed in this Chapter.

Call parameter counting and source-visible literal format string parsing are implemented based on regular expressions on the structure of calls. This script enables the detection of C pointers,

which may force integrity analysis to be delayed to execution where the illuminated memory locations can be enumerated. Finally, the automated scripting scheme employs an indexing scheme with the encountered call index, preamble split call (parameter) index, and integrity measurement type in order to ensure unique labeling of all declared symbols defined for the purpose of integrity protection.

Whereas the purpose of the prototype is the evaluation of the effectiveness and feasibility of the measurement and exposure of call integrity information to external evaluators, generalized analysis of C code (or any other programming language) is outside the scope of this prototype and not attempted. The extension of call integrity protection to other software environments is a task left to work in those domains.

```
int bytesum00 = strSum(mystring);
long int result_00arglen = 1414416707 + 1;
asm volatile ("mov %0, %%edx;"
:: "r" (bytesum00)
);
func_00(mystring);
PostCallIntASMarglen00

int bytesum01 = strSum(garr);
long int result_01arglen = 1414416707 + 1;
asm volatile ("mov %0, %%edx;"
:: "r" (bytesum01)
);
func_01(garr);
PostCallIntASMarglen01

func_02(a);
func(mystring,garr,a,2,3.0); //original call
```

Listing 7.2: Sample Instrumentation Preamble Code Segment

Together with binary instrumentation, the inserted instructions implement the exposure and measurement of call integrity at run-time. The per-call inserted preamble (see Figure 6.4 and Listing 7.2) includes a set of $k$ split calls where $k$ is the number of parameters whose integrity must be analyzed within the source code call. Constant parameters are not analyzed, since their participation in memory interactions is entirely visible in source code. Each split call is preceded and succeeded by a set of instrumented integrity measurement instructions.

```
int strSum(char *str){
int sum = 0,i=0;

while (str[i] != '\0'){
    sum = sum + (int)str[i++];
    }
return sum;
}
```

Listing 7.3: Instrumentation Header Code: Terminating Condition Generation

There is an instrumented function call inserted to calculate a terminating condition value for enumeration of parameter bytes in memory. The condition chosen is the integer sum of the bytes of the parameter whose integrity is being analyzed in memory. This is implemented by the insertion of a simple function for the calculation of the byte sum, the header code of which is shown in Listing 7.3. Next, a short sequence of inlined assembly instructions is inserted to copy this terminator value to a register for the byte enumerating operations to come. This short sequence is visible in Listing 7.2 lines 3-5.

Each split call is further marked with a special variable allocation to present a marker to the instruction introspection layer (DynamoRIO) indicating that this is a split call rather than an existing call instruction. This statement is a marking, initializing assignment instruction which identifies a memory value and location to external integrity evaluation entities with memory access. The marking, initializing assignment statement is visible in Listing 7.2 line 2.

The split call is a nominal source code call which accepts the parameter being measured, yet implements no action (see line 6 in Listing 7.2). The split call technique causes the compiler to impose order on the layout of working memory in order to locate each parameter, enabling measurement of the integrity primitives detailed below. The split call causes the memory for the parameter to be arranged at the end of the working memory stack during run-time, while leaving the logic of the surrounding program code unchanged. The necessary definitions for all inserted instructions are inserted in newly created header files during instrumentation by the automated script (see Section B.6 and B.7).

Directly succeeding each split call a special assembly language instruction sequence is inserted which implements the particular integrity primitive measurement: either parameter length or format function call parity. An example of one such sequence is visible in line 7 in Listing 7.2 as a

type-defined label. Each type of sequence will be fully analyzed in the sections below. Both measurements require the ability to precisely enumerate the *actual bytes used by the run-time parameter in working memory.*

Two critical call integrity primitive measurements are implemented: parameter memory length and parameter parity (or presence). Along with others produced from data (e.g. structure), these dynamic call integrity primitives can be used to construct arbitrary dynamic integrity conditions which are to be checked by the evaluator engine. Dynamic integrity conditions are statements which impose requirements on the relationships between integrity primitive measurements imposed by instructions and procedure calls.

Parameter memory length and presence are critical integrity primitives measured by this prototype. The techniques implemented to measure and expose these are detailed in the sections below. However, there are additional dynamic integrity primitive properties inherent in memory use (not exhaustive):

1. length in memory
2. presence or parity
3. structure
4. sequence
5. linkage (e.g. control flow)

As shown in Figure 7.1, dynamic integrity primitives are aspects of distinct memory objects which exhibit integrity when they are caused to interact by software instructions or other memory interactions (e.g. control flow) via a transfer of requirements. The methods implemented here measure these properties for the purpose of the construction of arbitrarily complex dynamic integrity conditions. These conditions are the basis for check instructions executed by integrity evaluation entities such as the component proposed in the integrity protection framework defined in this work.

Structure is an integrity primitive by virtue of the imposition of requirements on the layout of memory region contents by instructions or instruction sequences. Structure is additionally an abstract primitive. Structural requirements imposed on data are dependent on the particular process using the data, and typically application-dependent.

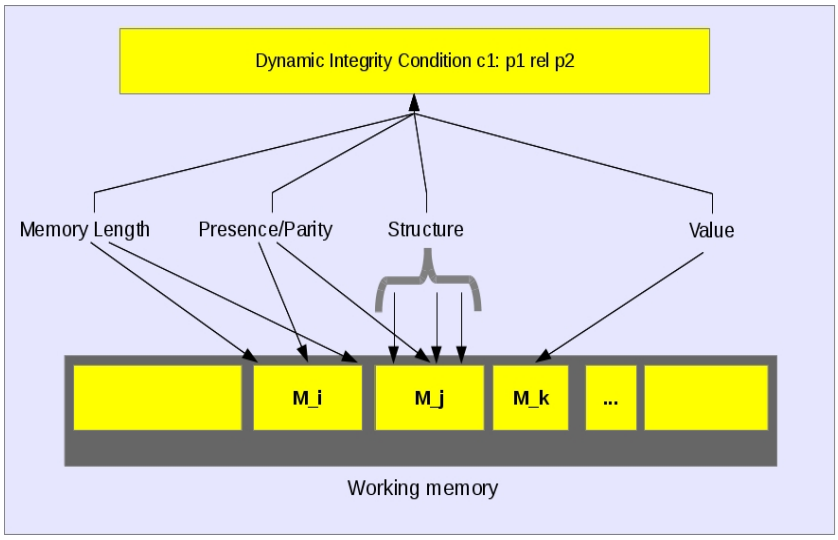


Figure 7.1: Dynamic Integrity Primitives for Condition Creation

The value inherent in the contents of a memory location is subject to interpretation for use, and thus a usable integrity primitive within dynamic integrity conditions after interpretation with straightforward checks on the value.

## 7.3 Parameter Length Measurement

For measurement of call parameter byte length, instrumentation instructions are inserted which count the running sum of the bytes encountered in the parameter's contiguous, working memory on the stack, stopping at a pre-produced terminal condition.

The process proceeds firstly by inserting assembly instructions (see Listing 7.4) to change the working index memory address to the address which is held in the location pointed to by the address in the esp (stack pointer) register, which means moving away from the end of the stack in process memory to the precise location of the parameter starting word. Recall from Listing 7.2 that the terminating condition for all byte enumerating operations has been stored in the edx register.

The extra jump may be necessary because the source code instructions of procedure calls accepting pointer type parameters typically are compiled into sequences such that the address of the pointer parameter is placed at the end of the stack (the address of which is held in the stack pointer register) rather than the actual (possibly multi-byte) parameter.

```
#define PostCallIntASMarglen00 asm volatile ("movl $0, %%ecx;"   \
"movl (%%esp), %%esi; "   \
"mov $0, %%ebx; "   \
"mov $0, %%edi; "   \
"add $1, %%ebx; "   \
"START00a: nop; "   \
"movl (%%esi), %%eax; "   \
```

Listing 7.4: Instrumentation Header Code Segment 1: Adjust Index to Parameter Memory

Inlined assembly instruction 2 in Listing 7.4 dereferences the stack pointer register, copying the four byte value held at the working memory location specified by the address in that register. The value is the actual memory address of the parameter in working memory. It is transferred to another register for use in the analysis of the bytes of the first word of the parameter memory. The final instruction shows how the parameter address is used to index the stack memory to copy the first word of the parameter for analysis.

Recall that esi was loaded with the parameter address in Listing 7.4. The first loop termination comparison is implemented by moving the low (first) byte of the parameter memory (held in the location addressed by the esi register) to the low byte of ecx, then copying the result to a free, zeroed register (edi). The parameter byte in edi is then compared to the prepared parameter integer byte sum in edx (see Listing 7.5). If the sum exceeds the value in edx (the running bytesum exceeds the pre-calculated sum, the terminating condition), execution jumps to the end of the sequence for wrap up.

```
"START00a: nop; "   \
"movl (%%esi), %%eax; "   \
"mov %%al, %%cl; "   \
"add %%ecx, %%edi; "   \
"cmp %%edx, %%edi; "   \
"jge DONE00a; "   \
```

Listing 7.5: Instrumentation Header Code Segment 1: Loop Termination Comparison

Throughout the loop assembly code, the edi register will hold the running count of bytes encountered. The ebx register is used as the counter for the number of bytes encountered, and will hold the result, the precise byte length of the parameter used in memory.

Three additional comparison blocks are implemented in order to ensure that each particular byte (out of four within each memory word encountered) of the parameter is encountered and added to the running sum. The remaining comparison blocks require shift and mask operations in order to access the byte of interest. Since only the mask operation and the shift differ among the blocks, one block will be analyzed in detail here (see Listing 7.6).

```
"jge DONE00a; "   \
"add $1, %%ebx; "   \
"movl $0, %%ecx; "   \
"mov %%ax, %%cx; "   \
"shr $8, %%ecx; "   \
"add %%ecx, %%edi; "   \
"cmp %%edx, %%edi; "   \
"jge DONE00a; "   \
```

Listing 7.6: Instrumentation Header Code Segment: Loop Byte Comparison Block

Accessing the second byte of the current word being analyzed is implemented by moving both low bytes of the parameter (held in eax) into the low two bytes of ecx, and adding a shift to the right of eight bits (see instruction 4-5 of Listing 7.6). Afterward, the value is moved to edi as usual and compared with the terminating bytesum. The third loop comparison block is implemented with a bit mask operation carried out by a logical *and* in order to ensure that the third byte is isolated (see Listing 7.7). Accessing the final, highest byte of the current word in memory requires only to shift the copied value to the right by 24 bits.

```
"jge DONE00a; "   \
"movl $0, %%ecx; "   \
"add $1, %%ebx; "   \
"mov %%eax, %%ecx; "   \
"and $16777215, %%ecx; "   \
"shr $16, %%ecx; "   \
"add %%ecx, %%edi; "   \
"cmp %%edx, %%edi; "   \
"jge DONE00a; "   \
```

Listing 7.7: Instrumentation Header Code Segment: Third Byte Extraction

The terminal condition for the byte-wise count of the parameter memory during run-time is the reaching of a loop iteration at which the running integer sum of bytes encountered exceeds

the pre-calculated sum of the parameter bytes (Figure 7.3) which was produced and placed into a register. The final instructions in Listing 7.8 illustrate the storing of the precise byte length measurement in a C integer for use and exposure.

```
"jge DONE00a;"   \
"add $1, %%ebx;"   \
"add $4, %%esi;"   \
"jmp START00a;"   \
"DONE00a: nop;"   \
"mov %%ebx, %0;"   \
: "=b" (result_00arglen) \
);
```

Listing 7.8: Instrumentation Header Code Segment: Storage of Parameter Byte Length

Typically, the number of additional instructions inserted by this method of parameter length counting is

$$16 + 46(argcount)$$

where *argcount* is the number of arguments whose length is measured in memory.

As shown in Table 7.1, the average relative slowdown per call for parameter byte length measurement is approximately 4 per call, or .19 per byte measured. The memory space requirement per argument for run-time argument length measurement includes space for two integers (for storage of a bytesum and the count result) and the use of the general purpose x86 registers.

## 7.4 Procedure Call Parameter Parity

The relationships between the presence properties of memory objects included in interactions imposed during software execution is an additional critical dimension across which transfers of requirements for compliance act. There are cases where the ongoing compliance of execution with creator intent requires that the presence of one parameter is matched by the presence of another, possibly as the corresponding memory area used in a conversion process. In fact, a major category of attacks based on format functions exploits imbalances of this presence dimension in memory interactions. For the measurement of dynamic call integrity related to parameter counts and presence, source code instrumentation was applied in this work for the analysis of the parity

primitive property of call parameters including in particular format strings which impose presence requirements on the remaining parameters.

Source code procedure calls typically expose the number of parameters to a static analysis. However, analyzing the presence/parity of call parameters is hindered by one obstacle: there may exist a single parameter which imposes a presence requirement on the remaining parameters and may further not be visible to static source code analysis (but visible in working memory during execution), thus "hiding" the requirement imposed.

```
#include "formatRT.c.dynamic.in.instr.c-cint.h"
//formatRT.c  demo format string parsing in memory, INT exposure
//

#include <stdio.h>
#include <string.h>
#define MAGICINT 1414416707

//CINTBEGIN

//global array
char garr[10] = "bbbbbbb";

int func(char *str,char *str2,int aa, int ab, int ac){
int b;
return 0;
}

int main(int argc, char argv[]){
char mystring[10] = "%aaaa%aaa";
char fmtstring[20];
int a = 1;

gets(fmtstring);
```

Listing 7.9: Code Segment: Format Function Call

A salient example of such parameters is the format string used in format functions provided in the C programming language. These format string parameters contain conversion character sequences which require corresponding actual parameter memory objects to be present in the procedure call, and implement conversion between primitive data and character types. Thus, the calling of conversion functions demonstrates the transfer of dynamic integrity requirements across the parity dimension. In order to protect dynamic call integrity, parameter presence relationships must be measured in a general manner during the execution of procedure calls.

The instrumentation process for parameter parity is similar to that which was implemented for parameter length analysis. An instrumented header file is created along with the newly instru-

mented code file for all required definitions. The difference lies in the distinct assembly sequence inserted after split calls. An example C code file instrumented for parameter parity measurement excluding the instrumentation preamble is shown in Listing 7.9. The instrumented preamble and original call are shown in Listing 7.10.

```
int bytesum20 = strSum(fmtstring);
long int result_20fmtstring = 1414416707 + 1;
asm volatile ("mov %0, %%edx;"
:: "r" (bytesum20)
);
printf_20(fmtstring);
PostCallIntASMfmtstring20

printf_21(a);

int bytesum22 = strSum(mystring);
long int result_22fmtstring = 1414416707 + 1;
asm volatile ("mov %0, %%edx;"
:: "r" (bytesum22)
);
printf_22(mystring);
PostCallIntASMfmtstring22

if (2 != result_20fmtstring){printf("format string violation:count mismatch!!");}

printf(fmtstring,a,mystring);
```

Listing 7.10: Instrumented Code Segment: Preamble

In the instrumentation process, the normal parameters visible in source code are counted, and the format string is the subject of further analysis to determine the presence requirement imposed on the remaining parameters. The format string in C format functions such as print or scanf is typically the first parameter, but such positioning can be considered a call integrity engine parameter (among others) that can be tuned to language, context or even by function group.

Since format strings are typically of pointer type, the string may include arbitrary string content. Further, C format strings as pointer types are permitted to be set at run-time, exposing them to the influence of adversaries.

If the format string is visible in source code (e.g. a literal string), it is parsed during source code instrumentation to determine the number of conversion sequences and hence the imposed presence requirement. As mentioned, the number of non-format string parameters is extracted from source code analysis during the instrumentation pass.

If the format string is known only at run-time, the methods for isolation and bounding of parameters used for argument length measurement are reapplied here to allow the format string

to be analyzed in working memory. Inlined assembly instructions are inserted to parse the format string in memory *during execution* to determine the number of format conversion sequences, each of which are identified by a '%' in C.

The approach to implement parsing of the run-time bound format string is similar to that of parameter length measurement, yet the goal in this case is to parse the string for conversion characters (see Listing 7.11). As before, an inlined assembly code sequence is employed to enumerate each parameter memory word (four bytes) to analyze each byte during execution (the full sequence is shown in the header file in Listing B.7 in the Appendix).

A comparison block within the assembly sequence is illustrated in Listing 7.11. The integer numeric value 37 for the percent character is used in the comparison of each byte. If the byte being analyzed matches the percent character, a conversion sequence is detected and a counter incremented in the ebx register.

Loop termination is based on the integer bytesum condition calculated prior to the split call, as with parameter length measurement. If the running sum of the bytes encountered exceeds the byte sum placed in the edx register prior to the execution of the parsing assembly sequence, execution jumps to the end of the sequence to store the parsed count to the exposure memory object implemented with the exposure instruction detailed in Section 7.6.

```
#define PostCallIntASMfmtstring20 asm volatile ("movl $0, %%ecx; "  \
"movl (%%esp), %%esi;"  \
"mov $0, %%ebx;"  \
"mov $0, %%edi;"  \
"START20f: nop;"  \
"movl (%%esi), %%eax;"  \
"mov %%al, %%cl;"  \
"add %%ecx, %%edi;"  \
"cmpl $37, %%ecx;"  \
"jne NOPCT120f;"  \
"add $1, %%ebx;"  \
"NOPCT120f: nop;"  \
```

Listing 7.11: Header File Assembly Code Segment: Format String Parse

In order to produce parity measurement, the format string imposed count is compared with the number of non-format string parameters by inserting source code instructions for a relational comparison. As shown in Listing 7.15, the number of statically determined normal parameters will be compared during execution with the number of parameters required by the format string. The result of this comparison is a value indicative of the dynamic integrity of conversion type

Table 7.1: C Source Code Call Integrity Measurement Performance

| Integrity Performance Measurements | | Argument Run-time Length Measurement | Argument Count + Format String Count Extraction (2 specifier, 2 arguments) | Argument Count + Run-Time Format String Count Extraction (2 specifier, 2 arguments) |
|---|---|---|---|---|
| Source Instrumentation | Avg. Relative slowdown | 4.1812, 0.19/byte | 0.9984,0.3328042027 per argument | 1.05866696, 0.3528889878 per argument |
| | Absolute Additional Time (ms) | 0.00015558, 0.00000707/byte | Negligible | Negligible |

function calls during execution with respect to the presence dimension. The format string category of attacks can be detected with the technique implemented here, as well as any attacks which depend on the imposition of presence disparities upon memory interactions which require parity in order for memory effects to comply with the program author intent.

The combination of format string parsing and argument count measurement resulted in a small performance impact per call as shown in Table 7.1. The average relative slowdown for measurement was timed as approximately 1, or 0.33 per argument. Measurement of relative performance impact of instrumentation on call execution was implemented with inlined assembly instructions to extract cycle offset using the time stamp counter processor feature.

The memory space requirement per argument for parameter count and run-time-bound format string extraction includes space for two integers (a bytesum for run-time format string parse termination and the format string specifier count result) and the use of the general purpose x86 registers.

Note that in case the format string is available in source code, no storage is necessary since the source code instrumentation can parse the format string and count the non-format parameters directly.

## 7.5 A Heuristic For Binary Level Call Parameter Count Determination

One goal of the experiments implemented here is to demonstrate the degree to which dynamic call integrity can be relocated to external engines. To that end, a technique for determining the number of parameters passed to procedure calls during execution without source code access is detailed here.

The run-time instruction manipulation library DynamoRIO is used to implement the heuristic [34]. DynamoRIO provides run-time access to the components of instructions (e.g. operand, op-code) as well as a set of execution events to which one can attach analysis code via the creation of a client program. The client program has access to events such as signal reception, thread exit, and system call execution.

In order to demonstrate the heuristic, the event signaling a basic block arrival is used as an attachment point to begin analysis code. When a basic block arrives for processing by DynamoRIO, we impose a check for procedure call instructions (which will necessarily end basic blocks, due to restrictions of the library used). If the block ending instruction is a call, we are able to analyze and transform the call instructions and all previous ones within the block.

```
//detect mov or fstp to base/disp at x(esp)
if (opnd_is_base_disp(dstop) &&
                ((reg = opnd_get_base(dstop)) == DR_REG_ESP)
                   ){

  int movdisp = opnd_get_disp(dstop);
  if (movdisp == (lastESPoff)){
     argcount += 1; //this is likely a call site proc. arg
        write
     lastESPoff += 4;
     }
  else
     {break;} //found noncontig. mem. mov->likely not an arg
  }
```

Listing 7.12: DynamoRIO Client Code Segment: Parameter Count Heuristic

To implement the heuristic, the block instructions are scanned from the terminal call in a reverse order, analyzing the operands to detect instructions which write (e.g. mov in x86) to contiguous locations relative to the stack pointer address (base + displacement). If the location written to is shown to be contiguous to one previously written to during the scan, the write is interpreted as a parameter write and the parameter counter is incremented. Note that for pointer-type arguments, the applicable pre-call writing instructions sought during scanning write the address of the argument (located elsewhere within the process memory) rather than the argument bytes. The heuristic

implementation logic is shown in Listing 7.12, while the full code listing is in Section B.5 in the Appendix.

The performance impact of the parameter count heuristic is driven largely by the impact of DynamoRIO, since the instructions to implement the heuristic include only a small number for scanning the basic block and analysis of instruction operands. DynamoRIO provides binary instrumentation capability at overheads between zero and thirty memory and time overhead via heavy optimizations based on instruction trace caches, repointing of branches to cache, etc [34]. However, near-native peformance is common with DynamoRIO instrumentation, as is observed in this case.

## 7.6 Exposure of Integrity Measurements to Evaluation

In order to serve the unified integrity protection framework in this work, integrity information must be exposed to an external engine for isolation and independence. In our experiments, the exposure of integrity information is carried out via the addition of source code instructions which perform *initializing assignments with marking identifier values*. An example of the exposure instruction is shown in the second instruction in Listing 7.2. This instruction provides storage for the primitive dynamic integrity measurement to be produced by the post-split call assembly sequence, but also exposes to our evaluation layer (DynamoRIO) both the memory location of the storage and the fact that a marked type of measurement is present (signalled by the marker value).

The power of these marking, initializing assignments for integrity exposure lies in their ability to cause compilers to emit binary instructions which reveal data and information which can be linked to unique source code-level constructs such as statements or variables. Typically, the initialization results in memory move instructions such as *mov* which copy marking bytes to a memory location indicated by a base+address offset or other decipherable location. The initialization value is a channel usable by implementors of integrity exposure to pass information to the integrity evaluation engine. These instructions are fully accessible to DynamoRIO during run-time prior to being dispatched to the processor for execution. DynamoRIO allows detailed analysis of each instruction operand, whether base+address memory references, registers, etc.

As depicted in Figure 7.2, exposure instructions reveal memory locations and integrity measurements to DynamoRIO during execution. Note that the exposure instruction may need to be paired with a separate exposure instruction to first reveal the location of the measurement storage

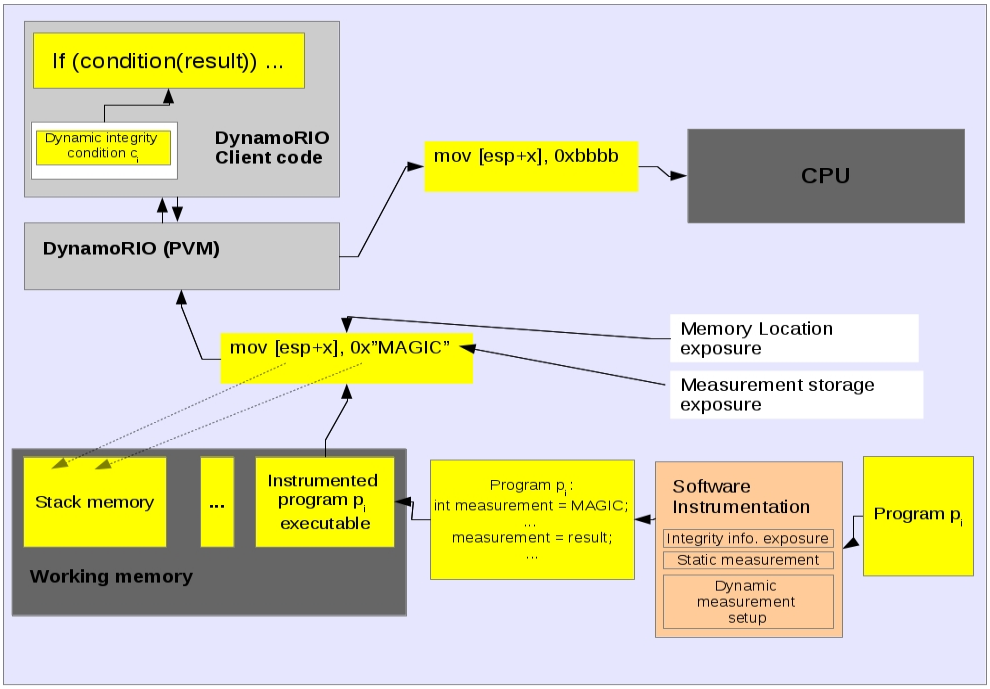


Figure 7.2: The Exposure of Integrity Measurements to DynamoRIO

to the evaluator, depending on the marking value scheme used and the type of integrity measurements taken. As a call integrity evaluation entity, the DynamoRIO process virtual machine (PVM) is extended by the creation of client program which implements integrity measurement and evaluation.

## 7.7 Evaluation of Integrity Primitives Against Conditions

Demonstration of Integrity specification evaluation was carried out via the insertion of instructions to implement simple comparisons to check compliance with the integrity conditions above (see Listing 7.15). In C source code, this is achievable with conditional code such as *if*.

The computational overhead of dynamic integrity specification evaluation demonstrated here is related to the number of conditions included in the specification and their complexity. Despite extensive use of basic block and trace caching to improve performance, the use of DynamoRIO for run-time program introspection and instrumentation introduces non-trivial time and memory overhead, specifically from zero to thirty percent including variance across applications [34].

Enforcement of a response to integrity violations is inherently possible in the evaluation methods demonstrated in this work. However, enforcement is intended to be discharged by an engine isolated from the protected software context (i.e. with respect to memory, control flow).

Nevertheless, the conditional statements described earlier in section 7.4 implement source-based enforcement as part of this prototype for demonstration purposes.

In DynamoRIO, remote enforcement is demonstrated by using a technique of nullification of procedure calls. If an integrity violation was detected as shown in Listing 7.13 and 7.14, the appropriate application or domain determined response can be issued (e.g. halt of operation, remediation, etc.). Violations are exposed to DynamoRIO using the exposure instructions detailed in Section 7.6.

The nullification of procedure calls is implemented by replacing the block-ending call instruction with a sequence of executable instructions which result in no meaningful control flow. In particular, the replacing sequence is an inert instruction, an instruction to set a register value, followed by an instruction to branch based on checking the value in the register (see Listing 7.16). Due to the apriori setting of the register value, the branch will never succeed, and thus the sequence replaces the original call instruction with an inert set of instructions, nullifying the call if such a response is justified by a violation.

The use of DynamoRIO in this prototype demonstrates an important property of the external dynamic integrity evaluation device proposed in this work: interception of integrity information for evaluation must occur without delay. DynamoRIO intercepts executable instructions just prior to their execution by the processor, leaving little opportunity for the prospect of attacks on integrity information exposed by instrumentation. Unnecessary delay of the interception of integrity information may enable vectors for side-channel time of check to time of use (TOCTTOU) attacks.

Ultimately, evaluation is to be left to an external hardware engine meeting certain requirements in the framework detailed in this work. In Section 7.6 the dynamic integrity information is shown to be readily exposed to engines having at least access similar to a process virtual machine entity such as DynamoRIO. Strictly speaking, DynamoRIO cannot evaluate integrity primitive information exactly as the framework monitoring device does due to its lack of access to data (stack) memory. Hence, evaluation was readily demonstrated by source code insertion of evaluation code. Nevertheless, the introspection capability of DynamoRIO is required and demonstrates the use of

the signaling channel populated by source instrumentatio. This prototype enables simulation of pure external evaluation by providing the capability to insert executable instructions for evaluation, although normally the evaluator would simply carry out an expression on integrity primitives in stack memory. Thus, measurement, exposure, and external analysis of integrity information are implemented by this prototype, demonstrating the effectiveness of unified software integrity protection.

This prototype demonstrates the realization of the software integrity model revealed in this work, capturing creator intent and extending it to the memory-interacting executing instruction sequences which were inexactly employed to carry out the programmatic goal.

Software memory interactions with respect to control flow and data influence can be analyzed robustly with hardware measures. Call integrity protection was implemented here by the efficient evaluation of conditions on transfers of compliance requirements during call memory interactions, solidifying comprehensive protection of software memory interactions. The constraining of software memory interactions to creator intent is the remaining aspect of software integrity protection which is unaddressed by existing work, and is the basis for future measures to fully protect software integrity. For the full prototype code the reader is referred to Appendix B.

```
if ((instrptr == instrlist_last(bb))/* && (instrptr != instrlist_first(bb))*/){

 ptr_int_t constant = 0;
 instr_t *iter2 = instr_get_prev(instrptr);
 //reverse scan for splitcall detection, bytecount var loc
 while (iter2 != NULL){

  if (instr_is_mov_constant(iter2,&constant)){
     //marker stmt for split call
     if ((constant > MAGICINT) && (constant <= (MAGICINT+100))){
```

Listing 7.13: DynamoRIO Client Code Segment: Violation Detection Part 1

```
     //Integrity violation code
     else if ((constant < (MAGICINT-100)) && (constant >= (MAGICINT-120))){
        //dr_printf("found precall INT violation!\n");
        violation = true;
        //dr_print_instr(drcontext,our_stdout,instrptr,"#####");
        }
     else
```

Listing 7.14: DynamoRIO Client Code Segment: Violation Detection Part 2

```
int bytesum20 = strSum(fmtstring);
long int result_20fmtstring = 1414416707 + 1;
asm volatile ("mov %0, %%edx;"
:: "r" (bytesum20)
);
printf_20(fmtstring);
PostCallIntASMfmtstring20

printf_21(a);

int bytesum22 = strSum(mystring);
long int result_22fmtstring = 1414416707 + 1;
asm volatile ("mov %0, %%edx;"
:: "r" (bytesum22)
);
printf_22(mystring);
PostCallIntASMfmtstring22

if (2 != result_20fmtstring){printf("format string violation:count mismatch!!");}

printf(fmtstring,a,mystring);
```

Listing 7.15: Instrumented Format Function Call Code Segment: Evaluation

```
int nullifyCall(void *drcontext, instrlist_t *bb,instr_t *calltodump){
        //catch and replace call instr//
        opnd_t regecx = opnd_create_reg(DR_REG_ECX);
        instr_t* newnop = INSTR_CREATE_nop(drcontext);
        instr_t* newjecxz = INSTR_CREATE_jecxz(drcontext,opnd_create_instr(newnop));
        instr_t* newmov = INSTR_CREATE_mov_imm(drcontext,regecx,opnd_create_immed_int(2,
            OPSZ_4));

        instr_set_translation(newjecxz,instr_get_app_pc(calltodump));
        instr_set_translation(newmov,instr_get_app_pc(newjecxz));
        instr_set_translation(newnop,instr_get_app_pc(newmov));

        instrlist_preinsert(bb,calltodump,newnop);
        instrlist_preinsert(bb,calltodump,newmov);
        instrlist_preinsert(bb,calltodump,newjecxz);
        instrlist_remove(bb,calltodump); //dump the call instr
return 0;
}
```

Listing 7.16: DynamoRIO Client Code Segment: Call Nullification

# APPENDIX A

# PUBLICATIONS

Jonathan Jenkins and Mike Burmester. Runtime Integrity for Cyber-Physical Infrastructures, Ninth Annual IFIP Working Group 11.10 International Conference on Critical Infrastructure Protection, Accepted, 2015.

Jonathan Jenkins, Sean Easton, David Guidry, Mike Burmester, Xiuwen Liu, Xin Yuan, Joshua Lawrence, and, Sereyvathana Ty. Trusted Group Key Management For Real-Time Critical Infrastructure Protection. Thirty-Second Annual IEEE Military Communications Conference (MILCOM 2013).

Jonathan Jenkins, Mike Burmester. Protecting infrastructure assets from real-time and runtime threats. Chapter 6, Critical Infrastructure Protection VII, 2013, Springer.

Mike Burmester, Joshua Lawrence, David Guidry, Sean Easton, Sereyvathana Ty,Xiuwen Liu, Xin Yuan, and Jonathan Jenkins. Towards a Secure Electricity Grid. 2013 IEEE Eighth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)

David Guidry, Mike Burmester, Xiuwen Liu, Jonathan Jenkins, Sean Easton, and Xin Yuan. A Trusted Computing Architecture for Secure Substation Automation. In Proceedings of the Seventh International Conference on Critical Information Infrastructures Security (CRITIS12), Lillehammer, Norway

R. Motta, W. Nienaber and J. Jenkins: Gnutella: integrating performance and security in fully decentralized P2P models. The Forty-Sixth ACM Southeast Conference, 2008

# APPENDIX B

# PROTOTYPE CALL INTEGRITY ENGINE CODE

## B.1 Call Integrity Instrumentation Script

Listing B.1: Call Integrity Instrumentation Script

```
#!/usr/bin/perl

#Call integrity source instrumentation
#Jon Jenkins 2014

use strict;
use warnings;

my $CallIndex = 0;#global found call index

sub formMagicBytes{

my $storage = shift;
my $length = shift;
my $i = $length;

while ($i > 0){

$storage = "T".$storage;
$i -= 1;

if ($i<=0){
   next;}

$storage = "N".$storage;
$i -= 1;

if ($i<=0){
   next;}

$storage = "I".$storage;
$i -= 1;

if ($i<=0){
   next;}
```

```
$storage = "C".$storage;
$i -= 1;

if ($i<=0){
   next;}

}

return $storage;
}

sub formMagicBytes2{

my $storage = shift;
my $length = shift;
my $i = 0;

$storage = $storage."\"";

while ($i < $length){

$storage = $storage."C";
$i += 1;

if ($i==$length){
   next;}

$storage = $storage."I";
$i += 1;

if ($i==$length){
   next;}

$storage = $storage."N";
$i += 1;

if ($i==$length){
   next;}

$storage = $storage."T";
$i += 1;

if ($i==$length){
   next;}
```

```
}

$storage = $storage."\"";

return $storage;
}

sub formMagicNumericBytes{
my $initString = shift;
my $magicLen = shift;
my $i = 0;

$initString = $initString."{";

while ($i < ($magicLen-1)){
    $initString = $initString."1414416707,";
    $i++;
     }
$initString = $initString."1414416707";
$initString = $initString."}";

return $initString;
  }

########################################
#if input functionname is format, return its fmt string pos
sub isFormatFunc{

########################################
##hash stores C language
##formatfunction->fmtstring position
my %formatFuncHashList = ("printf",0,
                          "scanf",1);
########################################
my $funcName = shift;

if (exists($formatFuncHashList{$funcName})){
    print "found format function: ".$funcName."\n";
    return $formatFuncHashList{$funcName};
    }

return -1;
}
```

```
########################################
##in the scope of a block
sub scopeMode {
#store the file,scope/func argument list,global decl strings,
my $inScopeSourceFile = shift;
my $scopeargs = shift;
my @allGloVarDeclList = shift;
my $instrdfile = shift;
my $headerfile = shift;

my @varTypes2 = ("int","long","char","float","double");
my @formatFuncList = ("printf","scanf");
my @numericTypes = ("int","long","float","double");
my $openscope = 0;
my @scopeVarDeclList;

while (<$inScopeSourceFile>){

#print "scope mode: line:$_\n";

if ($_ =~ /^\/\//){#ignore comments, stop instrum on magictag
    if ($_ =~ /\/\/CINTEND/){
       print "found cint end tag. Leaving scope mode.\n";
       return;
       }
    next;
    }
#in scope variable decl. possible assignment
#array decl with assignment
elsif ($_ =~ /^(unsigned|long)?\s* (\S+) \s+ (\S+) \[\s*\S+\s*\]\s* =\s
   * (\S+) ;/x){
     #print "in scope var decl w/ assign\n";
     if (grep $2,@varTypes2){
        print "array var decl: #$_# type=$2 name=$3 initstring=$4\n";
        push @scopeVarDeclList,$_;
        }
     }
#array decl w/o assignment
elsif ($_ =~ /^(unsigned|long)?\s* (\S+) \s+ (\S+)\[\s*\S+\s*\];/x){
   #print "in scope var decl no assign.\n";

   if (grep $2,@varTypes2){
        print "array var decl w/o assign: #$_# type=$2 name=$3\n";
        push @scopeVarDeclList,$_;
        }
```

```
    }
#non-array var decl
elsif ($_ =~ /^(unsigned|long)?(int|float|double|char) \s* (\S+)\s* (=\
    s*)? (\S+)?;/x){
    #print "in scope non-array decl\n";
    if (grep $2,@varTypes2){
        if (defined($5)){
            print "non-array var decl: #$_# type=$2 name=$3 initstring=
                $5\n";
            }
        push @scopeVarDeclList,$_;
        }
    }
elsif ($_ =~ /\{/){#control flow scope entry (e.g. for), not a new
    procedure
  $openscope = 1;
  }
elsif ($_ =~ /asm\((\S|\s)*/){
  #ignore asm statements
  }
elsif ($_ =~ /\S+\((\S|\s)*\"(\S|\s)*,(\S|\s)*\"(\S|\s)*/){
   #calls with arguments which contain a , are not handled yet
   }
#CALL. pre: all vars in vardecllist or global vardecllist
elsif ($_ =~ /(\S+)\(((((\S|\s)+,|\S+)+)\);/){ #exclude void calls
      my @callArgs = split ',',$2;
      my $argCount = $#callArgs+1;
      my $isPointer = 0; my $varType = "";
      my $isConstant = 0;
      my $callArgIndex = 0;
      my $callFuncName = $1;
      my @fmtStringTokens;
      my $formatStringArgPosition = -1;
      my $fmtStringIsLiteral = 0; my $literalFmtStringCount = 0;
      my @callArgsIsPointer;

      print "call: funcname $callFuncName, args @callArgs\n";

      $formatStringArgPosition = isFormatFunc($callFuncName);

      if (($formatStringArgPosition >-1) && ($callArgs[
          $formatStringArgPosition]
=~ /\"((\S|\s)+)\"/)){
          $fmtStringIsLiteral = 1;
          my @fmtStringSpecifiers = split '%',$1;
```

```
        $literalFmtStringCount = $#fmtStringSpecifiers;
        }

      foreach (@callArgs){#produce 1 instrumented split call per arg
         my $ptrStgLen = 0;my $initString;my $initContents;my
            @numLiteralList;
         my $ptrLiteralLen = 0;
         $isPointer = 0;
         $isConstant = 0;
         if ($_ =~ /([0-9]+|[0-9]*\.[0-9]+)/ || $_ =~ /\"((\S|\s)+\")
            /){
            $isConstant = 1;}

         my $splitCallArgName = $_;

         ##lookup found call var in normal var DB
         ##check that 'decl matches a regex which includes this call
            arg name'
         ##magic copying is obsolete, so initial values are not needed
            anymore
         foreach my $varInList (@scopeVarDeclList){ #unhandled: arrays
            w/ literal length
            if ($varInList =~ /^(unsigned\s+|long\s+)?\s*
(int|char|double|float) \s+ $_ (\[\s*(\S+)\s*\])? \s* (=\s*)? (\S+)? ;/
   x){
               $ptrStgLen= $3;$varType = $2; $initString= $6;


               if ($varInList =~ /((\S|\s)+) $_ \s* (\[\s*(\S+)\s*\])
                  (\S|\s)*/x){
                  $isPointer = 1;
                  }

               if ($varInList =~ /^(unsigned\s+|long\s+)? (int|char|
                  double|float) *$_ /x){
                  $isPointer = 1;
                  }

               if (defined ($initString)){
                  print "at call, $_ init string: $initString\n";

                 ##decl is initialized to a literal string
                  if ($initString =~ /\s*=\s*\"(.*)\"/){
                   $initContents = $1;
                   $ptrLiteralLen = length($initContents);
```

```
        }

        ##decl is initialized to numeric array literal
        if ($initString =~ /\{(\S+)\}/){
         $initContents = $1;
         @numLiteralList = split ",",$initContents;
         $ptrLiteralLen = $#numLiteralList+1;
         }

        if (not(defined $3)){
           $ptrStgLen = $ptrLiteralLen;
           }
        }
     }
  }


##also check for var in the global vars
foreach my $gloVarInList (@allGloVarDeclList){
   if ($gloVarInList =~ /^(unsigned\s|long\s)?(\S+)\s+$_\[(\S
      +)?\]\s*=\s*(\S+);/){
      $ptrStgLen= $3; $isPointer = 1;$varType = $2;$initString
         = $4;
      if ($initString =~ /\s*=\s*\"(.*)\"/){
         $initContents = $1;
         $ptrLiteralLen = length($initContents);
         }
      if ($initString =~ /\{(.*)\}/){#numeric array literal
         $initContents = $1;
         @numLiteralList = split $initContents,",";
         $ptrLiteralLen = $#numLiteralList;
         }

      if (not(defined $3)){
         $ptrStgLen = $ptrLiteralLen;
         }
      }
   }

if ($isPointer == 1){
   push(@callArgsIsPointer,1);
   }
else {
   push(@callArgsIsPointer,0);
   }
```

```
#splitcall protos to header
my $ptrsymbol = "";
if ($isPointer==1){
   $ptrsymbol = "*";
   }

#
if (!$isConstant){#a C variable (ptr or not)

#prototype write to header
print $headerfile "void ".$callFuncName."_".$CallIndex.
      $callArgIndex."(".$varType." ".$ptrsymbol."$_){}\n";

##instrum bytesum calculation stmt
if ($isPointer == 1){
   my $bytesumstmt = "\nint bytesum".$CallIndex.$callArgIndex.
       " = strSum(".$splitCallArgName.");\n";
   print $instrdfile $bytesumstmt;

   print $instrdfile "long int result_".$CallIndex.
         $callArgIndex."arglen = 1414416707 + 1;\n";
   if (($fmtStringIsLiteral == 0) && ($formatStringArgPosition
      >-1)){
      print $instrdfile "long int result_".$CallIndex.
            $callArgIndex."fmtstring = 1414416707 + 1;\n";
      }
  }

##instrum the edx bytesum post stmt
if ($isPointer == 1){
   my $asminstrbs1 = "asm volatile (\"mov %0, %%edx;\"  \n";
   my $asminstrbs2 = ":: \"r\" (bytesum".$CallIndex.
      $callArgIndex.")\n";
   my $asminstrbs3 = ");\n";
   my $asminstrbsfull = $asminstrbs1.$asminstrbs2.$asminstrbs3
      ;
   print $instrdfile $asminstrbsfull;
   }

##print splitcall    $callFuncName($_)########
print $instrdfile $callFuncName."_".$CallIndex.
      $callArgIndex."(".$splitCallArgName.");\n";
```

```
if ($isPointer == 1){

   #write asm arglen object
   print $instrdfile "PostCallIntASMarglen".$CallIndex.
      $callArgIndex."\n";

  #write asm format string object
   if (($fmtStringIsLiteral == 0) && ($formatStringArgPosition
       >-1)){
      print $instrdfile "PostCallIntASMfmtstring".$CallIndex.
         $callArgIndex."\n";
      }
   print $instrdfile "//\n";
   }

 ##arglength asm sequence
 #must escape backsl, no var interp
 my $asminstr1 = "#define PostCallIntASMarglen".$CallIndex.
                 $callArgIndex." asm volatile (\"movl \$0, %%
                    ecx;\"  \\\n";
my $asminstr2 = "\"movl (%%esp), %%esi; \"  \\\n";
   my $asminstr3 = "\"mov \$0, %%ebx; \"  \\\n";
my $asminstr3a = "\"mov \$0, %%edi; \"  \\\n";
   my $asminstr4 = "\"add \$1, %%ebx; \"  \\\n";
my $asminstr5 = "\"START".$CallIndex.$callArgIndex."a: nop; \"
    \\\n";
   my $asminstr6 = "\"movl (%%esi), %%eax; \"  \\\n";
my $asminstr7 = "\"mov %%al, %%cl; \"  \\\n";
   my $asminstr8 = "\"add %%ecx, %%edi; \"  \\\n";
my $asminstr9 = "\"cmp %%edx, %%edi; \"  \\\n";
   my $asminstr10 = "\"jge DONE".$CallIndex.$callArgIndex."a;
      \"  \\\n";
my $asminstr11 = "\"add \$1, %%ebx; \"  \\\n";
   my $asminstr12 = "\"movl \$0, %%ecx; \"  \\\n";
my $asminstr13 = "\"mov %%ax, %%cx; \"  \\\n";
   my $asminstr14 = "\"shr \$8, %%ecx; \"  \\\n";
my $asminstr15 = "\"add %%ecx, %%edi; \"  \\\n";
   my $asminstr16 = "\"cmp %%edx, %%edi; \"  \\\n";
my $asminstr17 = "\"jge DONE".$CallIndex.$callArgIndex.
   "a; \"  \\\n";  my $asminstr18 = "\"movl \$0, %%ecx; \"  \\\
      n";
my $asminstr19 = "\"add \$1, %%ebx; \"  \\\n";
   my $asminstr20 = "\"mov %%eax, %%ecx; \"  \\\n";
my $asminstr21 = "\"and \$16777215, %%ecx; \"  \\\n";
   my $asminstr22 = "\"shr \$16, %%ecx; \"  \\\n";
```

```
my $asminstr23 = "\"add %%ecx, %%edi; \"  \\\n";
   my $asminstr24 = "\"cmp %%edx, %%edi; \"  \\\n";
my $asminstr25 = "\"jge DONE".$CallIndex.$callArgIndex."a; \"
   \\\n";
my $asminstr26 = "\"add \$1, %%ebx; \"  \\\n";
   my $asminstr27 = "\"movl \$0, %%ecx; \"  \\\n";
my $asminstr28 = "\"mov %%eax, %%ecx; \"  \\\n";
   my $asminstr29 = "\"shr \$24, %%ecx; \"  \\\n";
my $asminstr30 = "\"add %%ecx, %%edi;\"  \\\n";
   my $asminstr31 = "\"cmp %%edx, %%edi;\"  \\\n";
my $asminstr32 = "\"jge DONE".$CallIndex.$callArgIndex."a;\"
   \\\n";
   my $asminstr33 = "\"add \$1, %%ebx;\"  \\\n";
my $asminstr34 = "\"add \$4, %%esi;\"  \\\n";
   my $asminstr35 = "\"jmp START".$CallIndex.$callArgIndex."a
      ;\"  \\\n";
my $asminstr36 = "\"DONE".$CallIndex.$callArgIndex."a: nop;\"
   \\\n";
   my $asminstr37 = "\"mov %%ebx, %0;\"  \\\n";
my $asminstr38 = ": \"=b\" (result_".$CallIndex.$callArgIndex."
   arglen) \\\n";
   my $asminstr39 = ");";

##format string parse asm sequence
 #must escape backsl, no var interp
 my $asminstrf1 = "#define PostCallIntASMfmtstring".$CallIndex.
    $callArgIndex." asm volatile (\"movl \$0, %%ecx; \"  \\\n";
my $asminstrf2 = "\"movl (%%esp), %%esi;\"  \\\n";
  my $asminstrf3 = "\"mov \$0, %%ebx;\"  \\\n";
my $asminstrf4 = "\"mov \$0, %%edi;\"  \\\n";
  my $asminstrf5 = "\"START".$CallIndex.$callArgIndex."f: nop
     ;\"  \\\n";
my $asminstrf6 = "\"movl (%%esi), %%eax;\"  \\\n";
  my $asminstrf7 = "\"mov %%al, %%cl;\"  \\\n";
my $asminstrf8 = "\"add %%ecx, %%edi;\"  \\\n";
  my $asminstrf9 = "\"cmpl \$37, %%ecx;\"  \\\n";
my $asminstrf10 = "\"jne NOPCT1".$CallIndex.$callArgIndex."f;\"
     \\\n";
  my $asminstrf11 = "\"add \$1, %%ebx;\"  \\\n";
my $asminstrf12 = "\"NOPCT1".$CallIndex.$callArgIndex."f: nop
   ;\"  \\\n";   my $asminstrf13 = "\"cmp %%edx, %%edi;\"  \\\n
   ";
my $asminstrf14 = "\"jge DONE".$CallIndex.$callArgIndex."f;\"
   \\\n";
  my $asminstrf15 = "\"movl \$0, %%ecx;\"  \\\n";
```

```
my $asminstrf16 = "\"mov %%ax, %%cx;\"   \\\n";
  my $asminstrf17 = "\"shr \$8, %%ecx;\"   \\\n";
my $asminstrf18 = "\"cmpl \$37, %%ecx;\"   \\\n";
  my $asminstrf19 = "\"jne NOPCT2".$CallIndex.$callArgIndex."f
      ;\"   \\\n";
my $asminstrf20 = "\"add \$1, %%ebx;\"   \\\n";
  my $asminstrf20a = "\"NOPCT2".$CallIndex.$callArgIndex."f:
     nop;\"   \\\n";
my $asminstrf21 = "\"add %%ecx, %%edi;\"   \\\n";
   my $asminstrf22 = "\"cmp %%edx, %%edi;\"   \\\n";
my $asminstrf23 = "\"jge DONE".$CallIndex.$callArgIndex."f;\"
   \\\n";
   my $asminstrf24 = "\"movl \$0, %%ecx;\"   \\\n";
my $asminstrf25 = "\"mov %%eax, %%ecx;\"   \\\n";
   my $asminstrf26 = "\"and \$16777215, %%ecx;\"   \\\n";
my $asminstrf27 = "\"shr \$16, %%ecx;\"   \\\n";
   my $asminstrf28 = "\"cmpl \$37, %%ecx;\"   \\\n";
my $asminstrf29 = "\"jne NOPCT3".$CallIndex.$callArgIndex."f;\"
     \\\n";
   my $asminstrf30 = "\"add \$1, %%ebx;\"   \\\n";
my $asminstrf31 = "\"NOPCT3".$CallIndex.$callArgIndex."f: nop
   ;\"   \\\n";
   my $asminstrf32 = "\"add %%ecx, %%edi;\"   \\\n";
my $asminstrf33 = "\"cmp %%edx, %%edi;\"   \\\n";
   my $asminstrf34 = "\"jge DONE".$CallIndex.$callArgIndex."f
      ;\"   \\\n";
my $asminstrf35 = "\"movl \$0, %%ecx;\"   \\\n";
   my $asminstrf36 = "\"mov %%eax, %%ecx;\"   \\\n";
my $asminstrf37 = "\"shr \$24, %%ecx;\"   \\\n";
   my $asminstrf38 = "\"cmpl \$37, %%ecx;\"   \\\n";
my $asminstrf39 = "\"jne NOPCT4".$CallIndex.$callArgIndex."f;\"
     \\\n";
   my $asminstrf40 = "\"add \$1, %%ebx;\"   \\\n";
my $asminstrf41 = "\"NOPCT4".$CallIndex.$callArgIndex."f: nop
   ;\"   \\\n";
   my $asminstrf42 = "\"add %%ecx, %%edi;\"   \\\n";
my $asminstrf43 = "\"cmp %%edx, %%edi;\"   \\\n";
   my $asminstrf44 = "\"jge DONE".$CallIndex.$callArgIndex."f
      ;\"   \\\n";
my $asminstrf45 = "\"add \$4, %%esi;\"   \\\n";
   my $asminstrf46 = "\"jmp START".$CallIndex.$callArgIndex."f
      ;\"   \\\n";
my $asminstrf47 = "\"DONE".$CallIndex.$callArgIndex."f: nop;\"
    \\\n";
   my $asminstrf48 = "\"mov %%ebx, %0;\"   \\\n";
```

```
my $asminstrf49 = ": \"=b\" (result_".$CallIndex.
    $callArgIndex."fmtstring) \\\n";
my $asminstrf50 = ");";

my $fullASMString = $asminstr1.$asminstr2.$asminstr3.
    $asminstr3a.
    $asminstr4.$asminstr5.$asminstr6.$asminstr7.$asminstr8.
       $asminstr9.
    $asminstr10.$asminstr11.$asminstr12.$asminstr13.$asminstr14.
    $asminstr15.$asminstr16.$asminstr17.$asminstr18.$asminstr19.
    $asminstr20.$asminstr21.$asminstr22.$asminstr23.$asminstr24.
       $asminstr25.
    $asminstr26.$asminstr27.$asminstr28.$asminstr29.$asminstr30.
    $asminstr31.$asminstr32.$asminstr33.$asminstr34.
    $asminstr35.$asminstr36.$asminstr37.$asminstr38.$asminstr39;

my $fullASMStringf = $asminstrf1.$asminstrf2.$asminstrf3.
    $asminstrf4.
    $asminstrf5.$asminstrf6.$asminstrf7.$asminstrf8.$asminstrf9.
    $asminstrf10.$asminstrf11.$asminstrf12.$asminstrf13.
       $asminstrf14.
    $asminstrf15.$asminstrf16.$asminstrf17.$asminstrf18.
       $asminstrf19.
    $asminstrf20.$asminstrf20a.$asminstrf21.$asminstrf22.
       $asminstrf23.
    $asminstrf24.$asminstrf25.
    $asminstrf26.$asminstrf27.$asminstrf28.$asminstrf29.
       $asminstrf30.
    $asminstrf31.$asminstrf32.$asminstrf33.$asminstrf34.
    $asminstrf35.$asminstrf36.$asminstrf37.$asminstrf38.
       $asminstrf39.
    $asminstrf40.$asminstrf41.$asminstrf42.
    $asminstrf43.$asminstrf44.$asminstrf45.$asminstrf46.
       $asminstrf47.
    $asminstrf48.$asminstrf49.$asminstrf50;


    if ($isPointer){
       print $headerfile "$fullASMString\n";
       }
    if (($fmtStringIsLiteral == 0) && ($formatStringArgPosition
       >-1)){
       print $headerfile "$fullASMStringf\n";
       }
```

```
      print $headerfile "\n";
      }
    $callArgIndex += 1;
    }

#print "args is pointer: @callArgsIsPointer\n";

$argCount -= 1;

##print integrity evaluation stmt
#OPTION: if (@callArgsIsPointer[x] == 1) can check
#"result_".$CallIndex.x."arglen" to cmp lengths


##int violation detection stmts##
if ($formatStringArgPosition >-1){

  if ($fmtStringIsLiteral == 0){
     print $instrdfile "//DETECTION \nif (".$argCount." !=
        result_".$CallIndex.
        $formatStringArgPosition.
         "fmtstring){printf(\"format string violation:count
            mismatch!!\");}\n\n";
     }
  else {
     print $instrdfile "//DETECTION \nif (".$argCount." != ".
        $literalFmtStringCount.
       "){printf(\"format string violation:count mismatch!!\");}\
          n\n";
     }
 }
#####################################

##OBSOLETE write magic markrealcall<i> allocation stmt
#print $instrdfile "long int ".$callFuncName."_magicmark".
   $asmCallIndex." = 1414416707 + 1;\n";

#write violation check bytecounts
#specify a requirement spec(bytecounts) here

##OBSOLETE write origcall argcount magic mark stmt
#print $instrdfile "long int ".$callFuncName."_magicargcount =
   1414416707 - ".($argCount+1).";\n";

print "call, func=$callFuncName
```

```
            args=@callArgs##globalssofar=@allGloVarDeclList varssofar=
               @scopeVarDeclList\n";

      ##index for ASM instrum labels
      $CallIndex += 1;
      }
#'}' without a previous '{', leave the scope
elsif ($_ =~ /\}$/){
      if ($openscope == 0){
         print "leaving scope (args $scopeargs)\n";
         print $instrdfile $_;
         return;
         }
      else {
        $openscope = 0;
        }
      }
else {print "unhandled scope mode string:".$_."\n";}

   #dont print calls until instrum has occurred
   print $instrdfile $_; #write in scope/function lines to instrumented
       file

   }
}
########################################


########################################
sub instrMode{

my $sourcefile = shift;
my $instrdfile = shift;
my $headerfile = shift;

my @varTypes = ("int","long","char","float","double");
my @gloVarDeclList;

while (<$sourcefile>){

  print $instrdfile $_; #write to instrumented file, outside of
     function scopes

  #find glo. variable decl
  if ($_ =~ /^(unsigned\s|long\s)?(\S+)\s+(\S+)(\s+=\s+\S+)?;/){
```

```
      if (grep $2,@varTypes){
         print "found global var decl: $_ type=$2 name=$3\n";
         push @gloVarDeclList,$_;
         }
      }

      #find function def/scope entry
   if ($_ =~ /\S+\s+\S+\(((\S+\s+\S+,|\S+\s+\S+)+)\)\{/xsm){
       my @defargs = split ',',$1;
       print "found scope entry: $_ args : @defargs\n";
       scopeMode($sourcefile,$1,@gloVarDeclList,$instrdfile,$headerfile)
          ;
       }

    if ($_ =~ /^\/\/CINTEND/){
       print "found cint end tag. Leaving Instrumentation mode.\n";
       return;
       }
    }
}
################################################


################################################
# toplevel

my $mainoutput = 0;
#open the input uninstrumented c file
open(my $infile, "<",$ARGV[0]) or die "failed to open $1: $!";

#write a new instrumented c file
open(my $outfile, ">",$ARGV[0].".instr.c") or die "failed to open $1
    for writing: $!";

#file.instr.c #include "cint.h"  write a header cint.h for ASMs and
    split call prototypes
open(my $headerfile, ">",$ARGV[0].".instr.c-cint.h") or die "failed to
    open $1 for writing: $!";

print $headerfile "int strSum(char *str){\n
//char *ptr = str;\n
int sum = 0,i=0;\n
while (str[i] != \'\\0\'){\n
   sum = sum + (int)str[i++];\n
   }\n
```

```
return sum;\n
}\n";

print $outfile "#include \"".$ARGV[0].".instr.c-cint.h\"";

while (<$infile>){

    print $outfile $_;
    chomp($_); #strip newlines

    if (substr($_,0,11) eq "//CINTBEGIN"){
        print "found call integrity start tag\n";
        $mainoutput = 1;
        instrMode($infile,$outfile,$headerfile);
        }
}
```

## B.2 Call Integrity Data Collection Script Using DynamoRIO

Listing B.2: Call Integrity Data Collection Script Using DynamoRIO

```
#!/usr/bin/perl

#some content shortened

my $i = 0;
my $datafile = $ARGV[0].".data";
#open($datafileh, ">>",$datafile) or die "failed to open $datafile for
    writing: $!";

my $cmd = "/usr/bin/time -a -f \"%e,%U,%S\" -o $datafile /.../bin32/
    drrun -v -c .../DynamoRIO-Linux-4.2.0-3/myclient/stackDemo -- ".
    $ARGV[0];

while ( $i < 1000 ) {
    my $result = qx($cmd); #qx and not system: capture prog output
    $i++;
    }
```

## B.3 Base Call Integrity Data Collection Script

Listing B.3: Base Call Integrity Data Collection Script

```
#!/usr/bin/perl

my $i = 0;
my $datafile = $ARGV[0].".data";
open($datafileh, ">>",$datafile) or die "failed to open $datafile for
    writing: $!";

my $cmd = "./".$ARGV[0];

while ( $i < 1000 ) {
   my $result = qx($cmd); #qx and not system: capture prog output
    print $datafileh $result."\n";
    $i++;
    }
```

## B.4 Call Integrity Data Collection Script: Run-time Use

Listing B.4: Call Integrity Data Collection Script: Run-time Use

```
#!/usr/bin/perl

my $i = 0;
my $datafile = $ARGV[0].".data";
open($datafileh, ">>",$datafile) or die "failed to open $datafile for
    writing: $!";

my $cmd = "echo \"%d %s\" | ./".$ARGV[0];

while ( $i < 1000 ) {
   my $result = qx($cmd); #qx and not system: capture prog output
    print $datafileh $result."\n";
    $i++;
    }
```

## B.5 DynamoRIO Binary Instrumentation Client Application

Listing B.5: DynamoRIO Binary Instrumentation Client Application

```
//stackDemo.c a dynamorio client (c) Jon Jenkins 2014
//
```

```
////////////////////////////////
//must define arch and OS for DR
#ifndef LINUX
#define LINUX
#endif
#ifndef X86_32
#define X86_32
#endif
////////////////////////////////

//#include <stdio.h>
//#include <string.h>
#include "dr_api.h"
/*
#include "dr_ir_instr.h"
#include "dr_ir_instrlist.h"
#include "dr_ir_utils.h"
#include "dr_ir_opnd.h"
#include "dr_tools.h"
*/

//#define MAGICINT 1128877652    hex CINT
#define MAGICINT 1414416707      //hex TNIC

static dr_emit_flags_t event_basic_block(void *drcontext,void *tag,
    instrlist_t *bb,
bool for_trace, bool translating);
void myRestoreState(void *drcontext, void *tag, dr_mcontext_t *mcontext
    ,
bool restore_memory, bool app_code_consistent);
static int checkUsage(byte *pc, byte *targetpc);

void * as_built_lock; //a DR mutex
opnd_t arglocs[30];
int arglocind = 0;

/*
//Will not actually overrun the buffer because
//DR requires stack protection disabled
int myStrcpy(char *deststr,char *srcstr,int length){

if (length > strlen(deststr)){
  printf("dangerous strcpy. Abort\n");
  return 1;
```

```
  }

return 0;
}
*/

//The instrumentation function to insert before USAGE
//currently OBSOLETE
static int checkUsage(app_pc pc, app_pc targetpc){
bool internal;

/*
//get into structure about memory
if ((internal = dr_memory_is_dr_internal()) == false){
dr_query_memory_ex (     const byte *     pc,
                         OUT dr_mem_info_t *      info
                         )
  }

*/
//dr_printf("time to check USAGEINT\n");

//dr_printf("eax \n");

//dr_printf("###checkUsage: pc: %x, destpc: %x\n",pc,targetpc);

return 0;
}


////////////////////////////////////
//The initialization point for DR//////
DR_EXPORT void dr_init(client_id_t id){
int i=0;

//restore state event callback for when DR wants to restore machine
    state
dr_register_restore_state_event(myRestoreState);

//DB: register our events with DR
 dr_register_bb_event(event_basic_block);

//USAGEINT: register an event on 'call'
//dr_insert_call_instrumentation()
```

```
//consider these events also (dr_register_pre_syscall_event(),
    dr_register_post_syscall_event(),
// since syscalls are a usage with high risks

//init. a mutex for synchronization of:
//USAGEINT a per-usage argument counter.
//USAGE = call or syscall sequence as seen by DR, to start with...
//as_built_lock = dr_mutex_create();

//


}

printAllInst(void *drcontext, instrlist_t *lst){

instr_t *instrptr = instrlist_first(lst);
dr_printf("#bb instruction list: \n");

while (instrptr != NULL){
  dr_print_instr(drcontext,our_stdout,instrptr,"  instr: ");
  instrptr = instr_get_next(instrptr);
  }

}

void myRestoreState(void *drcontext, void *tag, dr_mcontext_t *mcontext
    , bool restore_memory, bool app_code_consistent){
//allow client to restore state upon a fault?

if (restore_memory){
  //restore the machine state to what it was
    ;
    }
}


int nullifyCall(void *drcontext, instrlist_t *bb,instr_t *calltodump){
        //catch and replace call instr//
        opnd_t regecx = opnd_create_reg(DR_REG_ECX);
        instr_t* newnop = INSTR_CREATE_nop(drcontext);
        instr_t* newjecxz = INSTR_CREATE_jecxz(drcontext,
           opnd_create_instr(newnop));
        instr_t* newmov = INSTR_CREATE_mov_imm(drcontext,regecx,
           opnd_create_immed_int(2,OPSZ_4));
```

```
        instr_set_translation(newjecxz, instr_get_app_pc(calltodump));
        instr_set_translation(newmov, instr_get_app_pc(newjecxz));
        instr_set_translation(newnop, instr_get_app_pc(newmov));

        instrlist_preinsert(bb, calltodump, newnop);
        instrlist_preinsert(bb, calltodump, newmov);
        instrlist_preinsert(bb, calltodump, newjecxz);
        instrlist_remove(bb, calltodump); //dump the call instr
return 0;
}

//insert instructions to measure byte length of arg, stopping at next
    arg esp byte offset
//measurement instrs will be inserted after this instr, but prior to
    call
int argLenMeasure(void *drcontext, instrlist_t *bb, instr_t *
    argLocationInstr, int nextArgOffset){

    ////determine reg holding the

    //int numsrcs = instr_num_srcs(argLocationInstr);
    //int numdsts = instr_num_dsts(argLocationInstr);
    opnd_t dstop, srcop;
    reg_id_t srcreg, dstreg, reg;
    int movdisp=0;

    if (instr_is_mov(argLocationInstr) || instr_is_floating(
        argLocationInstr)) {

        dstop = instr_get_dst(argLocationInstr, 0);
        //srcop = instr_get_src(argLocationInstr, 0);

        //mov is to esp+x
        if (opnd_is_base_disp(dstop) &&
            ((reg = opnd_get_base(dstop)) == DR_REG_ESP)){
            movdisp = opnd_get_disp(dstop);
            }
        }


    ////insert asm to measure
    opnd_t dsteax = opnd_create_reg(DR_REG_EAX);
    opnd_t srcespoff = opnd_create_base_disp(DR_REG_ESP, DR_REG_NULL,  1,
        /*scale*/movdisp, OPSZ_4);
```

```
    instr_t* newmov = INSTR_CREATE_mov_ld(drcontext, dsteax, srcespoff);
       //mov eax,(esp+off); (<-)

}

////speculatively extract the esp arg offset operands for an original
    near direct call
int extractArgOffsetOpnds(void *drcontext,instr_t *extractIter, instr_t
     *endMarkerInstr){
opnd_t dstop,srcop;
reg_id_t srcreg,dstreg,reg;
int movdisp = 0, i = 0;

while (!instr_same(extractIter,endMarkerInstr)){

    if (instr_is_mov(extractIter) || instr_is_floating(extractIter)) {

       dstop = instr_get_dst(extractIter,0);
       //srcop = instr_get_src(argLocationInstr,0);

       //mov is to esp+x
       if (opnd_is_base_disp(dstop) &&
          ((reg = opnd_get_base(dstop)) == DR_REG_ESP)){
          movdisp = opnd_get_disp(dstop);

          arglocs[arglocind++] = dstop;
          }
       }
    extractIter = instr_get_prev(extractIter);
    }

return 0;
}


int addressLoadSearch(void *drcontext,instr_t *extractIter, instr_t *
    endMarkerInstr){
opnd_t dstop,srcop;
reg_id_t srcreg,dstreg,reg;
int movdisp = 0, opcode  = 0;

while (!instr_same(extractIter,endMarkerInstr)){

     if ((opcode = instr_get_opcode(extractIter)) == OP_lea) {
```

```
        //dstop = instr_get_dst(extractIter,0);
        srcop = instr_get_src(extractIter,0); //a base+disp address for a
            ptr arg

        //
        if (opnd_is_base_disp(srcop) &&
           (((reg = opnd_get_base(srcop)) == DR_REG_ESP) || (reg ==
              DR_REG_EBP))  ){

           ////with ptr arg address, speculatively measure bytelength


           //arglocs[arglocind++] = dstop;
           }
        }
     extractIter = instr_get_prev(extractIter);
     }

return 0;
}

int argCountHeuristic(instr_t *iter, int lastESPoff, void *drcontext){
int argcount=0;
opnd_t dstop,srcop;
reg_id_t srcreg,dstreg,reg;

while (iter != NULL){
          //mov to esp+x or fstp to esp+x. Literals not handled
          if (instr_is_mov(iter) || instr_is_floating(iter)) { //

             dstop = instr_get_dst(iter,0);
             srcop = instr_get_src(iter,0);
             int movInt = 0;

             //stop if instrum. magic mark instrs. are found
             if (opnd_is_immed_int(srcop) &&
                  ((movInt=opnd_get_immed_int(srcop)) == MAGICINT)){
                 break;
                 }

             //found the stack frame setup instr..leave
             if (opnd_is_reg(dstop) && opnd_is_reg(srcop)){
                 if ( ((srcreg = opnd_get_reg(srcop)) == DR_REG_ESP) &&
                           ((dstreg = opnd_get_reg(dstop)) == DR_REG_EBP)
                              )
```

```
                break;
            }

        //detect mov or fstp to base/disp at x(esp)
        if (opnd_is_base_disp(dstop) &&
                    ((reg = opnd_get_base(dstop)) == DR_REG_ESP)
                        ){

          int movdisp = opnd_get_disp(dstop);
          if (movdisp == (lastESPoff)){
             argcount += 1; //this is likely a call site proc. arg
                write
             lastESPoff += 4;
             }
          else
             {break;} //found noncontig. mem. mov->likely not an arg
          }
        }
      dr_printf("origcall:instr(backwards_from_call):_");
      dr_print_instr(drcontext,our_stdout,iter,"#####");//DB
      iter = instr_get_prev(iter);
      }

return argcount;
}

static dr_emit_flags_t event_basic_block(void *drcontext,void *tag,
   instrlist_t *bb,
bool for_trace, bool translating){
bool getresult;
dr_mcontext_t mycontext;
dr_mcontext_t *con_ptr = &mycontext;
int regebp = 0, regesp = 0;
instr_t *instrptr = NULL;
instr_t *lasti = NULL,*firsti = NULL;

////////////////////////////////
//got a basic block. Now do USAGEINT

//con_ptr->size = sizeof(mycontext);
//con_ptr->flags = 0;
//con_ptr->flags =  con_ptr->flags ^ DR_MC_INTEGER;
//con_ptr->flags =  con_ptr->flags ^ DR_MC_CONTROL;
```

```
instrptr = instrlist_first(bb);
firsti = instrlist_first(bb);
lasti = instrlist_last(bb);

/*
dr_printf("$$$ begin bb preview");
///preprint BB //DB
while (instrptr != NULL){
  dr_print_instr(drcontext, our_stdout, instrptr,"#####instr: "); //DB
  instrptr = instr_get_next(instrptr);
}
dr_printf("$$$ end preview");

instrptr = instrlist_first(bb);
*/

int opc;

while (instrptr != NULL){

  //dr_print_instr(drcontext, our_stdout, instrptr,"#####ANY instr: ");
     //DB

  opc = instr_get_opcode(instrptr);

   if (opc == OP_sub){

     int numsrcs = instr_num_srcs(instrptr);
     int numdsts = instr_num_dsts(instrptr);

   ////////////////////////////////////////////
   //locate stack allocation instr, extract distance
   //dr_printf("##sub: %d dstopnds %d srcopnds\n",numsrcs,numdsts);
   //dr_print_instr(drcontext, our_stdout, instrptr,"##instr: ");
   if (numsrcs < 2) {continue;}

   if (opnd_is_reg(instr_get_dst(instrptr,0)) && opnd_is_reg(
      instr_get_src(instrptr,1))  &&
        opnd_is_immed_int(instr_get_src(instrptr,0)) ) {

   opnd_t dst1op = instr_get_dst(instrptr,0);
   opnd_t src1op = instr_get_src(instrptr,0); //immed
   opnd_t src2op = instr_get_src(instrptr,1); //esp
   reg_id_t myreg;
```

```
if ( ((myreg=opnd_get_reg(dst1op)) == DR_REG_ESP) &&
     ( (myreg=opnd_get_reg(src2op)) == DR_REG_ESP) ){

  instr_t *newinstr = NULL; instr_t *where = NULL;
  instr_t *translTarget = instr_get_next(instrptr);

  //dr_mcontext_t mc = {sizeof(mc),DR_MC_CONTROL/*only need xsp*/};
 //dr_get_mcontext(dr_get_current_drcontext(), &mc);

  int subtrahend = opnd_get_immed_int(src1op);//get the hex byte sub
      operand

  if (subtrahend>4){//found routine stack alloc instr..magic mark

     //dr_print_instr(drcontext,our_stdout,instrptr,"##sub instr: ")
        ;
     int byte = 0 ;//DBDB !
     int displacement = 0;

     where = instr_get_next(instrptr);

     while (byte<subtrahend){//
        //build dword move of 'subtrahend' magic bytes onto stack
        opnd_t mvdst = opnd_create_base_disp(DR_REG_ESP,DR_REG_NULL,
             1, //scale
             displacement,
              OPSZ_4);
        opnd_t mvsrc =  opnd_create_immed_int (MAGICINT,OPSZ_4); //
           magic '0INT' bytes
        newinstr = INSTR_CREATE_mov_imm(drcontext, mvdst, mvsrc);

        instr_set_translation(newinstr,instr_get_app_pc(translTarget
           ));
        instrlist_preinsert(bb,where,newinstr);

        translTarget = newinstr;
        instr_set_translation(newinstr,instr_get_app_pc(translTarget
           ));

        where = instr_get_prev(where);

        displacement += 4;
        byte+=4;
        }
     }
```

```
    }

    }

  //dr_printf("$$$end sub instrum\n");//DB
  }


if (instr_is_near_call_direct(instrptr)){//call must be 'near',
    direct (nonreg)

    //dr_printf("$$$begin dcall instrum\n");//dr_printf("call pc=%x\n
        ",instr_get_app_pc(instrptr));//dr_printf("$$$1\n");

    bool issplitcall = false,isorigcall = false,violation = false;
    opnd_t bytectbasedisp;
    int srcargcount = 0;
     ////////////////////////////////////////////////////
     //detect split call, argbytecount storage loc.///////
     //DISABLEDREQ: call must not be lone//////////////////////////////
     if ((instrptr == instrlist_last(bb))/* && (instrptr !=
         instrlist_first(bb))*/){

      ptr_int_t constant = 0;
      instr_t *iter2 = instr_get_prev(instrptr);
      //reverse scan for splitcall detection, bytecount var loc
      while (iter2 != NULL){

       if (instr_is_mov_constant(iter2,&constant)){
          //marker stmt for split call
          if ((constant > MAGICINT) && (constant <= (MAGICINT+100))){
             dr_printf("found split call: ");
            //dr_print_instr(drcontext,our_stdout,instrptr,"#####");
            issplitcall = true;

             break;
             }
          //marker stmt for an orig. call, extract offsets
          else if ((constant < MAGICINT) && (constant >= (MAGICINT
             -100))){
              srcargcount = MAGICINT-constant;
              isorigcall = true;

              instr_t *extractIter = instr_get_prev(instrptr);
```

```
            ////extract offsets with reverse esp+x write scan,
                bounded at the marker mov instr

            extractArgOffsetOpnds(drcontext,extractIter,iter2);


            extractIter = instr_get_prev(instrptr);
            ////address load search
            addressLoadSearch(drcontext,extractIter,iter2);

            //this real call will end any bb
            }
        //Integrity violation code
        else if ((constant < (MAGICINT-100)) && (constant >= (
            MAGICINT-120))){
            //dr_printf("found precall INT violation!\n");
            violation = true;
            //dr_print_instr(drcontext,our_stdout,instrptr,"#####");
            }
        else
            ;//this mov is near some uninstrumented call. Ignore
        }
    iter2 = instr_get_prev(iter2);
    }
   ////////////////////////////////////////


////////////////////////////////////////////
int argcount=0, maxarg = 20, lastESPoffset = 0,movedint = 0;
instr_t *iter = instr_get_prev(instrptr);
opnd_t dstop,srcop;
reg_id_t srcreg,dstreg,reg;

////////////////////////////////////////////
//call argcount heuristic
if (isorigcall) {

    argcount = argCountHeuristic(iter,lastESPoffset,drcontext);


    //offsets were extracted when origcall marker was detected
        earlier
     int i = 0;
    dr_printf("DR heuristic argcount: %d, orig. call bytecount
        offsets: \n",argcount);
```

```
 while (i<arglocind){
   dr_printf("%x,",opnd_get_disp(arglocs[i]));
   i++;
   }
 dr_printf("\n");
 arglocind = 0;

 ////measure arglenths pure BININSTR


//if violation, apply enforcement
 //SPEC VIOLATION ENFORCEMENT: avoidance
 if (violation){
     dr_printf("at origcall: violation");
     nullifyCall(drcontext,bb,instrptr);
     }
////////////////////////////////////////

 ////DB
 dr_printf("$$$ begin bb ocall postview");
 instr_t* tmp4 = instrlist_first(bb);

 while (tmp4 != NULL){
    //dr_print_instr(drcontext,our_stdout,tmp4,"#####instr: ");
       //DB
    tmp4 = instr_get_next(tmp4);
    }
 dr_printf("$$$ end ocall postview");
 ////DB
}


 ////////////////////////////////////////////////////////////
 //replace BB end instrs and prepare registers for length check
 if (issplitcall){

 ////////////////////////////////////
 //detect a split call pointer argument
 iter2 = instr_get_prev(instrptr);
 bool argislocalpointer = false;
 int maxleadistance = 5, distance = 0;

 instr_t *newebxmov = NULL;
 instr_t *leaptr = NULL;
 int testopcode = 0;
```

```
while ((iter2 != NULL) && (distance<maxleadistance)){
   testopcode = instr_get_opcode(iter2);
   if (testopcode == OP_lea){argislocalpointer = true;leaptr =
      iter2;}
   iter2 = instr_get_prev(iter2);
   distance += 1;
   }

opnd_t regebx = opnd_create_reg(DR_REG_EBX);
opnd_t regeax = opnd_create_reg(DR_REG_EAX);
opnd_t regesp = opnd_create_reg(DR_REG_ESP);

////////////////////////////////////////
//PRE-ASM setup: MOV adjust ebx for pointer args  OBSOLETE
/*
if (argislocalpointer){
   newebxmov = instr_create_1dst_1src(drcontext, OP_mov_ld,
      regebx, regeax);
   //instr_set_translation(newebxmov, instr_get_app_pc(
      instr_get_next(leaptr)));//xl8 ptr to the instr after LEA
   instrlist_postinsert(bb, leaptr, newebxmov);
   }
else {//a split call, but no {eff. address preload to eax}
   instr
       //1) data mov'd to (esp)  OR
       //2) global var IMEM address written to (esp)
   //glo. var currently handled by copy-with-mark
   newebxmov = instr_create_1dst_1src(drcontext, OP_mov_ld,
      regebx, regesp);
   //instr_set_translation(newebxmov, instr_get_app_pc(instrptr)
      );
   instrlist_preinsert(bb, instrptr, newebxmov);
   }
*/
////////////////////////////////////////

nullifyCall(drcontext,bb,instrptr);

//DB//
dr_printf("$$$ begin bb scall postview");
instr_t* tmp3 = instrlist_first(bb);

while (tmp3 != NULL){
        dr_print_instr(drcontext, our_stdout, tmp3,"#####instr: "
           ); //DB
```

```
                tmp3 = instr_get_next(tmp3);
                }
        dr_printf("$$$ end scall postview");
        //////
        }

       }
      //dr_printf("end dcall instr\n");
     }

  instrptr = instr_get_next(instrptr);
}


/*
dr_printf("$$$ begin bb postview");
///preprint BB //DB
 instrptr = instrlist_first(bb);

while (instrptr != NULL){
  dr_print_instr(drcontext, our_stdout, instrptr, "\n#####instr: "); //DB
  instrptr = instr_get_next(instrptr);
  }
dr_printf("$$$ end postview");
*/

//dr_printf("$$$end BB");

////////////////////////////////////////////////

return DR_EMIT_STORE_TRANSLATIONS;//causes DR to store xlation info for
    bb's, with cost //DR_EMIT_DEFAULT;//
}


/* //don't need main. This file is just functions used by DR
int main(int argc, char argv[]){
//float myfloat = 1.5;
char mystring1[3] = "aaa";
char mystring2[4] = "bbbb";

//enable DR functionalities, inserts instrum. functions
//dr_init(client_id_t id);

//the goal is to catch the mismatch BEFORE the call
```

```
//myStrcpy(mystring1,mystring2,4);

return 0;
}
*/

int testEval(void *drcontext,instrlist_t *bb,instr_t *lasti){

////////////////////////////////////////////////
     //e.g. length
      //insert instr to write both INTPRIMS to regs
      opnd_t opndevalecx = opnd_create_reg(DR_REG_ECX);

      opnd_t opndevaleax = opnd_create_reg(DR_REG_EAX);

      opnd_t opndevalesp = opnd_create_reg(DR_REG_ESP);
      reg_id_t evalregesp =  DR_REG_ESP;

      instr_t* evalmov1 = INSTR_CREATE_mov_ld(drcontext,opndevalecx,
         opnd_create_base_disp(evalregesp,DR_REG_NULL,1,
         INTPRIMbaseDisps[0],OPSZ_4) );
      instr_t* evalmov2 = INSTR_CREATE_mov_ld(drcontext,opndevaleax,
         opnd_create_base_disp(evalregesp,DR_REG_NULL,1,
         INTPRIMbaseDisps[1],OPSZ_4) );
      instr_t* evalcmp = INSTR_CREATE_cmp(drcontext,opndevalecx,
         opndevaleax);

      //instr_t* newnop = INSTR_CREATE_nop(drcontext);

      //  instr_t* newjecxz = INSTR_CREATE_jecxz(drcontext,
         opnd_create_instr(newnop));
      //  instr_t* newmov = INSTR_CREATE_mov_imm(drcontext,regecx,
         opnd_create_immed_int(2,OPSZ_4));

      instr_set_translation(evalcmp,instr_get_app_pc(lasti));
      instr_set_translation(evalmov2,instr_get_app_pc(evalcmp));
      instr_set_translation(evalmov1,instr_get_app_pc(evalmov2));

        //segv
        //dr_save_arith_flags(drcontext,bb,instrptr,SPILL_SLOT_3);
        instrlist_preinsert(bb,lasti,evalmov1);
        //dr_save_reg(drcontext,bb,instrptr,DR_REG_ECX,SPILL_SLOT_2);//
           save ecx
        instrlist_preinsert(bb,lasti,evalmov2);
        instrlist_preinsert(bb,lasti,evalcmp);
```

```
return 0;
}
```

## B.6 Sample C Header File: Parameter Length Measurement

Listing B.6: Sample C Header File: Parameter Length Measurement

```
int strSum(char *str){
int sum = 0,i=0;

while (str[i] != '\0'){
   sum = sum + (int)str[i++];
   }
return sum;
}
void func_00(char *mystring){}
#define PostCallIntASMarglen00 asm volatile ("movl $0, %%ecx;"  \
"movl (%%esp), %%esi; "  \
"mov $0, %%ebx; "  \
"mov $0, %%edi; "  \
"add $1, %%ebx; "  \
"START00a: nop; "  \
"movl (%%esi), %%eax; "  \
"mov %%al, %%cl; "  \
"add %%ecx, %%edi; "  \
"cmp %%edx, %%edi; "  \
"jge DONE00a; "  \
"add $1, %%ebx; "  \
"movl $0, %%ecx; "  \
"mov %%ax, %%cx; "  \
"shr $8, %%ecx; "  \
"add %%ecx, %%edi; "  \
"cmp %%edx, %%edi; "  \
"jge DONE00a; "  \
"movl $0, %%ecx; "  \
"add $1, %%ebx; "  \
"mov %%eax, %%ecx; "  \
"and $16777215, %%ecx; "  \
"shr $16, %%ecx; "  \
"add %%ecx, %%edi; "  \
"cmp %%edx, %%edi; "  \
"jge DONE00a; "  \
"add $1, %%ebx; "  \
```

```
"movl $0, %%ecx; "  \
"mov %%eax, %%ecx; "  \
"shr $24, %%ecx; "  \
"add %%ecx, %%edi;"  \
"cmp %%edx, %%edi;"  \
"jge DONE00a;"  \
"add $1, %%ebx;"  \
"add $4, %%esi;"  \
"jmp START00a;"  \
"DONE00a: nop;"  \
"mov %%ebx, %0;"  \
: "=b" (result_00arglen) \
);

void func_01(char *garr){}
#define PostCallIntASMarglen01 asm volatile ("movl $0, %%ecx;"  \
"movl (%%esp), %%esi; "  \
"mov $0, %%ebx; "  \
"mov $0, %%edi; "  \
"add $1, %%ebx; "  \
"START01a: nop; "  \
"movl (%%esi), %%eax; "  \
"mov %%al, %%cl; "  \
"add %%ecx, %%edi; "  \
"cmp %%edx, %%edi; "  \
"jge DONE01a; "  \
"add $1, %%ebx; "  \
"movl $0, %%ecx; "  \
"mov %%ax, %%cx; "  \
"shr $8, %%ecx; "  \
"add %%ecx, %%edi; "  \
"cmp %%edx, %%edi; "  \
"jge DONE01a; "  \
"movl $0, %%ecx; "  \
"add $1, %%ebx; "  \
"mov %%eax, %%ecx; "  \
"and $16777215, %%ecx; "  \
"shr $16, %%ecx; "  \
"add %%ecx, %%edi; "  \
"cmp %%edx, %%edi; "  \
"jge DONE01a; "  \
"add $1, %%ebx; "  \
"movl $0, %%ecx; "  \
"mov %%eax, %%ecx; "  \
"shr $24, %%ecx; "  \
```

```
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE01a;"   \
"add $1, %%ebx;"   \
"add $4, %%esi;"   \
"jmp START01a;"   \
"DONE01a: nop;"   \
"mov %%ebx, %0;"   \
: "=b" (result_01arglen) \
);

void func_02(int a){}
```

## B.7 Sample C Header File: Format Function Parity Measurement

Listing B.7: Sample C Header File: Format Function Parity Measurement

```
int strSum(char *str){

//char *ptr = str;

int sum = 0,i=0;

while (str[i] != '\0'){

    sum = sum + (int)str[i++];

    }

return sum;

}
void memset_00(char *fmtstring){}
#define PostCallIntASMarglen00 asm volatile ("movl $0, %%ecx;"   \
"movl (%%esp), %%esi; "   \
"mov $0, %%ebx; "   \
"mov $0, %%edi; "   \
"add $1, %%ebx; "   \
"START00a: nop; "   \
"movl (%%esi), %%eax; "   \
"mov %%al, %%cl; "   \
"add %%ecx, %%edi; "   \
```

```
"cmp %%edx, %%edi; "   \
"jge DONE00a; "   \
"add $1, %%ebx; "   \
"movl $0, %%ecx; "   \
"mov %%ax, %%cx; "   \
"shr $8, %%ecx; "   \
"add %%ecx, %%edi; "   \
"cmp %%edx, %%edi; "   \
"jge DONE00a; "   \
"movl $0, %%ecx; "   \
"add $1, %%ebx; "   \
"mov %%eax, %%ecx; "   \
"and $16777215, %%ecx; "   \
"shr $16, %%ecx; "   \
"add %%ecx, %%edi; "   \
"cmp %%edx, %%edi; "   \
"jge DONE00a; "   \
"add $1, %%ebx; "   \
"movl $0, %%ecx; "   \
"mov %%eax, %%ecx; "   \
"shr $24, %%ecx; "   \
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE00a;"   \
"add $1, %%ebx;"   \
"add $4, %%esi;"   \
"jmp START00a;"   \
"DONE00a: nop;"   \
"mov %%ebx, %0;"   \
: "=b" (result_00arglen) \
);

void gets_10(char *fmtstring){}
#define PostCallIntASMarglen10 asm volatile ("movl $0, %%ecx;"   \
"movl (%%esp), %%esi; "   \
"mov $0, %%ebx; "   \
"mov $0, %%edi; "   \
"add $1, %%ebx; "   \
"START10a: nop; "   \
"movl (%%esi), %%eax; "   \
"mov %%al, %%cl; "   \
"add %%ecx, %%edi; "   \
"cmp %%edx, %%edi; "   \
"jge DONE10a; "   \
"add $1, %%ebx; "   \
```

```
"movl $0, %%ecx; "  \
"mov %%ax, %%cx; "  \
"shr $8, %%ecx; "  \
"add %%ecx, %%edi; "  \
"cmp %%edx, %%edi; "  \
"jge DONE10a; "  \
"movl $0, %%ecx; "  \
"add $1, %%ebx; "  \
"mov %%eax, %%ecx; "  \
"and $16777215, %%ecx; "  \
"shr $16, %%ecx; "  \
"add %%ecx, %%edi; "  \
"cmp %%edx, %%edi; "  \
"jge DONE10a; "  \
"add $1, %%ebx; "  \
"movl $0, %%ecx; "  \
"mov %%eax, %%ecx; "  \
"shr $24, %%ecx; "  \
"add %%ecx, %%edi;"  \
"cmp %%edx, %%edi;"  \
"jge DONE10a;"  \
"add $1, %%ebx;"  \
"add $4, %%esi;"  \
"jmp START10a;"  \
"DONE10a: nop;"  \
"mov %%ebx, %0;"  \
: "=b" (result_10arglen) \
);

void printf_20(char *fmtstring){}
#define PostCallIntASMarglen20 asm volatile ("movl $0, %%ecx;"  \
"movl (%%esp), %%esi; "  \
"mov $0, %%ebx; "  \
"mov $0, %%edi; "  \
"add $1, %%ebx; "  \
"START20a: nop; "  \
"movl (%%esi), %%eax; "  \
"mov %%al, %%cl; "  \
"add %%ecx, %%edi; "  \
"cmp %%edx, %%edi; "  \
"jge DONE20a; "  \
"add $1, %%ebx; "  \
"movl $0, %%ecx; "  \
"mov %%ax, %%cx; "  \
"shr $8, %%ecx; "  \
```

```
"add %%ecx, %%edi; "   \
"cmp %%edx, %%edi; "   \
"jge DONE20a; "   \
"movl $0, %%ecx; "   \
"add $1, %%ebx; "   \
"mov %%eax, %%ecx; "   \
"and $16777215, %%ecx; "   \
"shr $16, %%ecx; "   \
"add %%ecx, %%edi; "   \
"cmp %%edx, %%edi; "   \
"jge DONE20a; "   \
"add $1, %%ebx; "   \
"movl $0, %%ecx; "   \
"mov %%eax, %%ecx; "   \
"shr $24, %%ecx; "   \
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE20a;"   \
"add $1, %%ebx;"   \
"add $4, %%esi;"   \
"jmp START20a;"   \
"DONE20a: nop;"   \
"mov %%ebx, %0;"   \
: "=b" (result_20arglen) \
);
#define PostCallIntASMfmtstring20 asm volatile ("movl $0, %%ecx; "   \
"movl (%%esp), %%esi;"   \
"mov $0, %%ebx;"   \
"mov $0, %%edi;"   \
"START20f: nop;"   \
"movl (%%esi), %%eax;"   \
"mov %%al, %%cl;"   \
"add %%ecx, %%edi;"   \
"cmpl $37, %%ecx;"   \
"jne NOPCT120f;"   \
"add $1, %%ebx;"   \
"NOPCT120f: nop;"   \
"cmp %%edx, %%edi;"   \
"jge DONE20f;"   \
"movl $0, %%ecx;"   \
"mov %%ax, %%cx;"   \
"shr $8, %%ecx;"   \
"cmpl $37, %%ecx;"   \
"jne NOPCT220f;"   \
"add $1, %%ebx;"   \
```

```
"NOPCT220f: nop;"   \
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE20f;"   \
"movl $0, %%ecx;"   \
"mov %%eax, %%ecx;"   \
"and $16777215, %%ecx;"   \
"shr $16, %%ecx;"   \
"cmpl $37, %%ecx;"   \
"jne NOPCT320f;"   \
"add $1, %%ebx;"   \
"NOPCT320f: nop;"   \
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE20f;"   \
"movl $0, %%ecx;"   \
"mov %%eax, %%ecx;"   \
"shr $24, %%ecx;"   \
"cmpl $37, %%ecx;"   \
"jne NOPCT420f;"   \
"add $1, %%ebx;"   \
"NOPCT420f: nop;"   \
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE20f;"   \
"add $4, %%esi;"   \
"jmp START20f;"   \
"DONE20f: nop;"   \
"mov %%ebx, %0;"   \
: "=b" (result_20fmtstring) \
);

void printf_21(int a){}
#define PostCallIntASMfmtstring21 asm volatile ("movl $0, %%ecx; "   \
"movl (%%esp), %%esi;"   \
"mov $0, %%ebx;"   \
"mov $0, %%edi;"   \
"START21f: nop;"   \
"movl (%%esi), %%eax;"   \
"mov %%al, %%cl;"   \
"add %%ecx, %%edi;"   \
"cmpl $37, %%ecx;"   \
"jne NOPCT121f;"   \
"add $1, %%ebx;"   \
"NOPCT121f: nop;"   \
```

```
"cmp %%edx, %%edi;"   \
"jge DONE21f;"   \
"movl $0, %%ecx;"   \
"mov %%ax, %%cx;"   \
"shr $8, %%ecx;"   \
"cmpl $37, %%ecx;"   \
"jne NOPCT221f;"   \
"add $1, %%ebx;"   \
"NOPCT221f: nop;"   \
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE21f;"   \
"movl $0, %%ecx;"   \
"mov %%eax, %%ecx;"   \
"and $16777215, %%ecx;"   \
"shr $16, %%ecx;"   \
"cmpl $37, %%ecx;"   \
"jne NOPCT321f;"   \
"add $1, %%ebx;"   \
"NOPCT321f: nop;"   \
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE21f;"   \
"movl $0, %%ecx;"   \
"mov %%eax, %%ecx;"   \
"shr $24, %%ecx;"   \
"cmpl $37, %%ecx;"   \
"jne NOPCT421f;"   \
"add $1, %%ebx;"   \
"NOPCT421f: nop;"   \
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE21f;"   \
"add $4, %%esi;"   \
"jmp START21f;"   \
"DONE21f: nop;"   \
"mov %%ebx, %0;"   \
: "=b" (result_21fmtstring) \
);

void printf_22(char *mystring){}
#define PostCallIntASMarglen22 asm volatile ("movl $0, %%ecx;"   \
"movl (%%esp), %%esi; "   \
"mov $0, %%ebx; "   \
"mov $0, %%edi; "   \
```

```
"add $1, %%ebx; "   \
"START22a: nop; "   \
"movl (%%esi), %%eax; "   \
"mov %%al, %%cl; "   \
"add %%ecx, %%edi; "   \
"cmp %%edx, %%edi; "   \
"jge DONE22a; "   \
"add $1, %%ebx; "   \
"movl $0, %%ecx; "   \
"mov %%ax, %%cx; "   \
"shr $8, %%ecx; "   \
"add %%ecx, %%edi; "   \
"cmp %%edx, %%edi; "   \
"jge DONE22a; "   \
"movl $0, %%ecx; "   \
"add $1, %%ebx; "   \
"mov %%eax, %%ecx; "   \
"and $16777215, %%ecx; "   \
"shr $16, %%ecx; "   \
"add %%ecx, %%edi; "   \
"cmp %%edx, %%edi; "   \
"jge DONE22a; "   \
"add $1, %%ebx; "   \
"movl $0, %%ecx; "   \
"mov %%eax, %%ecx; "   \
"shr $24, %%ecx; "   \
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE22a;"   \
"add $1, %%ebx;"   \
"add $4, %%esi;"   \
"jmp START22a;"   \
"DONE22a: nop;"   \
"mov %%ebx, %0;"   \
: "=b" (result_22arglen) \
);
#define PostCallIntASMfmtstring22 asm volatile ("movl $0, %%ecx; "   \
"movl (%%esp), %%esi;"   \
"mov $0, %%ebx;"   \
"mov $0, %%edi;"   \
"START22f: nop;"   \
"movl (%%esi), %%eax;"   \
"mov %%al, %%cl;"   \
"add %%ecx, %%edi;"   \
"cmpl $37, %%ecx;"   \
```

```
"jne NOPCT122f;"   \
"add $1, %%ebx;"   \
"NOPCT122f: nop;"   \
"cmp %%edx, %%edi;"   \
"jge DONE22f;"   \
"movl $0, %%ecx;"   \
"mov %%ax, %%cx;"   \
"shr $8, %%ecx;"   \
"cmpl $37, %%ecx;"   \
"jne NOPCT222f;"   \
"add $1, %%ebx;"   \
"NOPCT222f: nop;"   \
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE22f;"   \
"movl $0, %%ecx;"   \
"mov %%eax, %%ecx;"   \
"and $16777215, %%ecx;"   \
"shr $16, %%ecx;"   \
"cmpl $37, %%ecx;"   \
"jne NOPCT322f;"   \
"add $1, %%ebx;"   \
"NOPCT322f: nop;"   \
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE22f;"   \
"movl $0, %%ecx;"   \
"mov %%eax, %%ecx;"   \
"shr $24, %%ecx;"   \
"cmpl $37, %%ecx;"   \
"jne NOPCT422f;"   \
"add $1, %%ebx;"   \
"NOPCT422f: nop;"   \
"add %%ecx, %%edi;"   \
"cmp %%edx, %%edi;"   \
"jge DONE22f;"   \
"add $4, %%esi;"   \
"jmp START22f;"   \
"DONE22f: nop;"   \
"mov %%ebx, %0;"   \
: "=b" (result_22fmtstring) \
);
```

# APPENDIX C

# PROFILER FOR TRUSTED CRITICAL INFRASTRUCTURE APPLICATIONS

## C.1 Client Code

Listing C.1: Client Code

```
/*Jon Jenkins 2012
 * p4client.c
 * p4 client: key agreement, multicast sv sample send
 * P4_REQ, P4_REP, P4_DATA
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netdb.h>
#include <ctype.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/un.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/select.h>
#include <time.h>

#define SA struct sockaddr

#define KDCAUTHPORT 8889
#define GKSPORT 8888
#define KDCTESTNAME "kdc"
#define KDCIP "127.0.0.1"
#define KDCNAME "noleptr4"

#define WELLKNOWNPORT 8888
```

```
#define ANOTHERWELLKNOWNPORT 8889
#define CLIREQACKPORT 8887
#define KRB_SERVICE "noleptr"
#define KRB_GKS_VERSION "KRB5_GKS_protocol_v0.1"
#define KRB_GKS_HOST "noleptr4.cs.fsu.edu"

#define P4TESTGRPIP "239.0.0.1"
#define P4TESTGRPPORT 5505

#include "p4defines.h"
#include "p4client.h"

//TPM
#include <tspi.h>
#include <trousers.h>
#include <tss.h>

//////////////////
//OpenSSL
#include <crypto.h>
#include <aes.h>
#include <sha.h>
#include <hmac.h>

////////////////
//Kerberos 5 api
#include <krb5.h>

#include "util.c"

int cli_SendN_Rel(int *sock,char *sendLine,int msgLen,char *ip,int port
    ){
int len = 0,bytesleft = 0,numsent = 0,curpos = 0;
int numrecvd = 0,resendCount = 0,expectedPos = 0;
char recvline[MSGLEN];
fd_set readset;
struct timeval timeout;
struct sockaddr_in senderAddr,dstAddr;
socklen_t adrlen;

bytesleft = msgLen;
len = bytesleft;

dstAddr.sin_family = AF_INET;
dstAddr.sin_port = htons(port);
```

```
dstAddr.sin_addr.s_addr = inet_addr(ip);

while (bytesleft > 0){
   adrlen = sizeof(dstAddr);
   while ((numsent = sendto(*sock, sendLine+curpos,(msgLen-curpos),
      0, (SA *)&dstAddr, adrlen)) < 0){
        if (errno != EINTR){
        printf("1: producer error sendto TGS_REQ: %s\n",strerror(errno)
            );
        return -1;
        }
      }

   printf("p4client sendto %d bytes\n",numsent);

   if (numsent>= msgLen){
     printf("p4client: sendto all bytes sent\n");
     break; //Full message sent. No ACK at this layer
     }

   FD_ZERO(&readset);    FD_SET(*sock, &readset);
   timeout.tv_sec = 0;   timeout.tv_usec = 10;
   resendCount = 0;

   //Wait for the ACK for 'timeout'
   while (select(*(sock)+1, &readset, NULL, NULL, &timeout) == 0) {
      adrlen = sizeof(dstAddr);
      if ((numsent = sendto(*sock, sendLine+curpos, (msgLen-curpos),
      0, (struct sockaddr *)&dstAddr, adrlen )) < 0){
          if (errno != EINTR){
            printf("producer error sendto TGS_REQ\n");
            return -1;
            }
        }

      resendCount++;
      FD_ZERO(&readset);    FD_SET(*sock, &readset);
      timeout.tv_sec = 0;   timeout.tv_usec = 10;
      }

    memset(recvline,'\0',MSGLEN);
    while ((numrecvd = recvfrom(*sock,recvline,MSGLEN,0,(SA *)&
       senderAddr,
                    (socklen_t*)sizeof(senderAddr))) < 0){
      if (errno != EINTR){
```

```
            printf("p4client err recvfrom ACK: %s\n",strerror(errno));
            return -1;
            }
        }

    if ((sscanf(recvline,"ACK %d",&expectedPos) == 1) && (expectedPos>
       curpos)){
       curpos = expectedPos;
       }
    else {printf("p4client: cli_TGS_REQ_Rel bad ACK\n"); return -1;}

   bytesleft = bytesleft - numsent;
   }

return 0;
}

//Pre: (loose) data holds data to write
//blob file convention: "p4tpmblob.tgc.sign" p4tpm.<entity>.<purpose>
int writeEncBlob(char *blobfilename,int datalen,BYTE *data){

        FILE *fptr;

        if ((fptr=fopen(blobfilename,"wb"))!=NULL){
                write(fileno(fptr),data,datalen);

                fclose(fptr);
        }

        return 0;
}

//Pre: blob allocated!
//blob file convention: "p4tpmblob.tgc.sign" p4tpm.<entity>.<purpose>
int loadEncBlob(char *blobfilename,int bloblen,BYTE *blob){

        FILE *fptr;

        if ((fptr=fopen(blobfilename,"rb"))!=NULL){
                read(fileno(fptr),blob,bloblen);

                fclose(fptr);
        }

        return 0;
```

```
}

//Pre: hash allocated
//tpm HMAC-SHA hash of hashdata, keyed with hmackey of length keylen,
    placed into hash
int p4TpmHMAC(BYTE *hmackey, int keylen, BYTE *hashdata,int hashdatalen
    ,BYTE *hash){

    int hashlen,res;
    BYTE *tmphash;

    ////////////////////////////////////////
    //TPM session is managed internally for Trspi
    res = Trspi_HMAC(TSS_HASH_SHA1, keylen, hmackey, hashdatalen,
        hashdata, hash);

    return 0;
}

//Seal an encrypted blob within
//TPM storage. Requires sealkey to decrypt. Optionally PCR statematch
//our use case: sealed data = key
//in: all
int p4TpmSeal(char blobname[], BYTE datatoseal[],int datalen){

        TSS_HCONTEXT hContext;
        TSS_HTPM hTPM;
        TSS_FLAG flg;
        TSS_BOOL fTpmState;
        TSS_RESULT tsr;
        TSS_HPOLICY srkpol,hSealkPol,tpmpol,newtpmpol;

        TSS_HKEY hSRK, hSealKey; //storage root key, sealing key handle
        TSS_HENCDATA hSealedData;
        TSS_HPCRS hPCRs, nullhPCRs;
        TSS_UUID srku = TSS_UUID_SRK;
        BYTE mylamesecret[] = {0,0,0,0,0,
                0,0,0,0,0,
                0,0,0,0,0,
                0,0,0,0,0};

        UINT32 unsealedDataLen,pcrLen;
        BYTE *unsealedData,*pcrVal;
        TSS_HPOLICY hPolicy, srkUsagePolicy;
```

```
////////////////////////////////
//Create context, connect to TPM
Tspi_Context_Create(&hContext);
Tspi_Context_Connect(hContext,NULL);//NULL: local, or
   TSS_UNICODE str for remote

////Get the TPM handle//////////
tsr=Tspi_Context_GetTpmObject(hContext, &hTPM);
printf("getting tpm object: %s\n",Trspi_Error_String(tsr) );


//////////////////////////////////////////////////////////
////Task: seal to PCRs: hash TPM state hashes, write to PCRs
//////////////////////////////////////////////////////////

//////////////////////////////////////////////////
////Create objects.. (all objects bound to the context)

//data object for sealed data
tsr = Tspi_Context_CreateObject(hContext,
   TSS_OBJECT_TYPE_ENCDATA,TSS_ENCDATA_SEAL,&hSealedData);
printf("create data obj test: %s\n",Trspi_Error_String(tsr) );

//..and the pcr composite object
//printf("create pcr comp obj test: %s\n",
//                 Trspi_Error_String(tsr =
    Tspi_Context_CreateObject(hContext,
//                                    TSS_OBJECT_TYPE_PCRS,
//                                    TSS_PCRS_STRUCT_DEFAULT,
//                                    &hPCRs)));


////Load SRK from TPM (create object insufficient)//
tsr = Tspi_Context_LoadKeyByUUID(hContext, TSS_PS_TYPE_SYSTEM,
   srku, &hSRK);
printf("create srk obj test: %s\n",Trspi_Error_String(tsr) );

tsr = Tspi_GetPolicyObject(hSRK, TSS_POLICY_USAGE, &srkpol);
printf("get srkpolobj: %s\n",Trspi_Error_String(tsr) );
tsr = Tspi_Policy_SetSecret(srkpol, TSS_SECRET_MODE_SHA1, 20,
                mylamesecret);
printf("setsrksec: %s\n",Trspi_Error_String(tsr) );
//////////////////////////////////////////////////
```

```
//////////////////////////////////////////////
////PCR values into composite, seal
//printf("pcrread1 test: %s\n",Trspi_Error_String(
//    tsr=Tspi_TPM_PcrRead(hTPM, 5, &pcrLen, &pcrVal) )); //pull
    ..
//printf("pcrset1 test: %s\n",Trspi_Error_String(
//    tsr=Tspi_PcrComposite_SetPcrValue(hPCRs, 5, pcrLen, pcrVal
    )));//and store pcr x


//seal (encrypt) key using the sealing key
//null pcrobj means no required match upon unseal
nullhPCRs = (TSS_HPCRS)NULL;

tsr = Tspi_Data_Seal(hSealedData,hSRK/*hSealKey*/,datalen,
   datatoseal,nullhPCRs);
printf("seal data test: %s %s %u\n",Trspi_Error_String(tsr),
                Trspi_Error_Layer(tsr), Trspi_Error_Code(tsr) )
                    ;


//////////////////////////////////////////////////
//Extract data blob, write to file for later "access"
BYTE *encData; //assume API allocates
int encDLen = P4KEYSIZE;
//memset(encData,'\0',P4KEYSIZE);

tsr = Tspi_GetAttribData( hSealedData,
                TSS_TSPATTRIB_ENCDATA_BLOB,
                TSS_TSPATTRIB_ENCDATABLOB_BLOB,
                &encDLen,
                &encData);

writeEncBlob(blobname,encDLen,encData);
//////////////////////////////////////////////////


//Tspi_Key_UnloadKey(hSealKey);
//Tspi_Context_CloseObject(hContext, hPCRs);
//Tspi_Context_CloseObject(hContext, hSealKey);
Tspi_Context_CloseObject(hContext, hSealedData);


//////////////////////////////////////////
```

```
        //Finalize
        Tspi_Context_Close(hContext);
        printf("Context Closed. TPM session over\n");

        return 0;
}

//Pre: outdata allocated and nulled!
//unseal an encrypted blob (here, a key) which was sealed
//to the TPM storage
int p4TpmBlobUnseal(char *blobfile,int *outdatalen,BYTE *outdata){

        TSS_HCONTEXT hContext;
        TSS_HTPM hTPM;
        TSS_FLAG flg;
        TSS_BOOL fTpmState;
        TSS_RESULT tsr;
        TSS_HKEY hSRK; //storage root key, sealing key handle
        TSS_HPCRS hPCRs, nullhPCRs;
        TSS_UUID srku = TSS_UUID_SRK;

        UINT32 unsealedDataLen,pcrLen;
        BYTE *unsealedData,*pcrVal;

        ////////////////////////////////////
        //Retrieve enc blob for sealed data
        int bloblen = P4KEYSIZE;
        BYTE *blob = (BYTE *)malloc(sizeof(BYTE)*P4KEYSIZE);

        loadEncBlob(blobfile,bloblen,blob);
        ////////////////////////////////////


        ////////////////////////////////////
        //Create context, connect to TPM
        Tspi_Context_Create(&hContext);
        Tspi_Context_Connect(hContext,NULL);//NULL: local, or
           TSS_UNICODE str for remote

        ////Get the TPM handle//////////
        tsr=Tspi_Context_GetTpmObject(hContext, &hTPM);
        printf("getting tpm object: %s\n",Trspi_Error_String(tsr) );
        ////////////////////////////////////
```

```
//////////////////////////////
//Recover blob data into object
TSS_HENCDATA hRecoveredData;

tsr = Tspi_Context_CreateObject(hContext,
                TSS_OBJECT_TYPE_ENCDATA,
                TSS_ENCDATA_SEAL,
                &hRecoveredData);
printf("create recovereddata obj: %s\n",Trspi_Error_String(tsr)
    );

tsr = Tspi_SetAttribData(hRecoveredData,
                TSS_TSPATTRIB_ENCDATA_BLOB,
                TSS_TSPATTRIB_ENCDATA_BLOB,
                bloblen,blob);
printf("set attrib blob into ENCDATA: %s\n",Trspi_Error_String(
   tsr) );
//////////////////////////////


////////////////////////////////////////////////
//Load SRK from TPM (create object insufficient)//
//Currently using SRK only as sealing key
tsr = Tspi_Context_LoadKeyByUUID(hContext, TSS_PS_TYPE_SYSTEM,
   srku, &hSRK);
printf("create srk obj test: %s\n",Trspi_Error_String(tsr) );

//unneeded?//////////////////////////
//tsr = Tspi_GetPolicyObject(hSRK, TSS_POLICY_USAGE, &srkpol);
//printf("get srkpolobj: %s\n",Trspi_Error_String(tsr) );
//tsr = Tspi_Policy_SetSecret(srkpol, TSS_SECRET_MODE_SHA1, 20,
//                mylamesecret);
//printf("setsrksec: %s\n",Trspi_Error_String(tsr) );
////////////////////////////////////////////////


////////////////////////////////////////////
//N.B. Read PCR not necessary on unseal (tpm internal guarantee
    of
//data/state link)
////////////////////////////////////////////


//decrypts data only if PCR's match seal config (unless PCRs
   were not linked at seal)
```

```
        tsr = Tspi_Data_Unseal(hRecoveredData, hSRK/*hSealKey*/,
            outdatalen, &outdata);
        printf("unseal data test: %s %s %u\n",Trspi_Error_String(tsr),
            Trspi_Error_Layer(tsr),Trspi_Error_Code(tsr) );

        ////////////////////////////////////////
        //Finalize

        //Tspi_Key_UnloadKey(hSealKey);
        //Tspi_Context_CloseObject(hContext, hPCRs);
        //Tspi_Context_CloseObject(hContext, hSealKey);

        Tspi_Context_Close(hContext);
        printf("Context Closed. TPM session over\n");
        ////////////////////////////////////////

        return 0;
}

//openSSL: decrypt the ciphertext using the encryption key,IV into
    ptext
int p4OpensslDecAES(UCHR *ctext,int *ctextlen,UCHR *encKey,UCHR *encIV,
    UCHR *ptext){
////max ciphertext len for a n bytes of plaintext is n + AES_BLOCK_SIZE
    -1 bytes
int lastlen = 0;
int ptextlen = *ctextlen;
int finalPtextLen = 0;

EVP_CIPHER_CTX decCtxt;

////////////////////////////////////////////
//Openssl decryption cipher init

EVP_CIPHER_CTX_init(&decCtxt);
EVP_DecryptInit_ex(&decCtxt, EVP_aes_256_cbc(), NULL, encKey, encIV);

EVP_DecryptInit_ex(&decCtxt, NULL, NULL, NULL, NULL);

EVP_DecryptUpdate(&decCtxt, ptext, &ptextlen, ctext, *ctextlen);
EVP_DecryptFinal_ex(&decCtxt, ptext+ptextlen, &finalPtextLen);

*ctextlen = ptextlen + finalPtextLen;

//Openssl finalize
```

```
EVP_CIPHER_CTX_cleanup(&decCtxt);

return 0;
}


//ctext must be large enough to hold enc{ptext}
int p4OpensslEncAES(UCHR *ptext,int ptextlen,UCHR *encKey,UCHR *encIV,
    UCHR *ctext){
//max ciphertext len for a n bytes of plaintext is n + AES_BLOCK_SIZE
    -1 bytes
int ctextlen = ptextlen + AES_BLOCK_SIZE, finallen = 0;
int lastlen = 0;
UCHR *ctextTmp = malloc(ctextlen);

////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
////Openssl: Encrypt ptext using encKey into ctext///////

////////////////////////////////////////////////////////////////
////Openssl cipher init
EVP_CIPHER_CTX enc_contxt;//openssl managerial sec op contexts
EVP_CIPHER_CTX_init(&enc_contxt);

//Init encryption: use symmetric key of group member
EVP_EncryptInit_ex(&enc_contxt, EVP_aes_256_cbc(), NULL,encKey,encIV);
EVP_EncryptUpdate(&enc_contxt, ctextTmp, &ctextlen,ptext, ptextlen);
EVP_EncryptFinal_ex(&enc_contxt, ctextTmp+ctextlen, &finallen);

lastlen = ctextlen + finallen;

memcpy(ctext,ctextTmp,lastlen);

//Openssl finalize
EVP_CIPHER_CTX_cleanup(&enc_contxt);

//free(ctextTmp);

return 0;
}

int cli_Send_AP_REP(int socket,char *clientPrincStr,UCHR *encKey,char *
    absGrpNameStr){
char sendLine[MSGLEN];
int numsent = 0;
```

```
struct timespec ts;

UCHR encPrincStr[P4PRINCIPALNMLEN+AES_BLOCK_SIZE];
UCHR encTimeStr[32+AES_BLOCK_SIZE];
UCHR timeValStr[500];

memset(encPrincStr,'\0',(P4PRINCIPALNMLEN+AES_BLOCK_SIZE));
memset(encTimeStr,'\0',(32+AES_BLOCK_SIZE));
memset(sendLine,'\0',MSGLEN);
memset(timeValStr,'\0',500);

if (clock_gettime( CLOCK_MONOTONIC, &ts))
   printf("cli_send_ap_rep; err clock_gettime\n");

sprintf(timeValStr,"%ld",ts.tv_nsec);

p4OpensslEncAES(timeValStr,strlen(timeValStr),encKey,iv_Dummy,
   encTimeStr);
p4OpensslEncAES(clientPrincStr,strlen(clientPrincStr),encKey,iv_Dummy,
   encPrincStr);

sprintf(sendLine,"AP_REP %s %s %s",absGrpNameStr,encPrincStr,encTimeStr
   );

if ((numsent = sendn(socket,sendLine,strlen(sendLine)))<0){
   printf("p4client err sendn: %dB sent\n",numsent);
   return -1;
   }

printf("p4client sent (%dB): %s\n",numsent,sendLine);

return 0;
}

//send TGS_REQ////////////////////////////////
//case 2: new group
int cli_TGS_REQ2(int socket,char *grpName,int grpSize,char *
   grpPrincipalList,UCHR *hash,char *clientPrinc){
char sendLine[MSGLEN];
int numsent = 0;

memset(sendLine,'\0',MSGLEN);

//IGNORE hash for now
```

```
sprintf(sendLine,"TGS_REQ %s %d %s %s",clientPrinc,grpSize,grpName,
   grpPrincipalList);

while ((numsent = send(socket,sendLine,MSGLEN,0)) < 0){
    if (errno != EINTR){
      printf("client: error sending P4_REQ\n");
      continue;
      }
   }
printf("p4client sent %s\n",sendLine);

return 0;
}

int cli_Send_TGS_REQ_Rel(struct sockaddr_in *tgsAddr,char *grpName,int
   reqGrpSize,char *grpPrincipalList,UCHR *hash,char *clientPrinc){
int sock = 0,len = 0,bytesleft = 0,numsent = 0,curpos = 0;
int numrecvd = 0,resendCount = 0,expectedPos = 0;
char sendline[MSGLEN],recvline[MSGLEN];
fd_set readset;
struct timeval timeout;
struct sockaddr_in senderAddr;
socklen_t adrlen;

memset(sendline,'\0',MSGLEN);

sprintf(sendline,"TGS_REQ %s %d %s %s",clientPrinc,reqGrpSize,grpName,
   grpPrincipalList);

bytesleft = strlen(sendline);
len = bytesleft;

if ((sock = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP)) < 0){
   printf("p4 cli: can't create UDP socket. Aborting\n");
   return -1;
   }

while (bytesleft > 0){
   adrlen = sizeof(tgsAddr);
   if ((numsent = sendto(sock, sendline+curpos,MSGLEN,
      0, (struct sockaddr *)tgsAddr,adrlen)) < 0){
        if (errno != EINTR){
        printf("producer error sendto TGS_REQ\n");
        return -1;
        }
```

```
    }

 FD_ZERO(&readset);
 FD_SET(sock, &readset);
 //timeout.tv_sec = 0;
 //timeout.tv_usec = 10;
resendCount = 0;

//Wait for the ACK, //blocking select: time null
while (select(sock+1, &readset, NULL, NULL, NULL/*&timeout*/) == 0)
    {
    adrlen = sizeof(tgsAddr);
    if ((numsent = sendto(sock, sendline+curpos, MSGLEN,
    0, (struct sockaddr *)tgsAddr, adrlen )) < 0){
      if (errno != EINTR){
      printf("producer error sendto TGS_REQ\n");
      return -1;
      }
    }

    resendCount++;
    FD_ZERO(&readset);
    FD_SET(sock, &readset);
    //timeout.tv_sec = 0;
    //timeout.tv_usec = 10;
    }

 memset(recvline, '\0',MSGLEN);
 if ((numrecvd = recvfrom(sock,recvline,MSGLEN,0,(SA *)&senderAddr,
                  (socklen_t*)sizeof(senderAddr))) < 0){
     if (errno != EINTR){
       printf("p4client err recvfrom ACK: %s\n",strerror(errno));
       return -1;
       }
    }

 if ((sscanf(recvline,"ACK %d",&expectedPos) == 1) && (expectedPos>
    curpos)){
    curpos = expectedPos;
    }
 else {printf("p4client: cli_TGS_REQ_Rel bad ACK\n"); return -1;}

bytesleft = bytesleft - numsent;
}
```

```
return 0;
}


int cli_Recv_TGS_REP(int socket,UCHR *encGrpKey,char *encGrpName,UCHR *
    encLongTrmGKSessGK){
int numrecvd = 0,ret = 0;
char recvline[MSGLEN];

memset(recvline,'\0',MSGLEN);

while ((numrecvd = recv(socket,recvline,MSGLEN,0))<0){
    if (errno != EINTR){
        printf("p4client: error recv TGS_REP: %s\n",strerror(errno));
        //break;//return -1;
        }
    continue;
    }

printf("p4client (TGS_REP) recvd: %s\n",recvline);
if ((ret = sscanf(recvline,"TGS_REP %s  %s  %s",encGrpKey,encGrpName,
    encLongTrmGKSessGK)) < 3){
    printf("p4client: parse fail on TGS_REP\n");
    return -1;
    }

return 0;
}

int cli_Mcast_Send_GRP_AP_REQ(char *grpName,UCHR *tkt_encLtgkSgk){
int grpsock,numsent;
struct sockaddr_in mcastaddr;
char sendline[MSGLEN];

////////////////////////////////////////////////////////
//Create multicast group socket//////////////////////////
if ((grpsock = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP)) < 0){
    printf("p4 cli/producer: can't create socket. Aborting\n");
    return -1;
    }

////////////////////////////////////////////////
//Multicast session key Ticket to group

mcastaddr.sin_family = AF_INET;
```

```
mcastaddr.sin_port = htons(P4TESTGRPPORT);

if (P4MCAST){
   mcastaddr.sin_addr.s_addr=inet_addr(P4TESTGRPIP);
   }
else {
   mcastaddr.sin_addr.s_addr = inet_addr(LOCALIP);
   }

memset(sendline,'\0',MSGLEN);
sprintf(sendline,"GRP_AP_REQ %s %s",grpName,tkt_encLtgkSgk);


//cli_SendN_Rel(&mcastaddr,&grpsock,sendline,strlen(sendline));
cli_SendN_Rel(&grpsock,sendline,strlen(sendline),LOCALIP,P4TESTGRPPORT)
   ;

/*
while ((numsent = sendto(grpsock,sendline,strlen(sendline)+1,0,(SA *)&
   mcastaddr,sizeof(mcastaddr))) < 0){
   if (errno != EINTR){
      printf("producer error mcast sendto\n");
      return -1;
      }
   }
*/

printf("P4 client/producer sent %s\n",sendline);

////////////////////////////////////////////////

close(grpsock);

return 0;
}

cli_Recv_Mcast_AP_REP(int socket,char *grpName,char *encPrincNm,char *
   encTimestamp){
char recvline[MSGLEN];
int numrecvd,recvSock,connfd;
struct sockaddr_in ownaddr;

memset(recvline,'\0',MSGLEN);

while ((numrecvd = recv(socket,recvline,MSGLEN,0)) < 0){
```

```
    if (errno != EINTR){
       printf("p4client: error recv mcast ap_rep\n");
       //close(kdcsock);
       return -1;
       }
    }

sscanf(recvline,"AP_REP %s %s %s",grpName,encPrincNm,encTimestamp);

printf("P4 client recvd %s\n",recvline);

return 0;
}

//Wait for all grp members to send AP_REP after mcast SGK dist
cli_Recv_Mcast_AP_REP_All(int grpSize,char *recvdGrpName){

int rcvAPREPsock,connfd,i,numAck,len;
struct sockaddr_in ownaddr,rcvrAddr;
UCHR encPrincNm[P4PRINCIPALNMLEN],encTimestamp[MSGLEN];

memset(encPrincNm,'\0',P4PRINCIPALNMLEN);
memset(encTimestamp,'\0',MSGLEN);

rcvAPREPsock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

ownaddr.sin_family = AF_INET;
ownaddr.sin_port = htons(CLIREQACKPORT);
ownaddr.sin_addr.s_addr=htonl(INADDR_ANY);

if (bind(rcvAPREPsock,(SA *)&ownaddr,sizeof(ownaddr)) < 0){
     printf("p4cli: err bind for mcast AP_REP. Abort\n");
     return 1;
     }

if (listen(rcvAPREPsock,LISTEN_Q) < 0){
    printf("p4cli: err listen for mcast AP_REP. Abort\n");
    return 1;
    }

numAck = 0;

//wait for all proposed group members to ACK mcast msg with AP_REP
while (numAck<grpSize){
```

```
    memset(&rcvrAddr,0,sizeof(rcvrAddr));
    len = sizeof(rcvrAddr);
    //Await mcast reciever connection
    if ((connfd = accept(rcvAPREPsock,(SA *)&rcvrAddr,&len)) < 0){
      printf("p4client mcast AP_REP accept error\n");
      return -1;
      }

    printf("p4client: wait for AP_REP from any receiver %d\n",numAck);
    cli_Recv_Mcast_AP_REP(connfd,recvdGrpName,encPrincNm,encTimestamp);

    close(connfd);

    numAck++;
    }

close(rcvAPREPsock);

return 0;
}

int main(int argc,char *argv[]){

        krb5_ap_rep_enc_part *rep_ret;
        krb5_auth_context auth_context = 0;
        krb5_ccache ccdef;
        krb5_context context;
        krb5_data cksum_data;
        krb5_data recv_data;
        krb5_error *err_ret;
        krb5_error_code retval;
        krb5_principal p4gkclient, p4gkserver;
        krb5_keyblock * sKey; /* Used as the long term shared key */

        char **adrlist_pptr;
        char kdcIP[16],str[INET_ADDRSTRLEN],protomsg[MSGLEN],recvline[
           MSGLEN],
         sendline[MSGLEN],clihost[MSGLEN];
        char grpName[P4GRPNAMELEN],grpPrinc1[P4PRINCIPALNMLEN],
           grpPrinc2[P4PRINCIPALNMLEN],msgtype[50];
        char tgsPrincList[MSGLEN];
        UCHR *tgsGrpKeyEnc;
        char *clientPrincipalStr, *strictGrpName,*tgsAbsGrpName;
        UCHR cliTGSSharedKey[KRBKEYSIZE];
        fd_set rfdset,readset; //potential select call...
```

```
int i, kdcsock,grpsock,ret,numsent,numrecvd,reqGrpSize;
struct hostent *kdchost;
struct hostent *ownhost;
struct ip_mreq mrq; //IP multicast request addr structure
struct sockaddr_in kdc, mcastaddr;
struct sockaddr_in ownaddr;
struct timeval timeout;
UCHR *hashres_p,*digeststr;
UCHR hashout[SHA1OUTSIZE];
UCHR hashresult[SHA1OUTSIZE];
UCHR us_protomsg[MSGLEN];
UCHR encLongTrmGKSessGK[P4KEYSIZE+AES_BLOCK_SIZE],tgsEncGK[
    P4KEYSIZE+AES_BLOCK_SIZE],
  tgsEncGrpName[P4GRPNAMELEN+AES_BLOCK_SIZE];
unsigned int len;

kdchost = (struct hostent *)malloc(sizeof(struct hostent));
kdcsock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

memset(clihost,'\0',MSGLEN);
memset(&kdc,0,sizeof(kdc));
if (gethostname(clihost,MSGLEN) < 0){
        printf("error gethostname\n");
    }

////////////////////////////////////////////
//Obtain kdc host info (Bypass if static IP)


if (LOCALTESTING){//TESTING
    strncpy(kdcIP,KDCIP,INET_ADDRSTRLEN);
    }
else {
  if ((kdchost = gethostbyname(KRB_GKS_HOST)) == NULL){
    printf("error gethostbyname: %s %s\n",KDCNAME,strerror(
       h_errno));
    return 1;
    }

    if (kdchost->h_addrtype == AF_INET){
      adrlist_pptr = kdchost->h_addr_list;
      memset(kdcIP,'\0',INET_ADDRSTRLEN);
      strcpy(kdcIP,inet_ntop(kdchost->h_addrtype, *adrlist_pptr,
          str, sizeof(str)));
      printf("client: kdc IP: %s\n",kdcIP);
```

```
        }
      else {
        printf("error gethostbyname: unexpected address type!.
           Abort");
        return -1;
        }
     }

    //writes kdcip into socket addr struct in nwbyteorder
    if(inet_pton(AF_INET,kdcIP,&kdc.sin_addr) <= 0){
        printf("inet_pton error\n");
        return 1;
     }
    ////////////////////////////////////////
    ////////////////////////////////////////
    //TSS API: Seal principal-member/kdc(tgs) key?

    ////////////////////////////////////////

//BEGIN P4 auth request (SENDAUTH)///////////////
kdc.sin_family = AF_INET;
    kdc.sin_port = htons(KDCAUTHPORT);


    ////////////////////////////////////////////
    //connect to kdc: AUTH//////////////////////////
    while ((ret = connect(kdcsock,(SA *)&kdc,sizeof(kdc))) < 0){
       if (errno == ETIMEDOUT){
        printf("p4client: kdc connect timed out\n");
        return 1;
        }
       else if (errno == ECONNREFUSED){
        printf("p4client: kdc connection refused!\n");
        return 1;
        }
       else
           return 1;
       }

    printf("client: connected to KDC\n");
    ////////////////////////////////////////////


    ////////////////////////////////////////////
    //BEGIN Kerberos client SENDAUTH/////////////
```

```
if (P4KRB) {

retval = krb5_init_context(&context);
if (retval) {
        perror("ERROR when initializing krb5\n");
        return 1;
}

retval = krb5_cc_default(context, &ccdef);//TODO fails here
if (retval) {
        perror("ERROR getting default cred cache\n");
        return 1;
}

retval = krb5_cc_get_principal(context, ccdef, &p4gkclient);
if (retval) {
        perror("ERROR getting cred cache defaul princ\n");
        return 1;
}

retval = krb5_sname_to_principal(context, KRB_GKS_HOST,
   KRB_SERVICE,
                                 KRB5_NT_SRV_HST, &p4gkserver);
if(retval)
{
        perror("ERROR while generating server princ name");
        return 1;
}

retval = krb5_sendauth(context, &auth_context, (krb5_pointer) &
   kdcsock,
                KRB_GKS_VERSION, p4gkclient, p4gkserver,
                AP_OPTS_MUTUAL_REQUIRED,
                &cksum_data,
                0,          // no creds, use ccache instead /
                ccdef, &err_ret, &rep_ret, NULL);
if (retval)
{
        printf("p4client: err krb5_sendauth: %s \n",
               krb5_get_error_message(context,retval));
        return 1;
}

retval = krb5_auth_con_getkey(context, auth_context, &sKey);
```

```
        //TODO : COPY KEY OFF HERE -Sean

        //memcpy(cliTGSSharedKey,0,KRBKEYSIZE);
        memcpy(cliTGSSharedKey,sKey->contents,sKey->length);

        retval = krb5_unparse_name(context,p4gkclient,&
            clientPrincipalStr);

        printf("%s key[",clientPrincipalStr);
        for(i = 0; i < sKey->length; ++i)
        {

                printf("\\x%X",(UCHR)cliTGSSharedKey[i]);
        }
        printf("]\n");


        krb5_free_keyblock(context, sKey);
printf("##\n");
        krb5_free_principal(context, p4gkserver);        // finished
            using it /
        krb5_free_principal(context, p4gkclient);
        krb5_cc_close(context, ccdef);
        krb5_free_context(context);
        }

        close(kdcsock);

        //END Kerberos client sendauth (CS, AP_REX)///
        ////////////////////////////////////////////


        ////////////////////////////////////////////
        //TSS API: Seal principal-member/kdc(tgs) key?

        ////////////////////////////////////////////


        //END P4 auth request (SENDAUTH)//////////
        ////////////////////////////////////////////


        ////////////////////////////////////////////
        //BEGIN P4 Group Key Update Request////////
```

```
        //
        kdcsock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
        kdc.sin_port = htons(GKSPORT);//port for [group key u/d] ONLY

        ////////////////////////////////////////////
        //connect to kdc//////////////////////////
        while ((ret = connect(kdcsock,(SA *)&kdc,sizeof(kdc))) < 0){
                if (errno == ETIMEDOUT){
                        printf("p4client: kdc connect timed out\n");
                        return 1;
                }
                else if (errno == ECONNREFUSED){
                        printf("p4client: kdc connection refused!\n");
                        return 1;
                }
                else
                        return 1;
        }
        ////////////////////////////////////////////

        printf("connected to TGS\n");

        ////////////////////////////////////////////
        //Form TGS_REQ Grp ///////////////////////
        memset(hashout,'\0',SHA1OUTSIZE);
        memset(us_protomsg,'\0',MSGLEN);
        sprintf(us_protomsg,"%d %s",TESTGRPID,clihost);
digeststr = (UCHR *)malloc(sizeof(UCHR)*MSGLEN);
strictGrpName = (char *)malloc(sizeof(char)*P4GRPNAMELEN);
tgsGrpKeyEnc = (UCHR *)malloc(sizeof(UCHR)*(P4KEYSIZE+AES_BLOCK_SIZE
    ));
tgsAbsGrpName = (char *)malloc(sizeof(char)*P4GRPNAMELEN);

memset(strictGrpName,'\0',P4GRPNAMELEN);
memset(digeststr,'\0',MSGLEN);
memset(grpName,'\0',P4GRPNAMELEN);
memset(grpPrinc1,'\0',P4PRINCIPALNMLEN);
memset(grpPrinc2,'\0',P4PRINCIPALNMLEN);
len = SHA1OUTSIZE;
memset(recvline,'\0',MSGLEN);
memset(msgtype,'\0',50);
memset(strictGrpName,'\0',P4GRPNAMELEN);
        memset(tgsAbsGrpName,'\0',P4GRPNAMELEN);
        memset(tgsPrincList,'\0',MSGLEN);
        memset(tgsGrpKeyEnc,'\0',(P4KEYSIZE+AES_BLOCK_SIZE));
```

```
        memset(encLongTrmGKSessGK, '\0',P4KEYSIZE+AES_BLOCK_SIZE);
        memset(tgsEncGK, '\0',P4KEYSIZE+AES_BLOCK_SIZE);
        memset(tgsEncGrpName, '\0',P4GRPNAMELEN+AES_BLOCK_SIZE);

        ////////////////////////////////////////
        //auth. P4_REQ with cli/KDC shkey, append

        //HMAC-SHA1, with tgc/cli shkey Digest longer than input
        //TODO replace:tpm hash?
        //if ( (hashres_p = HMAC(EVP_sha1(),(const void *)
            cliTGSSharedKey,P4KEYSIZE,us_protomsg,
        //  strlen(us_protomsg),
        //  hashout,&len)) == NULL){
        //      printf("hmac error\n");
        //      return -1;}

        ////////////////////////////////////////////
        //TPM HMAC SHA keyed, using tgc/cli shared key
        //p4TpmHMAC((BYTE *)k_Dummy,P4KEYSIZE, (BYTE *)us_protomsg,
            strlen(us_protomsg),
                //      (BYTE*)hashout);

        ////////////////////////////////////////////////


        ////////////////////////////////////////////
        //send TGS_REQ//////////////////////////////
strcpy(grpName,"NEWGRP");
strcpy(grpPrinc1,"rcvr");
strcpy(grpPrinc2,"jon");
reqGrpSize = 1;

//getHexStr(hashout,strlen(hashout),digeststr);
//case 2: new group
cli_TGS_REQ2(kdcsock,grpName,reqGrpSize,grpPrinc1,digeststr,
    grpPrinc2);

////////////////////////////////////////////////////////
//await either AP_REQ (new group) or TGS_REP (existing group)

while ((numrecvd = recv(kdcsock,recvline,MSGLEN,0)) < 0){
    if (errno != EINTR){
        printf("client: error sending P4_REQ\n");
        close(kdcsock);
        continue;
```

```
        }
      }

sscanf(recvline,"%s %*s",msgtype);

printf("P4 client recvd %s\n",recvline);

if (!strcmp(msgtype,"TGS_REP")){
      ;//do nothing?
      }
else if (!strcmp(msgtype,"AP_REQ")){
     //recvd AP_REQ <grpname grpprincipallist> k_cli/tgs{groupkey}

     sscanf(recvline,"AP_REQ %*s %s",tgsGrpKeyEnc);
     sscanf(recvline,"AP_REQ %s %*s",tgsAbsGrpName);
     //sscanf(tgsAbsGrpName,"APREQ %s %*s",strictGrpName);

     //only reply if(clientPrinc in tgsAbsGrpName) //TODO

     //send AP_REP clientname k_cli/tgs{groupkey}
     //TODO need i/TGS shkey instead of k_Dummy
     cli_Send_AP_REP(kdcsock,grpPrinc2,k_Dummy,tgsAbsGrpName);

     cli_Recv_TGS_REP(kdcsock,tgsEncGK,tgsEncGrpName,
         encLongTrmGKSessGK);

     //mcast <grpname> LTGK{SGK} to group
     cli_Mcast_Send_GRP_AP_REQ(grpName,encLongTrmGKSessGK);

     char recvdGrpNm[P4GRPNAMELEN];
     memset(recvdGrpNm,'\0',P4GRPNAMELEN);

     cli_Recv_Mcast_AP_REP_All(reqGrpSize,recvdGrpNm);
     }
else {
     printf("P4Client: err: bad TGS reply. Abort\n");
     return -1;
     }

close(kdcsock);
//P4 group key update request END//////////////////////////////
////////////////////////////////////////////////////////////////


if (P4KRB){krb5_free_unparsed_name(context,clientPrincipalStr);}
```

```
    return 0;
}
```

## C.2 Client Header

Listing C.2: Client Header

```
#ifndef _P4CLIENT_H_
#define _P4CLIENT_H_

#define MINHEADERLEN 15
#define MAXHEADERLEN 500

int recvn(int sockfd,char *dest,int bytesToGet){
int chunk = 0;
int bytesleft = bytesToGet,numrecvd = 0;
char *ptr;

ptr = dest;

while (bytesleft > 0){
  chunk = MSGLEN;
  if (bytesleft < MSGLEN)
     {chunk = bytesleft;}
  if ((numrecvd = recv(sockfd,ptr,chunk,0)) < 0){
     if (errno == EINTR)
       numrecvd = 0;
     else {
       printf("client: recvn recieve error\n");
       return -1;
       }
     }
  else if (numrecvd == 0)
     break;

  ptr += numrecvd;
  bytesleft -= numrecvd;
  }

return (bytesToGet-bytesleft);
}

int sendn(int sockfd,char *buf,int bytesToSend){
```

```
int bytesleft = bytesToSend,numsent = 0;
char *ptr;

ptr = buf;

while (bytesleft > 0)
  {
  if ((numsent = send(sockfd,ptr,bytesleft,0)) <= 0){
     if ((numsent < 0) && (errno == EINTR)){
       numsent = 0;
       }
     else {
       printf("client: sendn error\n");
       return -1;
       }
     }

  bytesleft -= numsent;
  ptr += numsent;
  }
return (bytesToSend-bytesleft);
}


#endif
```

## C.3 Client Receiver Code

Listing C.3: Client Receiver Code

```
/*Jon Jenkins 2012
 * p4client.c
 * p4 client: key agreement,multicast sv sample send
 * P4_REQ,P4_REP,P4_DATA
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netdb.h>
#include <ctype.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/un.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/select.h>
#include <time.h>

#define SA struct sockaddr

#define KDCPORT 5501
#define KDCNAME "noleptr4.cs.fsu.edu" //
#define KDCIP "127.0.0.1" //TODO
#define KDCAUTHPORT 8889
#define CLIREQACKPORT 8887

#define WELLKNOWNPORT 8888
#define ANOTHERWELLKNOWNPORT 8889
#define KRB_SERVICE "noleptr"
#define KRB_GKS_VERSION "KRB5_GKS_protocol_v0.1"
#define KRB_GKS_HOST "noleptr4.cs.fsu.edu"

#define P4TESTGRPIP "239.0.0.1"
#define P4BCASTIP "225.0.0.255"
#define P4TESTGRPPORT 5505

#include "p4defines.h"
#include "p4client.h"

#include <tspi.h>
#include <trousers.h>
#include <tss.h>
//////////////////
//OpenSSL
#include <crypto.h>
#include <aes.h>
#include <sha.h>
#include <hmac.h>

/////////////////
////Kerberos 5 api
#include <krb5.h>
```

```
int srv_RecvN_Rel(char *recvLine,int *msgLen,int *sock,char *ip,int
    port,char *sndrIP){
int len = 0,bytesleft = 0,numsent = 0,curpos = 0;
int numrecvd = 0,numrecvd2 = 0,resendCount = 0,totalrecvd = 0;
char sendline[MSGLEN],tmprecvline[MSGLEN],tmp2recvline[MSGLEN],str[
    INET_ADDRSTRLEN];
fd_set readset;
struct timeval timeout;
struct sockaddr_in ownAddr,senderAddr,tmpAddr;
char *retstr;

memset(tmprecvline,'\0',MSGLEN);
memset(str,'\0',INET_ADDRSTRLEN);

memset(&ownAddr,0,sizeof(ownAddr));
ownAddr.sin_family = AF_INET;
ownAddr.sin_port = htons(port);
ownAddr.sin_addr.s_addr=inet_addr(ip);

memset(&senderAddr,0,sizeof(senderAddr));
senderAddr.sin_family = AF_INET;
senderAddr.sin_port = htons(port);
memset(&tmpAddr,0,sizeof(tmpAddr));

if (bind(*sock,(SA *)&ownAddr,sizeof(ownAddr)) < 0){
     printf("server: err bind for srv_recvn. Abort\n");
     return -1;
     }

curpos = 0;
totalrecvd = 0;

while (1){

    while ((numrecvd = recvfrom(*sock,tmprecvline+curpos,MSGLEN,0,(SA *)
       &senderAddr,
                        (socklen_t*)&len)) < 0){
      if (errno != EINTR){
        printf("p4clientR: srv_RecvN_Rel: error recv GRP_AP_REQ: %s\n",
           strerror(errno));
        return -1;
        }
      }
```

```
if ((retstr = inet_ntop(senderAddr.sin_family,&senderAddr.sin_addr,
   str,
              INET_ADDRSTRLEN))==NULL){
   printf("p4client: srv_RecvN_Rel: err inet_ntop");
   return -1;
   }

memset(tmp2recvline,'\0',MSGLEN);

FD_ZERO(&readset);    FD_SET(*sock, &readset);
timeout.tv_sec = 0;   timeout.tv_usec = 10;

if (select(*(sock)+1, &readset, NULL, NULL, &timeout) > 0){

   //MSG_PEEK for arg 4. Dont consume the data, just check
   while ((numrecvd2 = recvfrom(*sock,tmp2recvline,MSGLEN,MSG_PEEK,
                    (SA *)&tmpAddr, (socklen_t*)&len)) < 0){
     if (errno != EINTR){
       printf("error recv GRP_AP_REQ: %s\n",strerror(errno));
       return -1;
       }
     }

  printf("rcvr peek: %d bytes\n",numrecvd2);
  }
else {
  printf("p4rcvr: recvfrom: no more bytes\n");
  break;
  }

memset(sendline,'\0',MSGLEN);
sprintf(sendline,"ACK %d",numrecvd);

printf("rcvr: sendto ACK\n");
//ACK (expect) the next desired byte
if ((numsent = sendto(*sock, sendline, strlen(sendline),
   0, (struct sockaddr *)&senderAddr,len)) < 0){
     if (errno != EINTR){
     printf("p4server err sendto TGS_REQ_Rel\n");
     return -1;
     }
   }
printf("recvr sent ACK: %s\n",sendline);

totalrecvd = totalrecvd + numrecvd;
```

```
    curpos = curpos + numrecvd;
    }

strncpy(recvLine,tmprecvline,strlen(tmprecvline));
*msgLen = totalrecvd;

strncpy(sndrIP,str,INET_ADDRSTRLEN);

return 0;
}

//Pre: dest must have sufficient memory
char *getHexDigestStr(unsigned char dig[],int len,unsigned char *dest){
int i = 0;
unsigned char c[5];

for (;i<len;i++){
     memset(c,'\0',5);
     sprintf(c,"%x",dig[i]);
     strcat(dest,c);
     }

return dest;
}

//openSSL: decrypt the ciphertext using the encryption key,IV into
    ptext
int p4OpensslDecAES(UCHR *ctext,int *ctextlen,UCHR *encKey,UCHR *encIV,
   UCHR *ptext){
////max ciphertext len for a n bytes of plaintext is n + AES_BLOCK_SIZE
     -1 bytes
int lastlen = 0;
int ptextlen = *ctextlen;
int finalPtextLen = 0;

EVP_CIPHER_CTX decCtxt;

////////////////////////////////////////////////
//Openssl decryption cipher init

EVP_CIPHER_CTX_init(&decCtxt);
EVP_DecryptInit_ex(&decCtxt, EVP_aes_256_cbc(), NULL, encKey, encIV);

EVP_DecryptInit_ex(&decCtxt, NULL, NULL, NULL, NULL);
```

```
EVP_DecryptUpdate(&decCtxt, ptext, &ptextlen, ctext, *ctextlen);
EVP_DecryptFinal_ex(&decCtxt, ptext+ptextlen, &finalPtextLen);

*ctextlen = ptextlen + finalPtextLen;

//Openssl finalize
EVP_CIPHER_CTX_cleanup(&decCtxt);

return 0;
}

//ctext must be large enough to hold enc{ptext}
int p4OpensslEncAES(UCHR *ptext,int ptextlen,UCHR *encKey,UCHR *encIV,
    UCHR *ctext){
//max ciphertext len for a n bytes of plaintext is n + AES_BLOCK_SIZE
    -1 bytes
int ctextlen = ptextlen + AES_BLOCK_SIZE, finallen = 0;
int lastlen = 0;
UCHR *ctextTmp = malloc(ctextlen);

////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
////Openssl: Encrypt ptext using encKey into ctext///////

////////////////////////////////////////////////////////////////
////Openssl cipher init
EVP_CIPHER_CTX enc_contxt;//openssl managerial sec op contexts
EVP_CIPHER_CTX_init(&enc_contxt);

//Init encryption: use symmetric key of group member
EVP_EncryptInit_ex(&enc_contxt, EVP_aes_256_cbc(), NULL,encKey,encIV);
EVP_EncryptUpdate(&enc_contxt, ctextTmp, &ctextlen,ptext, ptextlen);
EVP_EncryptFinal_ex(&enc_contxt, ctextTmp+ctextlen, &finallen);

lastlen = ctextlen + finallen;

memcpy(ctext,ctextTmp,lastlen);

//Openssl finalize
EVP_CIPHER_CTX_cleanup(&enc_contxt);

//free(ctextTmp);

return 0;
}
```

```
//send ////////////////////////////////
int cli_Send_AP_REP(int socket,char *clientPrincStr,UCHR *encKey,char *
    absGrpNameStr){
char sendLine[MSGLEN];
int numsent = 0;
struct timespec ts;

UCHR encPrincStr[P4PRINCIPALNMLEN+AES_BLOCK_SIZE];
UCHR encTimeStr[32+AES_BLOCK_SIZE];
UCHR timeValStr[500];

memset(encPrincStr,'\0',(P4PRINCIPALNMLEN+AES_BLOCK_SIZE));
memset(encTimeStr,'\0',(32+AES_BLOCK_SIZE));
memset(sendLine,'\0',MSGLEN);
memset(timeValStr,'\0',500);

if (clock_gettime( CLOCK_MONOTONIC, &ts))
    printf("cli_send_ap_rep; err clock_gettime\n");

sprintf(timeValStr,"%ld",ts.tv_nsec);

p4OpensslEncAES(timeValStr,strlen(timeValStr),encKey,iv_Dummy,
    encTimeStr);
p4OpensslEncAES(clientPrincStr,strlen(clientPrincStr),encKey,iv_Dummy,
    encPrincStr);

sprintf(sendLine,"AP_REP %s %s %s",absGrpNameStr,encPrincStr,encTimeStr
    );

printf("p4clientR send %s\n",sendLine);

if ((numsent = sendn(socket,sendLine,strlen(sendLine)))<0){
    printf("p4client err sendn: %dB sent\n",numsent);
    return -1;
    }

return 0;
}

//Receive mcast GRP_AP_REQ from client/requestor
int cli_Mcast_Recv_GRP_AP_REQ(char *mcastAddr,char *grpName,UCHR *
    tkt_encLtgkSgk,char *senderIP){
int grpsock = 0,numrecvd = 0,recvdLen = 0;
struct ip_mreq mrq;
```

```
char recvline[MSGLEN];

memset(recvline,'\0',MSGLEN);

////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//Multicast reception


if ((grpsock = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP)) < 0){
   printf("p4 reciever: can't create mcast socket. Aborting\n");
   return 1;
   }

//senderAddr.sin_family = AF_INET;
//senderAddr.sin_port = htons(P4TESTGRPPORT);
//memset(&ownAddr,0,sizeof(ownAddr));
//ownAddr.sin_family = AF_INET;
//ownAddr.sin_port = htons(P4TESTGRPPORT);
//ownAddr.sin_addr.s_addr=htonl(INADDR_ANY);

if (P4MCAST){
   //Join test P4 mcast group
   mrq.imr_multiaddr.s_addr=inet_addr(mcastAddr);
   mrq.imr_interface.s_addr=htonl(INADDR_ANY);
   setsockopt(grpsock,IPPROTO_IP,IP_ADD_MEMBERSHIP,&mrq,sizeof(mrq));
   }

/*
if (bind(grpsock,(SA *)&ownAddr,sizeof(ownAddr)) < 0){
    printf("p4clientR: err bind for mcast recv AP_REQ. Abort\n");
    return -1;
    }
*/


////////////////////////////////////////////////////////
//recieve GP_AP_REQ////////////////////////////////////

srv_RecvN_Rel(recvline,&recvdLen,&grpsock,LOCALIP,P4TESTGRPPORT,
   senderIP);
/*
memset(recvline,'\0',MSGLEN);
while ((numrecvd = recvfrom(grpsock,recvline,MSGLEN,0,(SA *)&senderAddr
   ,(socklen_t*)sizeof(senderAddr))) < 0){
```

```
    if (errno != EINTR){
        printf("error recv GRP_AP_REQ: %s\n",strerror(errno));
        break;//return -1;
        }
    continue;
    }
perror("recvfrom");
*/

printf("P4 client/reciever recv %s\n",recvline);
sscanf(recvline,"GRP_AP_REQ %s %s",grpName,tkt_encLtgkSgk);
//inet_ntop(senderAddr.sin_family,&senderAddr.sin_addr,senderIP,
    INET_ADDRSTRLEN);
printf("clientR recieved mcast from %s\n",senderIP);

////////////////////////////////////////

close(grpsock);

return 0;
}

int cliR_Recv_AP_REQ(int socket,char *grpNmPrincList,char *grpNm,char *
    cliPrincStr,UCHR *encLTGK){
UCHR recvline[MSGLEN];
int numrecvd = 0;

memset(recvline,'\0',MSGLEN);

while ((numrecvd = recv(socket,recvline,MSGLEN,0))<0){
    if (errno != EINTR){
        printf("p4receiver : error recv AP_REQ: %s\n",strerror(errno));
        //break;//return -1;
        }
    }

printf("p4reciever recvd %dB: %s\n",numrecvd,recvline);

sscanf(recvline,"AP_REQ %s %*s",grpNmPrincList);
sscanf(recvline,"AP_REQ %*s %s",encLTGK);

if (grpNmPrincList!=NULL){
    sscanf(grpNmPrincList,"%s %*s",grpNm);
    sscanf(grpNmPrincList,"%*s %s",cliPrincStr);
    }
```

```
else {
   printf("p4rcvr err grpnameplist parse fail\n");
   //close(tgssock);
   return -1;
   }

return 0;
}

int cliR_Send_AP_REP_Reqr(char *requestorIP,char *ownPrinc,UCHR *
   sessGrpKey,char *grpName){
int cliGRPREPsock,ret;
struct sockaddr_in cliRequestor;

cliGRPREPsock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

cliRequestor.sin_family = AF_INET;
cliRequestor.sin_port = htons(CLIREQACKPORT);

if((requestorIP==NULL) || (inet_pton(AF_INET,requestorIP,&cliRequestor.
   sin_addr) <= 0)){
      printf("p4clientR: inet_pton error senderIP for AP_REP\n");
      return 1;
      }

////////////////////////////////////////////////////
//connect to client/requestor
while ((ret = connect(cliGRPREPsock,(SA *)&cliRequestor,sizeof(
   cliRequestor))) < 0){
     if (errno == ETIMEDOUT){
         printf("p4clientR: cli/Req connect timed out\n");
         return 1;
         }
     else if (errno == ECONNREFUSED){
         printf("p4clientR: cli/Req connection refused!\n");
         return 1;
         }
     else
        return 1;
        }

printf("clientR: connected to client/requestor at %s\n",requestorIP);

cli_Send_AP_REP(cliGRPREPsock,ownPrinc,sessGrpKey,grpName);
```

```
return 0;
}

//setup socket for TGS connection AP exchange, return newly
//connected tgs socket
int cliR_AwaitTGS(int acceptSock,struct sockaddr_in *tgsAddr){
struct sockaddr_in ownaddr;
int newConnFd,len;

////////////////////////////////////////
//socket address structures/////////////
ownaddr.sin_family = AF_INET;
ownaddr.sin_port = htons(GKCLIENTPORT);
ownaddr.sin_addr.s_addr=htonl(INADDR_ANY);
////////////////////////////////////////


//////////////////////////////////////////////////////
if (bind(acceptSock,(SA *)&ownaddr,sizeof(ownaddr)) < 0){
   printf("p4rcvr: err bind. Abort\n");
   return 1;
   }

if (listen(acceptSock,LISTEN_Q) < 0){
   printf("p4rcvr: err listen. Abort\n");
   return 1;
   }
//////////////////////////////////////////////////////


//////////////////////////////////////////////////////
//Await TGS connection (simple test model: AUTH, GKU, recv mcast
   scenario)
memset(tgsAddr,0,sizeof(tgsAddr));
len = sizeof(tgsAddr);

if ((newConnFd = accept(acceptSock,(SA *)tgsAddr,&len)) < 0){
   printf("p4rcvr accept error\n");
   return -1;
   }

//return the new tgs socket
return newConnFd;
}
```

```
int main(int argc,char *argv[]){

struct hostent *kdchost,*ownhost;
struct timeval timeout;
char **adrlist_pptr;
char ownIP[16],str[INET_ADDRSTRLEN],protomsg[MSGLEN],recvline[MSGLEN];
char kdcIP[INET_ADDRSTRLEN];
unsigned char us_protomsg[MSGLEN];
char *clihostname;
unsigned char hashresult[SHA1OUTSIZE];
unsigned char *hashres_p,*digeststr;
unsigned char hashout[SHA1OUTSIZE];
UCHR grpName[P4GRPNAMELEN],princList[P4PRINCIPALNMLEN],encLTGK[
    P4KEYSIZE+AES_BLOCK_SIZE];
UCHR grpNamePrincList[P4GRPNAMELEN+P4PRINCIPALNMLEN];
char *clientPrincStr;
UCHR cliTGSSharedKey[P4KEYSIZE];
struct sockaddr_in ownaddr,tgsaddr;
struct sockaddr_in kdc,cliRequestor;
struct ip_mreq mrq; //IP multicast request addr structure
int tgssock,kdcsock,grpsock,ret,numsent,numrecvd,i,connfd;
int cliGRPREPsock;
unsigned int len;

krb5_ap_rep_enc_part *rep_ret;
krb5_auth_context auth_context = 0;
krb5_ccache ccdef;
krb5_context context;
krb5_data cksum_data;
krb5_data recv_data;
krb5_error *err_ret;
krb5_error_code retval;
krb5_principal p4gkclient, p4gkserver;
krb5_keyblock * sKey; /* Used as the long term shared key */

memset(grpName,'\0',P4GRPNAMELEN);memset(princList,'\0',
    P4PRINCIPALNMLEN);
memset(encLTGK,'\0',P4KEYSIZE+AES_BLOCK_SIZE);

//ownhost = (struct hostent *)malloc(sizeof(struct hostent));


////////////////////////////////////////////////////////
//KDC AUTH. Perhaps not useful on rcvr
```

```
kdcsock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

if (LOCALTESTING){//TESTING
   strncpy(kdcIP,KDCIP,INET_ADDRSTRLEN);
   }
else {
     if ((kdchost = gethostbyname(KRB_GKS_HOST)) == NULL){
        printf("error gethostbyname: %s %s\n",KDCNAME,strerror(h_errno)
           );
        return 1;
        }

     if (kdchost->h_addrtype == AF_INET){
         adrlist_pptr = kdchost->h_addr_list;
         memset(kdcIP,'\0',INET_ADDRSTRLEN);
         strcpy(kdcIP,inet_ntop(kdchost->h_addrtype, *adrlist_pptr, str
            , sizeof(str)));
         printf("client: kdc IP: %s\n",kdcIP);
         }
     else {
         printf("error gethostbyname: unexpected address type!. Abort")
            ;
         return -1;
         }
      }

//writes kdcip into socket addr struct in nwbyteorder
    if(inet_pton(AF_INET,kdcIP,&kdc.sin_addr) <= 0){
      printf("inet_pton error\n");
      return 1;
      }

kdc.sin_family = AF_INET;
kdc.sin_port = htons(KDCAUTHPORT);

////////////////////////////////////////////////////
//connect to kdc: AUTH//////////////////////////////
 while ((ret = connect(kdcsock,(SA *)&kdc,sizeof(kdc))) < 0){
     if (errno == ETIMEDOUT){
         printf("p4client: kdc connect timed out\n");
         return 1;
         }
     else if (errno == ECONNREFUSED){
         printf("p4client: kdc connection refused!\n");
```

```
        return 1;
        }
    else
       return 1;
       }

printf("client: connected to KDC\n");
////////////////////////////////////////////

if (P4KRB) {

        retval = krb5_init_context(&context);
        if (retval) {
                perror("ERROR when initializing krb5\n");
                return 1;
        }

        retval = krb5_cc_default(context, &ccdef);
        if (retval) {
                perror("ERROR getting default cred cache\n");
                return 1;
        }

        retval = krb5_cc_get_principal(context, ccdef, &p4gkclient);
        if (retval) {
                perror("ERROR getting cred cache defaul princ\n");
                return 1;
        }

        retval = krb5_sname_to_principal(context, KRB_GKS_HOST,
           KRB_SERVICE,
                                        KRB5_NT_SRV_HST, &p4gkserver);
        if(retval)
        {
                perror("ERROR while generating server princ name");
                return 1;
        }

        retval = krb5_sendauth(context, &auth_context, (krb5_pointer) &
           kdcsock,
                        KRB_GKS_VERSION, p4gkclient, p4gkserver,
                        AP_OPTS_MUTUAL_REQUIRED,
                        &cksum_data,
                        0,          // no creds, use ccache instead /
                        ccdef, &err_ret, &rep_ret, NULL);
```

```
        if (retval)
        {
                printf("p4client: err krb5_sendauth: %s \n",
                   krb5_get_error_message(context,retval));
                return 1;
        }

retval = krb5_auth_con_getkey(context, auth_context, &sKey);

        //TODO : COPY KEY OFF HERE -Sean

        //memcpy(cliTGSSharedKey,0,KRBKEYSIZE);
        memcpy(cliTGSSharedKey,sKey->contents,sKey->length);

        retval = krb5_unparse_name(context,p4gkclient,&clientPrincStr);

        printf("%s key[",clientPrincStr);
        for(i = 0; i < sKey->length; ++i)
        {

                printf("\\x%X",(UCHR)cliTGSSharedKey[i]);
        }
        printf("]\n");


        krb5_free_keyblock(context, sKey);
printf("##\n");
        krb5_free_principal(context, p4gkserver);        // finished
           using it /
        krb5_free_principal(context, p4gkclient);
        krb5_cc_close(context, ccdef);
        krb5_free_context(context);
        }

        close(kdcsock);


////////////////////////////////////////////////
//TSS API: Seal principal-member/kdc(tgs) key

////////////////////////////////////////////


////////////////////////////////////////////////////
////////////////////////////////////////////////////
```

```
//Await TGS connection (simple test model: AUTH, GKU, recv mcast
    scenario)

if ((tgssock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)) < 0){
    printf("p4 reciever: can't create socket. Aborting\n");
    return 1;
    }

memset(&tgsaddr,0,sizeof(tgsaddr));
len = sizeof(tgsaddr);

if ((connfd = cliR_AwaitTGS(tgssock,&tgsaddr)) < 0){
    printf("p4clientR: err waiting for TGS conn\n");
    return -1;
    }
//////////////////////////////////////////////////////


//////////////////////////////////////////////////////
//////////////////////////////////////////////////////
//Await AP_REQ from GKS
//"AP_REQ" <grpname grpprincipallist> K_TGS/si[LTGK]

if ((ret = cliR_Recv_AP_REQ(connfd,grpNamePrincList,grpName,
    clientPrincStr,encLTGK)) < 0){
    printf("err p4client: failed recv AP_REQ from TGS\n");
    return -1;
    }

UCHR decLTGK[P4KEYSIZE];
memset(decLTGK,'\0',P4KEYSIZE);

//TODO need real cli/TGS shared key from KRB for arg 3
p4OpensslDecAES(encLTGK,&len,k_Dummy,iv_Dummy,decLTGK);

//////////////////////////////////////////////////////
//send AP_REP to TGS as ACK/CONF
cli_Send_AP_REP(connfd,"rcvr"/*TODO p4gkclient*/,k_Dummy/*TODO
    cliTGSSharedKey */,grpNamePrincList);
//////////////////////////////////////////////////////
close(connfd);

//TGS connection unneeded for this group setup scenario
close(tgssock);
//////////////////////////////////////
```

```
//////////////////////////////////////
//////////////////////////////////////
//Mcast reception of SGK, and AP_REP////

UCHR senderIP[INET_ADDRSTRLEN],cliGRPGrpName[P4GRPNAMELEN],
  tkt_encLtgkSgk[P4KEYSIZE+AES_BLOCK_SIZE];
UCHR decSgk[P4KEYSIZE];
memset(senderIP,'\0',INET_ADDRSTRLEN);
memset(cliGRPGrpName,'\0',P4GRPNAMELEN);
memset(tkt_encLtgkSgk,'\0',P4KEYSIZE+AES_BLOCK_SIZE);
memset(decSgk,'\0',P4KEYSIZE);

cli_Mcast_Recv_GRP_AP_REQ(P4TESTGRPIP,cliGRPGrpName,tkt_encLtgkSgk,
    senderIP);

//TODO need shared key from KRB
p4OpensslDecAES(tkt_encLtgkSgk,&len/*P4KEYSIZE+AES_BLOCK_SIZE*/,decLTGK
    ,iv_Dummy,decSgk);

//TODO dont know own principal name from KRB, using rcvr
cliR_Send_AP_REP_Reqr(senderIP,"rcvr",decSgk,cliGRPGrpName);


return 0;
}
```

## C.4 Definitions Header

Listing C.4: Definitions Header

```
/*Jon Jenkins P4 2012
 * global variables, defines
 */

#ifndef _P4DEFINES_H_
#define _P4DEFINES_H_

//Use AES max keysize. TPMRSA=2048bits
#define P4KEYSIZE 32
#define P4KEYGENROUNDS 10
#define P4HASHSIZE 20 //SHA1 used by TPM
#define P4PRINCIPALNMLEN 200
```

```
#define P4KEYBLOBNAMELEN 2*P4KEYSIZE
#define P4GRPNAMELEN 500

//TPM defs
#define TPMBLOBSIZE P4KEYSIZE*2

//networking
#define GKCLIENTPORT 8880
#define LISTEN_Q 1024
#define LOCALIP "127.0.0.1"

#define MAXMCSTGRPSIZE 20
#define MAXMCASTGROUPS 100
#define MSGLEN 500
#define MAXLINE 4096
#define HOSTNMLEN 100
#define TESTGRPID 0
#define SHA1OUTSIZE 20
#define MAXCLIENTS 50
#define MAXGRPNAMELEN 500

//enable/disable kerberized message exchanges!!!
#define P4KRB 0

#define KRBKEYSIZE 128 //???
#define KRBKEYSIZE2 24 //bytes

#define LOCALTESTING 1

#define P4MCAST 0

#define MYBYTE 0x8
typedef unsigned char UCHR;

#define DUMMYKEY0  \
        {'0','1','2','3','4','5','6','7','8','9','2','3','4','5','6','7
            ', \
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            , \
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            , \
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            , \
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            , \
```

```
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            , \
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            , \
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            , \
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            ,\
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            ,\
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            ,\
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            , \
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            , \
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            , \
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'
            , \
        '0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','1'
           }

#define DUMMYKEY1  \
        {'0','1','2','3','4','5','6','7','8','9','2','3','4','5','6','7
            ', \
                '0','1','2','3','4','5','6','7','8','9','2','3','4','
                   5','6','7', \
                '0','1','2','3','4','5','6','7','8','9','2','3','4','
                   5','6','7', \
                '0','1','2','3','4','5','6','7','8','9','2','3','4','
                   5','6','7', \
        '0','1','2','3','4','5','6','7','8','9','2','3','4','5','6','7'
            , \
                '0','1','2','3','4','5','6','7','8','9','2','3','4','
                   5','6','7', \
                '0','1','2','3','4','5','6','7','8','9','2','3','4','
                   5','6','7', \
                '0','1','2','3','4','5','6','7','8','9','2','3','4','
                   5','6','7', \
                '0','1','2','3','4','5','6','7','8','9','2','3','4','
                   5','6','7', \
                '0','1','2','3','4','5','6','7','8','9','2','3','4','
                   5','6','7', \
```

```
        '0','1','2','3','4','5','6','7','8','9','2','3','4','5','6','7'
            , \
              '0','1','2','3','4','5','6','7','8','9','2','3','4','5','6
                  ','7', \
                    '0','1','2','3','4','5','6','7','8','9','2','3','4','
                       5','6','7', \
                    '0','1','2','3','4','5','6','7','8','9','2','3','4','
                       5','6','7', \
                    '0','1','2','3','4','5','6','7','8','9','2','3','4','
                       5','6','7', \
                    '0','1','2','3','4','5','6','7','8','9','2','3','4','
                       5','6','7'}

//32B 256bit max AES keysize
#define DUMMYKEYAES  \
        {'0','1','2','3','4','5','6','7','8','9','2','3','4','5','6','7
            ', \
                    '0','1','2','3','4','5','6','7','8','9','2','3','4','
                       5','6','7'}

#define DUMMYIV  \
        {'0','1','2','3','4','5','6','7','8','9','2','3','4','5','6','7
            ', \
        '0','1','2','3','4','5','6','7','8','9','2','3','4','5','6','7'
           }

#define DUMMYSESSKEY  \
        {'0','1','2','3','4','5','6','7','8','9','2','3','4','5','6','7
            ', \
        '0','1','2','3','4','5','6','7','8','9','2','3','4','5','6','7'
           }



struct client {
   int id;
   char principalString[P4PRINCIPALNMLEN];
   char ipaddr[INET_ADDRSTRLEN];
   char name[HOSTNMLEN];
   int port;
   int sockfd;
   int dead;
   int handlerpid;
   unsigned char shkey[P4KEYSIZE];
   int groupList[MAXMCASTGROUPS];
```

```
    int groupCount;
};

struct child {
    int pid;
    int status;
    int clientfd;

};

struct mcastGrpClient{
    int id;
    char ipaddr[16];
    char princString[P4PRINCIPALNMLEN];

};

struct mcastgrp {
    int size;
    int grpid;
    //int clientIDList[MAXCLIENTS];

    struct mcastGrpClient clients[MAXCLIENTS];

    char grpName[MAXGRPNAMELEN];
    char clientPrincipalList[MAXCLIENTS][P4PRINCIPALNMLEN];
    char sharedKeyBlobName[P4KEYBLOBNAMELEN];
};

//dummy producer/kdc shared key
unsigned char k_Dummy[P4KEYSIZE] = DUMMYKEYAES;
unsigned char iv_Dummy[P4KEYSIZE] = DUMMYIV;
unsigned char sesskey_Dummy[P4KEYSIZE] = DUMMYSESSKEY;

//Multicast group storage (GKS)
struct mcastgrp MCGrpList[MAXMCASTGROUPS];

struct frame{
  int seq,type,size;
  char data[MSGLEN];
};

#endif
```

## C.5 Protocol Description

Listing C.5: Protocol Description

```
P4 protocol sketch

Jon Jenkins

p4gkc: 'centralized' branch version with group key distribution done by
    kdc/tgc.


Preconditions:

-All nodes trusted operation by virtue of TPM
-Shared symmetric key between each principal and KDC (K_j for each (j,
   KDC))
-Group members known a-priori, static


1. p4cli: D_i   K_Di/tgc{REQ(mcast,G_i)}  -> kdc=(auth,tgs)
   P4_REQ *     ~AS_REQ
   p4srv: K_Di/tgc{REQ(mcast,G_i)}

   p4srv: KerberosAPI: gen. K_i. K_i,1 = enc(K_1/tgc,K_i)
                                                    K_i,2    = enc(
                                                       K_2/tgc,K_i)



   p4srv: TKT = K_Di/tgc{K_i,1,K_i,2,...}
      P4_REP*
XXX 2. p4cli: D_i <---      K_Di/tgc{TKT,s1}          p4srv
                          ~AS_REPLY

2. p4srv: IPMcast:   TKT,s1                  -> D_1
   P4_MDIST     ~AP_REQ
                                  TKT,s1     -> D_2
                                  TKT,s1     -> D_3
                                               ...


*message format is not identical to Kerberos, so either modify Kerberos
or call it with interfaced message information
```

## C.6 Server Code

Listing C.6: Server Code

```
//#define _POSIX_C_SOURCE 200112L
#define _POSIX_C_SOURCE 200809L

/*Jon Jenkins  p4 2012
  p4server.c KDC
  KDC(tgs,auth server)
 */

#include <arpa/inet.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <netdb.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/un.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>

#ifdef HAVE_SYS_SOCKIO_H
#include <sys/sockio.h>
#endif


#define GKSPORT 8888
#define AUTHPORT 8889
#define KRB_SERVICE "noleptr"
#define KRB_GKS_VERSION "KRB5_GKS_protocol_v0.1"

#define SA struct sockaddr

#ifndef INET_ADDRSTRLEN
```

```
#define INET_ADDRSTRLEN 16
#endif

#define P4TESTGRPIP "239.0.0.1"
#define P4TESTGRPPORT 5505


#include "p4defines.h"
#include "p4server.h"

//TPM
#include <tspi.h>
#include <trousers.h>
#include <tss.h>

//OpenSSL
#include <crypto.h>
#include <aes.h>
#include <openssl/evp.h>

////////////////
////Kerberos 5 api
#include <krb5.h>

#include "util.c"

int form_P4Rep(char *msg,int grpid,const char hostnm[],char *kstr,UCHR
    *hash){

        memset(msg,'\0',MSGLEN);
        sprintf(msg,"P4_REP %d %s %s %s",grpid,hostnm,kstr,hash);

        return 0;
}

int form_P4Mdist(char *msg,int grpid,const char hostnm[],char *kstr,
    UCHR *hash){

        memset(msg,'\0',MSGLEN);
        sprintf(msg,"P4_MDIST %d %s %s",grpid,kstr,hash);

        return 0;
}

//Create signing key using TPM processes
```

```
//Will be slow, recommend not creating key with TPM
//Kerberos/openssl instead
//int createSigningKey(){ //TODO

//}


//Pre: (loose) data holds data to write
//blob file convention: "p4tpmblob.tgc.sign" p4tpm.<entity>.<purpose>
//in: all
int writeEncBlob(char *blobfilename,int datalen,BYTE *data){

        FILE *fptr;

        if ((fptr=fopen(blobfilename,"wb"))!=NULL){
                write(fileno(fptr),data,datalen);

                fclose(fptr);
        }

        return 0;
}

//Pre: blob allocated!
//blob file naming convention: "p4tpmblob.tgc.sign" p4tpm.<entity>.<
   purpose>
//in: blobfilename,bloblen out: blob
int loadEncBlob(char *blobfilename,int bloblen,BYTE *blob){

        FILE *fptr;

        if ((fptr=fopen(blobfilename,"rb"))!=NULL){
                read(fileno(fptr),blob,bloblen);

                fclose(fptr);
        }

        return 0;
}

//Pre: hash allocated
//tpm HMAC-SHA hash of hashdata, keyed with hmackey of length keylen,
   placed into hash
int p4TpmHMAC(BYTE *hmackey, int keylen, BYTE *hashdata,int hashdatalen
   ,BYTE *hash){
```

```
    int hashlen,res;
    BYTE *tmphash;

    ////////////////////////////////////////
    //TPM session is managed internally for Trspi
    res = Trspi_HMAC(TSS_HASH_SHA1, keylen, hmackey, hashdatalen,
        hashdata, hash);

    return 0;
}

//Pre: hash allocated
//tpm hash of data UNUSABLE: keyless hash
int p4TpmHash(BYTE *hashdata,int hashdatalen,BYTE *hash){

        TSS_RESULT tsr;
        TSS_HCONTEXT hContext;
        TSS_HTPM hTPM;
        TSS_HHASH hFinalHash;

        int hashlen;
        BYTE *tmphash;

        ///Init TPM session//////////////
        //Create context, connect to TPM
        Tspi_Context_Create(&hContext);
        Tspi_Context_Connect(hContext,NULL);//NULL: local, or
            TSS_UNICODE str for remote

        ////Get the TPM handle//////////
        tsr=Tspi_Context_GetTpmObject(hContext, &hTPM);
        printf("getting tpm object: %s\n",Trspi_Error_String(tsr) );
        ////////////////////////////////


        //Create generic hash obj (tss: SHA1 only)
        tsr = Tspi_Context_CreateObject(hContext,
                        TSS_OBJECT_TYPE_HASH,
                        TSS_HASH_SHA1,
                        &hFinalHash);
        printf("create hash obj: %s\n",Trspi_Error_String(tsr) );

        //hash data into object
        tsr = Tspi_Hash_UpdateHashValue(hFinalHash,
```

```
                        hashdatalen,
                        hashdata);
        printf("hash data: %s\n",Trspi_Error_String(tsr) );

        //extract hash to bytearray (out var)
        tsr = Tspi_Hash_GetHashValue(hFinalHash,
                        &hashlen,
                        &tmphash);
        printf("extract hash data: %s\n",Trspi_Error_String(tsr) );

        memcpy(hash,tmphash,P4HASHSIZE);//copy to out var..direct write
            fails

        ////////////////////////////////////////////
        //Session Cleanup

        Tspi_Context_CloseObject(hContext, hFinalHash);

        //Finalize
        Tspi_Context_Close(hContext);
        printf("Context Closed. TPM session over\n");
        ////////////////////////////////////////////


        return 0;
}

//Seal an encrypted blob to
//TPM storage. Requires sealkey to decrypt. Optionally PCR statematch
//our use case: sealed data = key
//in: all
int p4TpmSeal(char blobname[], BYTE datatoseal[],int datalen){

TSS_HCONTEXT hContext;
TSS_HTPM hTPM;
TSS_FLAG flg;
TSS_BOOL fTpmState;
TSS_RESULT tsr;
TSS_HPOLICY srkpol,hSealkPol,tpmpol,newtpmpol;

TSS_HKEY hSRK, hSealKey; //storage root key, sealing key handle
TSS_HENCDATA hSealedData;
TSS_HPCRS hPCRs,nullhPCRs;
TSS_UUID srku = TSS_UUID_SRK;
BYTE mylamesecret[] = {0,0,0,0,0,
```

```
        0,0,0,0,0,
        0,0,0,0,0,
        0,0,0,0,0};

UINT32 unsealedDataLen,pcrLen;
BYTE *unsealedData,*pcrVal;
TSS_HPOLICY hPolicy, srkUsagePolicy;


////////////////////////////////
//Create context, connect to TPM
Tspi_Context_Create(&hContext);
Tspi_Context_Connect(hContext,NULL);//NULL: local, or TSS_UNICODE str
   for remote

////Get the TPM handle//////////
tsr=Tspi_Context_GetTpmObject(hContext, &hTPM);
printf("getting tpm object: %s\n",Trspi_Error_String(tsr) );


//////////////////////////////////////////////////////////
////Task: seal to PCRs: hash TPM state hashes, write to PCRs
//////////////////////////////////////////////////////////

//////////////////////////////////////////////////////
////Create objects.. (all objects bound to the context)

//data object for sealed data
tsr = Tspi_Context_CreateObject(hContext,TSS_OBJECT_TYPE_ENCDATA,
   TSS_ENCDATA_SEAL,&hSealedData);
printf("create data obj test: %s\n",Trspi_Error_String(tsr) );

//..and the pcr composite object
//printf("create pcr comp obj test: %s\n",
//    Trspi_Error_String(tsr = Tspi_Context_CreateObject(hContext,
//    TSS_OBJECT_TYPE_PCRS,
//    TSS_PCRS_STRUCT_DEFAULT,
//    &hPCRs)));


////Load SRK from TPM (create object insufficient)//
tsr = Tspi_Context_LoadKeyByUUID(hContext, TSS_PS_TYPE_SYSTEM, srku, &
   hSRK);
printf("create srk obj test: %s\n",Trspi_Error_String(tsr) );
```

```
tsr = Tspi_GetPolicyObject(hSRK, TSS_POLICY_USAGE, &srkpol);
printf("get srkpolobj: %s\n",Trspi_Error_String(tsr) );
tsr = Tspi_Policy_SetSecret(srkpol, TSS_SECRET_MODE_SHA1, 20,
                mylamesecret);
printf("setsrksec: %s\n",Trspi_Error_String(tsr) );
////////////////////////////////////////////////

////////////////////////////////////////////
////PCR values into composite, seal
//printf("pcrread1 test: %s\n",Trspi_Error_String(
//    tsr=Tspi_TPM_PcrRead(hTPM, 5, &pcrLen, &pcrVal) )); //pull..
//printf("pcrset1 test: %s\n",Trspi_Error_String(
//    tsr=Tspi_PcrComposite_SetPcrValue(hPCRs, 5, pcrLen, pcrVal )));//
   and store pcr x


//seal (encrypt) key using the sealing key
//null pcrobj means no required match upon unseal
nullhPCRs = (void *)NULL;

tsr = Tspi_Data_Seal(hSealedData,hSRK/*hSealKey*/,datalen,datatoseal,
   nullhPCRs);
printf("seal data test: %s %s %u\n",Trspi_Error_String(tsr),
            Trspi_Error_Layer(tsr), Trspi_Error_Code(tsr) );


////////////////////////////////////////////////
//Extract data blob, write to file for later "access"
BYTE *encData; //assume API allocates
int encDLen = P4KEYSIZE;
//memset(encData,'\0',P4KEYSIZE);

tsr = Tspi_GetAttribData( hSealedData,
    TSS_TSPATTRIB_ENCDATA_BLOB,
    TSS_TSPATTRIB_ENCDATABLOB_BLOB,
    &encDLen,
    &encData);

writeEncBlob(blobname,encDLen,encData);
////////////////////////////////////////////////


//Tspi_Key_UnloadKey(hSealKey);
//Tspi_Context_CloseObject(hContext, hPCRs);
//Tspi_Context_CloseObject(hContext, hSealKey);
```

```
Tspi_Context_CloseObject(hContext, hSealedData);

////////////////////////////////////////
//Finalize
Tspi_Context_Close(hContext);
printf("Context Closed. TPM session over\n");

return 0;
}

//Pre: outdata allocated and nulled!
//unseal an encrypted blob (here, a key) which was sealed
//to the TPM storage
//in: blobfile, outdatalen out: outdata
int p4TpmBlobUnseal(char *blobfile,int *outdatalen,BYTE *outdata){

TSS_HCONTEXT hContext;
TSS_HTPM hTPM;
TSS_FLAG flg;
TSS_BOOL fTpmState;
TSS_RESULT tsr;
TSS_HKEY hSRK; //storage root key, sealing key handle
TSS_HPCRS hPCRs, nullhPCRs;
TSS_UUID srku = TSS_UUID_SRK;

UINT32 unsealedDataLen,pcrLen;
BYTE *unsealedData,*pcrVal;

////////////////////////////////////
//Retrieve enc blob for sealed data
int bloblen = P4KEYSIZE;
BYTE *blob = (BYTE *)malloc(sizeof(BYTE)*P4KEYSIZE);

loadEncBlob(blobfile,bloblen,blob);
////////////////////////////////////


////////////////////////////////
//Create context, connect to TPM
Tspi_Context_Create(&hContext);
Tspi_Context_Connect(hContext,NULL);//NULL: local, or TSS_UNICODE str
   for remote

////Get the TPM handle//////////
tsr=Tspi_Context_GetTpmObject(hContext, &hTPM);
```

```
printf("getting tpm object: %s\n",Trspi_Error_String(tsr) );
////////////////////////////////


////////////////////////////////
//Recover blob data into object
TSS_HENCDATA hRecoveredData;

tsr = Tspi_Context_CreateObject(hContext,
   TSS_OBJECT_TYPE_ENCDATA,
   TSS_ENCDATA_SEAL,
   &hRecoveredData);
printf("create recovereddata obj: %s\n",Trspi_Error_String(tsr) );

tsr = Tspi_SetAttribData(hRecoveredData,
   TSS_TSPATTRIB_ENCDATA_BLOB,
   TSS_TSPATTRIB_ENCDATA_BLOB,
   bloblen,blob);
printf("set attrib blob into ENCDATA: %s\n",Trspi_Error_String(tsr) );
////////////////////////////////


////////////////////////////////////////////////
//Load SRK from TPM (create object insufficient)//
//Currently using SRK only as sealing key
tsr = Tspi_Context_LoadKeyByUUID(hContext, TSS_PS_TYPE_SYSTEM, srku, &
   hSRK);
printf("create srk obj test: %s\n",Trspi_Error_String(tsr) );

//unneeded?//////////////////////////
//tsr = Tspi_GetPolicyObject(hSRK, TSS_POLICY_USAGE, &srkpol);
//printf("get srkpolobj: %s\n",Trspi_Error_String(tsr) );
//tsr = Tspi_Policy_SetSecret(srkpol, TSS_SECRET_MODE_SHA1, 20,
//                 mylamesecret);
//printf("setsrksec: %s\n",Trspi_Error_String(tsr) );
////////////////////////////////////////////////


////////////////////////////////////////////////
//N.B. Read PCR not necessary on unseal (tpm internal guarantee of
//data/state link)
////////////////////////////////////////////////
```

```
//decrypts data only if PCR's match seal config (unless PCRs were not
   linked at seal)
tsr = Tspi_Data_Unseal(hRecoveredData, hSRK/*hSealKey*/, outdatalen, &
   outdata);
printf("unseal data test: %s %s %u\n",Trspi_Error_String(tsr),
   Trspi_Error_Layer(tsr),Trspi_Error_Code(tsr) );

////////////////////////////////////////
//Finalize

//Tspi_Key_UnloadKey(hSealKey);
//Tspi_Context_CloseObject(hContext, hPCRs);
//Tspi_Context_CloseObject(hContext, hSealKey);

Tspi_Context_Close(hContext);
printf("Context Closed. TPM session over\n");
////////////////////////////////////////

return 0;
}


//get the client ID -which is a direct index to the GKS client records-
//for the given principalName
int getClientID(char *principalName,struct client *clist,int
   curClientCount){
int i = 0;

for (i = 0;i < curClientCount;i++){

   //printf("DB getclientid client %s at %d\n",clist[i].principalString
      ,i);

   if (!strcmp(clist[i].principalString,principalName)){
      //printf("DB getclientid found %s at %d\n",clist[i].
         principalString,i);
      return i;
      }
   }

printf("err: getclientid: client %s not found\n",principalName);
return -1;
}
```

```
int getClientIDList(char *principalNameList[],struct client clist[],int
    curClientCount,int *clientIDList,int requestCount){
int i;

for (i = 0;i < requestCount;i++){
   *(clientIDList+i) = getClientID(principalNameList[i],clist,
      curClientCount);
   }

return 0;
}

//add client info to the GKS *client* records
//in: all
int addClient(char *principalname,UCHR *finalSharedKey,struct client *
   clist,int *clientCount,char *ipAddr){
int i=0;

if (*clientCount < (MAXCLIENTS-1)){
   i = *clientCount;

   memset(clist[i].principalString,'\0',P4PRINCIPALNMLEN);
   strncpy(clist[i].principalString,principalname,strlen(principalname)
      );
   memset(clist[i].shkey,'\0',P4KEYSIZE);
   memcpy(clist[i].shkey,finalSharedKey,P4KEYSIZE);

   //printf("add client: %s added\n",clist[i].principalString);

   //store ipaddress...needed for later GK ops
   memset(clist[i].ipaddr,'\0',INET_ADDRSTRLEN);
   strncpy(clist[i].ipaddr,ipAddr,INET_ADDRSTRLEN);

   clist[i].groupCount = 0;

   int j=0;

   for (j=0;j<MAXMCASTGROUPS;j++)
      {clist[i].groupList[j] = -1;}

   clist[i].id = *clientCount;

   (*clientCount)++;
}
```

```
return 0;
}

//If principal princNameStr exists in group grpNameStr, return
//the index, else -1
int mcastGroupMemberExists(int groupIndex,char *princNameStr,char *
    grpNameStr){
int index,grpSize,i;

if ((index = getGroupIndex(-1,grpNameStr))<0){
    printf("err mcastgroupmemberexists: group %s not found\n",grpNameStr
        );
    return -1;
    }

grpSize = MCGrpList[index].size;

for (i=0;i<grpSize;i++){
    if (!(MCGrpList[index].clients[i].princString,princNameStr))
        {return i;}
    }

return -1;
}

//Give membership in group newGroupId to client principalname
//Pre: principalnames must be unique
//in/out: clist
int addClientGroupMembership(char principalname[],int newGroupId,struct
     client *clist,int clientCount){
int i = 0,j = 0;

for (;i<clientCount;i++){
    if (!strcmp(principalname,clist[i].principalString)){

        j = clist[i].groupCount;

        if (j < (MAXMCASTGROUPS-1))
            {clist[i].groupList[j] = newGroupId;}

        clist[i].groupCount++;
        }
}

return 0;
```

```
}

//add new client to existing GKS mcast group record
//in: all
//return: index in mcast grp structure inserted at
int addMcastGrpMember(int groupIndex,char *princName,int newClientID,
    char *newIpAddr){
int index = 0;

if (groupIndex<0){
    printf("err addmcastgrpmember: group index %d invalid. Abort\n",
        groupIndex);
    return -1;
    }

if (MCGrpList[groupIndex].size < MAXCLIENTS){

    index = MCGrpList[groupIndex].size;

    MCGrpList[groupIndex].clients[index].id = newClientID;
    strncpy(MCGrpList[groupIndex].clients[index].ipaddr,newIpAddr,
        INET_ADDRSTRLEN);
    strncpy(MCGrpList[groupIndex].clients[index].princString,princName,
        P4PRINCIPALNMLEN);

    MCGrpList[groupIndex].size = MCGrpList[groupIndex].size+1;

    return MCGrpList[groupIndex].size;
    }

printf("err addmcastgrpmember: group %d storage full\n",groupIndex);
return -1;
}

int addMcastGrpMembersByPrinc(int groupIndex,int principalCount,char
    principalListStr[],struct client clist[],int cliCount){
int i = 0,clientID = 0;
char *princStr;

if (principalCount<1)
        return -1;

////////////////////////////////////////////////////////////////
//tokenize,add members from principal string//////////////////////
if ((princStr = strtok(principalListStr," ")) == NULL){
```

```
    return -1;
    }

//get client's index into client records
if ((clientID = getClientID(princStr,clist,cliCount)) < 0){
    printf("err addmcastgrpmembersbyprinc: client %s not found\n",
        princStr);
    return -1;
    }

//then add the client to the mcast group
addMcastGrpMember(groupIndex,princStr,clientID,clist[clientID].ipaddr);

i = 1;

while (i<principalCount){
    if ((princStr = strtok(NULL," ")) == NULL)
        break;//bad request

    if ((clientID = getClientID(princStr,clist,cliCount)) < 0){
        printf("err addmcastgrpmembersbyprinc: client %d not found\n",i);
        return -1;
        }

    addMcastGrpMember(groupIndex,princStr,clientID,clist[clientID].
        ipaddr);

    i++;
    }
////////////////////////////////////////////////////////////////////////

return 0;
}

//Add a multicast group to the GKS run-time *group* record
//in/out: curGroupIndex in: rest
//newGroupId need NOT be an index, is abstract
//curGroupIndex holds 'index of next group to be inserted'
int addMcastGroup(int newAbsGroupId, char *newGroupName,int *
    curGroupIndex,char keyBlobName[]){
int insertedIndex = -1;

if ((*(curGroupIndex)+1) < (MAXMCASTGROUPS-1)){

    //Global
```

```
    MCGrpList[*curGroupIndex].grpid = newAbsGroupId;
    MCGrpList[*curGroupIndex].size = 0;

    if (newGroupName != NULL){
        strncpy(MCGrpList[*curGroupIndex].grpName,newGroupName, strlen(
            newGroupName));
        //printf("addmcastgroup inserted group %s\n",MCGrpList[*
            curGroupIndex].grpName);
        }

    if (keyBlobName!= NULL)
        strncpy(MCGrpList[*curGroupIndex].sharedKeyBlobName,keyBlobName,
            strlen(keyBlobName));

    insertedIndex = *curGroupIndex;
    (*curGroupIndex)++;

    }
else {
    printf("err P4:addmcastgroup:exceeded max group count. Abort\n");
    return -1;
    }

return insertedIndex;
}

int getGroupIndex(int groupID,char *groupName){
int index = 0;

while ((index < MAXMCASTGROUPS) && (MCGrpList[index].size > 0) ){
    //if (MCGrpList[index].grpid == groupID)
    //  return index;

    if (groupName != NULL){
        if (!strcmp(groupName,MCGrpList[index].grpName))
          return index;
        }

    index++;
    }

//printf("err getgroupindex: group %d/%s does not exist\n. Abort\n",
    groupID,groupName);

return -1;
```

```
}

int initMcastState(void){
int i = 0;

for (i = 0;i<MAXMCASTGROUPS;i++){
   MCGrpList[i].size = 0;
   }

return 0;
}

//determine whether group 'groupID' (or "grpName"?) exists
//NOT assumed that groupIDs will be contiguous or sequential
//return index of group if exists, else -1;
int mcastGroupExists(int groupID,char grpName[]){
int i = 0;

i = getGroupIndex(groupID,grpName);

if (i > -1)
   return i;

return -1;
}

//For group groupID, fetch the list of member client ID's into
   memberList
//Requires: pre-allocated memory in groupPrincipalList, memberIDList
int getMcastGroupMembershipList(int groupID,char grpName[],int *
   memberIDList,char *groupPrincipalList[]){
int i = 0,grpInd = 0;
int grpsize = 0;

if ((grpInd = getGroupIndex(groupID,grpName)) < 0){
   printf("err getmcastgroupmembershiplist: Abort\n");
   return -1;
   }

grpsize = MCGrpList[grpInd].size;

for (i = 0;i<MAXCLIENTS;i++)
   *(memberIDList+i) = -1;

i = 0;
```

```
while (i<MCGrpList[grpInd].size){
    *(memberIDList+i) = MCGrpList[grpInd].clients[i].id;
    strncpy(groupPrincipalList[i],MCGrpList[grpInd].clients[i].
        princString,
      strlen(MCGrpList[grpInd].clients[i].princString));
    i++;
    }

return grpsize;
}

//openSSL: decrypt the ciphertext using the encryption key,IV into
    ptext
int p4OpensslDecAES(UCHR *ctext,int *ctextlen,UCHR *encKey,UCHR *encIV,
   UCHR *ptext){
////max ciphertext len for a n bytes of plaintext is n + AES_BLOCK_SIZE
    -1 bytes
int lastlen = 0;
int ptextlen = *ctextlen;
int finalPtextLen = 0;

EVP_CIPHER_CTX decCtxt;

////////////////////////////////////////////////
//Openssl decryption cipher init

EVP_CIPHER_CTX_init(&decCtxt);
EVP_DecryptInit_ex(&decCtxt, EVP_aes_256_cbc(), NULL, encKey, encIV);

EVP_DecryptInit_ex(&decCtxt, NULL, NULL, NULL, NULL);

EVP_DecryptUpdate(&decCtxt, ptext, &ptextlen, ctext, *ctextlen);
EVP_DecryptFinal_ex(&decCtxt, ptext+ptextlen, &finalPtextLen);

*ctextlen = ptextlen + finalPtextLen;

//Openssl finalize
EVP_CIPHER_CTX_cleanup(&decCtxt);

return 0;
}

//ctext must be large enough to hold enc{ptext}
```

```
int p4OpensslEncAES(UCHR *ptext,int ptextlen,UCHR *encKey,UCHR *encIV,
    UCHR *ctext){
//max ciphertext len for a n bytes of plaintext is n + AES_BLOCK_SIZE
    -1 bytes
int ctextlen = ptextlen + AES_BLOCK_SIZE, finallen = 0;
int lastlen = 0;
UCHR *ctextTmp = malloc(ctextlen);

////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
////Openssl: Encrypt ptext using encKey into ctext///////

////////////////////////////////////////////////////////////////
////Openssl cipher init
EVP_CIPHER_CTX enc_contxt;//openssl managerial sec op contexts
EVP_CIPHER_CTX_init(&enc_contxt);

//Init encryption: use symmetric key of group member
EVP_EncryptInit_ex(&enc_contxt, EVP_aes_256_cbc(), NULL,encKey,encIV);
EVP_EncryptUpdate(&enc_contxt, ctextTmp, &ctextlen,ptext, ptextlen);
EVP_EncryptFinal_ex(&enc_contxt, ctextTmp+ctextlen, &finallen);

lastlen = ctextlen + finallen;

memcpy(ctext,ctextTmp,lastlen);

//Openssl finalize
EVP_CIPHER_CTX_cleanup(&enc_contxt);

//free(ctextTmp);

return 0;
}

//gks socket send to client
//TGS_REP k_tgs/i{grpkey} k_tgs/i{grpname} k_grplongterm{sessgrpkey}
int gks_TGS_REP(int clientSock,char sessGrpKey[],int clientID,struct
    client clist[],int existingGroupID){
int numsent = 0;
char sendline[MSGLEN];
UCHR ptext[P4KEYSIZE];
UCHR grpNamePtext[P4GRPNAMELEN];
UCHR clientEncKey[P4KEYSIZE];
UCHR encGrpName[P4GRPNAMELEN+AES_BLOCK_SIZE];
UCHR encGrpKey[P4KEYSIZE+AES_BLOCK_SIZE];
```

```
UCHR encGroupPart[P4KEYSIZE+AES_BLOCK_SIZE];

memset(sendline,'\0',MSGLEN);memset(encGrpKey,'\0',P4KEYSIZE+
    AES_BLOCK_SIZE);
memset(encGroupPart,'\0',P4KEYSIZE+AES_BLOCK_SIZE);
memset(encGrpName,'\0',P4GRPNAMELEN+AES_BLOCK_SIZE);
memset(ptext,'\0',P4KEYSIZE);memset(clientEncKey,'\0',P4KEYSIZE);


////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
//Openssl: Encrypt session group key for client///////
p4OpensslEncAES(sessGrpKey,P4KEYSIZE,clist[clientID].shkey,iv_Dummy,
    encGrpKey);

//encrypt groupname
p4OpensslEncAES(MCGrpList[existingGroupID].grpName,strlen(MCGrpList[
    existingGroupID].grpName),clist[clientID].shkey,iv_Dummy,encGrpName)
    ;
////////////////////////////////////////////////////////


////////////////////////////////////////////////////////
//Openssl: Enc session grpkey using long term group key
//requires unseal
//TODO (P4KEYSIZE*2) long enough for unsealed blob?
UCHR longTrmGrpKey[TPMBLOBSIZE];
memset(longTrmGrpKey,'\0',TPMBLOBSIZE);

//p4TpmBlobUnseal(MCGrpList[existingGroupID].sharedKeyBlobName,&
    unsealedBlobLen,longTrmGrpKey);
//Reseal?

//The group session key encrypted with existing long term group key
p4OpensslEncAES(sessGrpKey,P4KEYSIZE,k_Dummy/*TODO longTrmGrpKey*/,
    iv_Dummy,encGroupPart);
////////////////////////////////////////////////////////


////////////////////////////////////////////////////
//Construct, send message "TGS_REP"
//k_tgs/i{grpkey} k_tgs/i{grpname} k_oldgrp{grpkey}
sprintf(sendline,"TGS_REP %s  %s  %s",encGrpKey,encGrpName,encGroupPart
    );
```

```
printf("gks send %s\n",sendline);

while ((numsent = send(clientSock,sendline,MSGLEN,0)) < 0){
   if (errno != EINTR){
     printf("GKS: error sending TGS_REP\n");
     //close(serverSock);
     continue;
     }
  }

return 0;
}

int gks_AP_REQ__REP(char *grpsesskey,int clientID,struct client *clist,
   char *principalListStr,char *grpName){
int serverSock = 0,ret = 0,numsent = 0,numrecvd = 0;
struct sockaddr_in serverAddr;
char sendline[MSGLEN],recvline[MSGLEN];
UCHR encGrpKey[P4KEYSIZE+AES_BLOCK_SIZE];

memset(sendline,'\0',MSGLEN);
memset(recvline,'\0',MSGLEN);
memset(encGrpKey,'\0',(P4KEYSIZE+AES_BLOCK_SIZE));

serverSock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(GKCLIENTPORT);

if(inet_pton(AF_INET,clist[clientID].ipaddr,&serverAddr.sin_addr) <= 0)
   {
  printf("gks inet_pton error\n");
  return 1;
  }

////////////////////////////////////////////
//connect to server (GK client)///////////////
while ((ret = connect(serverSock,(struct sockaddr *)&serverAddr,
        sizeof(serverAddr))) < 0){
   if (errno == ETIMEDOUT){
      printf(": server connect timed out\n");
      return 1;
      }
   else if (errno == ECONNREFUSED){
      printf(": server connection refused!\n");
      return 1;
```

```
      }
    else {
      printf("err: connect GK client %s fail: ?\n",clist[clientID].
         principalString);
      return 1;
      }
    }
////////////////////////////////////////////

////////////////////////////////////////////////////
//Send "AP_REQ" <grpname princlist> K_TGS/si[grpkey]

p4OpensslEncAES(grpsesskey,P4KEYSIZE,clist[clientID].shkey,iv_Dummy,
    encGrpKey);

sprintf(sendline,"AP_REQ %s %s %s",grpName,principalListStr,encGrpKey);

printf("gks send %s\n",sendline);

if ((numsent = sendn(serverSock,sendline,strlen(sendline)))<0){
    printf("GKS: error sending AP_REQ\n");
    return -1;
    }

////////////////////////////////////////////////////
//Await AP_REP ACK/CONF from group member//////////
//"AP_REP" <serverprincipal> K_TGS/si[grpkey]
while ((numrecvd = recv(serverSock,recvline,MSGLEN,0))<0){
    if (errno != EINTR){
        printf("GKS: error recv AP_REP: %s\n",strerror(errno));
        break;//return -1;
        }
    continue;
    }

printf("GKS recvd: %s\n",recvline);

////////////////////////////////////////////////////

close(serverSock);

return 0;
}
```

```
//Nokrb recv of TGS_REQ
int srv_Recv_AS_REQ(int socket,UCHR *recvLine,UCHR *newSharedKey){
;
//TODO
return 0;
}

int krb_Recv_AS_REQ(int *socket,krb5_auth_context *authc,krb5_ticket *
   tkt,krb5_context *con,krb5_keytab *kt,krb5_principal *srv){
krb5_error_code retval;

if (retval = krb5_recvauth(*con,authc,
    (krb5_pointer)socket, KRB_GKS_VERSION, *srv,
     0,                          /* no flags */
    *kt,          /* default keytab is NULL */
    &tkt) )
    printf("GKS(AS): err recvauth  %s\n",krb5_get_error_message(*con,
       retval));

return 0;
}

int gks_Recv_AP_REP(int socket,UCHR *encTimeStr){
int numrecvd = 0;
char recvline[MSGLEN];

memset(recvline,'\0',MSGLEN);

while ((numrecvd = recv(socket,recvline,MSGLEN,0))<0){
   if (errno != EINTR){
      printf("GKS: error recv AP_REP: %s\n",strerror(errno));
      //break;//return -1;
      }
   continue;
   }

printf("GKS recvd: %s\n",recvline);
// grpname k_i/tgs{principalname} k_i/tgs{timestamp}
sscanf(recvline,"AP_REP %*s %*s %s",encTimeStr);

return 0;
}

int gks_Send_AP_REQ(int socket,char *tgsAbsGrpNameStr,UCHR *grpKey,int
   clientID,struct client *clist){
```

```
int numsent = 0;
char sendline[MSGLEN];
UCHR *grpKeyEnc = (UCHR *)malloc(sizeof(UCHR)*(P4KEYSIZE+AES_BLOCK_SIZE
    ));

memset(sendline, '\0',MSGLEN);
memset(grpKeyEnc,0,P4KEYSIZE+AES_BLOCK_SIZE);

p4OpensslEncAES(grpKey,P4KEYSIZE, clist[clientID].shkey,iv_Dummy,
    grpKeyEnc);

sprintf(sendline,"AP_REQ %s %s",tgsAbsGrpNameStr,grpKeyEnc);
printf("gks send %s\n",sendline);

while ((numsent = send(socket,sendline,MSGLEN,0)) < 0){
     if (errno != EINTR){
        printf("GKS: error sending AP_REQ\n");
                  //close(serverSock);
                  continue;
        }
     }

free(grpKeyEnc);

return 0;
}

//Openssl: generate new 256 bit key
//nulls provided memory
//in/out: newKey in: rest
int openSSLGenKey(UCHR *newiv,UCHR *salt,UCHR *newKey){
int resultkeysize;
UCHR keyHexStr[P4KEYSIZE];

//keyHexStr = (UCHR *)malloc(sizeof(UCHR)*P4KEYSIZE);

//32B=256b
//3: salt: 8B buffer, 'p4gkgrp' for now, dummy sesskey bad randomness
resultkeysize = EVP_BytesToKey(EVP_aes_256_cbc(), EVP_sha1(),"p4gkgroup
    ",
                  sesskey_Dummy,P4KEYSIZE, P4KEYGENROUNDS, newKey, newiv)
                     ;
//printf("EVP_BytesToKey: %d bytes, key=%s\n",resultkeysize,getHexStr(
    newKey,P4KEYSIZE,keyHexStr));
```

```
return 0;
}

int srv_Recv_TGS_REQ(int socket,char *recvLine,char *grpName,int *
    grpSize,char *princListStr,char *cliPrinc,char *grpNameContainer){
int numrecvd = 0;
int msgItemMatch = 0;

////////////////////////////////////////////////
//"TGS_REQ" <grpname>    OR
//"TGS_REQ" <clientprinc> <grpname> <grpsize> <principal1> <principal2>
    ...

while ((numrecvd = recv(socket,recvLine,MSGLEN,0))<0){
    if (errno != EINTR){
        printf("GKS: error recv TGS_REQ: %s\n",strerror(errno));
        //break;//return -1;
        }
    continue;
    }

//ignore hash for now, 2 item of interest
msgItemMatch = sscanf(recvLine,"TGS_REQ %s %d %s %*s",cliPrinc,grpSize,
    grpName);

msgItemMatch = sscanf(recvLine,"TGS_REQ %*s %*s %s",grpNameContainer);
    //get <grpname grpprincipallist> for lookup

if (msgItemMatch = sscanf(recvLine,"TGS_REQ %*s %*d %*s %s",
    princListStr) < 1){
    printf("GKS err srv_recv_tgs_req bad parse\n");
    return -1;
    }

printf("gks recvd %s\n",recvLine);

return 0;
}

int main(int argc,char* argv[])
{
        FILE * f_ptr;
        char clihostname[HOSTNMLEN];
        char * cname, *sname;
        char keystr[MSGLEN], authstr[SHA1OUTSIZE];
```

```
    char method[30], clientIP[INET_ADDRSTRLEN], ARheader[50];
    char recvline[MSGLEN], sendline[MSGLEN];
    char resource[300], host[300], response_hdr[MSGLEN],
       response_body[MSGLEN];
char groupPrincList[3];
char *groupPrincListDummy[3];
    fd_set master; // master file descriptor list
    fd_set read_fds; // temp file descriptor list for select()
    int clientCount, numsent, contentlen, err, req_grpid, maxFD,
       readyFD;
    /* authSock is used for incoming KRB5 auth requests, gkuSock
       for grpKeyReq*/
    int i, grpsock, authSock, gkuSock, connfd, numrecvd,groupIndex;
    int requestedGroupIndex=0;
int groupIDList[MAXCLIENTS];
int *groupIDListDummy[3];
    socklen_t clientlen;
    struct client clientList[MAXCLIENTS];
    struct sockaddr_in kdcaddr, cliaddr, mcastaddr, authaddr;

    krb5_auth_context auth_context = NULL;
    krb5_error_code retval;
    krb5_principal server;
    krb5_ticket * ticket;
    krb5_context context;
    krb5_keytab keytab = NULL;
    krb5_keyblock * sKey; /* Used as the long term shared key */

clientCount = 0;
groupIndex = 0;

    ////////////////////////////////
    ////Init dummy static mcast group

 initMcastState();

 //addMcastGroup(0,"P4GRP",groupIDList,&groupIndex);

 //Hack: add the rcvr locally to client records
 //addClient("rcvr",k_Dummy,clientList,&clientCount,"127.0.0.1");

 ////////////////////////////////

    if ((gkuSock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)) < 0){
          printf("sserver: can't create socket. Aborting\n");
```

```
        return 1;
}

/*SO_REUSEPORT not available*/
i = 1;
if (setsockopt(gkuSock,SOL_SOCKET,SO_REUSEADDR,&i,sizeof(int))
   < 0)
        perror(": setsockopt");

////////////////////////////////////////////////
//Prepare kdc address structures
//
/*zero out the sockaddr struct*/
memset(&kdcaddr,0,sizeof(kdcaddr));

/*set parameters for the server address structure*/
kdcaddr.sin_family = AF_INET;
kdcaddr.sin_port = htons(GKSPORT);

/*currently use [localhost] KDC IP (network hosts)*/
kdcaddr.sin_addr.s_addr = htonl(INADDR_ANY);


if ((authSock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)) < 0){
        printf("GKS: can't create socket. Aborting\n");
        return 1;
}

/*SO_REUSEPORT not available*/
i = 1;
if (setsockopt(authSock,SOL_SOCKET,SO_REUSEADDR,&i,sizeof(int))
    < 0)
        perror(": setsockopt");


////////////////////////////////////////////////
//Prepare auth address structures
//
/*zero out the sockaddr struct*/
memset(&authaddr,0,sizeof(authaddr));

/*set parameters for the server address structure*/
authaddr.sin_family = AF_INET;
authaddr.sin_port = htons(AUTHPORT);
authaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
        /*
         *      INIT KRB5 VARS
         */

if (P4KRB){

        retval = krb5_init_context(&context);
        if (retval){
                // ERROR while initializing krb5
                return 1;
            }

        retval = krb5_sname_to_principal(context, NULL, KRB_SERVICE,
                                         KRB5_NT_SRV_HST, &server);
        if(retval){
                // ERROR while generating server princ name
                return 1;
            }
 }

        /*
         *      END - INIT KRB5 VARS
         */

        // clear the master and temp sets
        FD_ZERO(&master);
        FD_ZERO(&read_fds);


        ////////////////////////////////////////
        //Listen for group member/producer connection req//
        if (bind(gkuSock,(SA *)&kdcaddr,sizeof(kdcaddr)) == -1){
                // ERROR BINDING TO PORT
                return 1;
        }

        if (listen(gkuSock,LISTEN_Q) == -1){
                // ERROR LISTENING ON PORT
                return 1;
        }

        if (bind(authSock,(SA *)&authaddr,sizeof(authaddr)) == -1){
                // ERROR BINDING TO PORT
                return 1;
```

```
    }

    if (listen(authSock,LISTEN_Q) == -1){
          // ERROR LISTENING ON PORT
          return 1;
    }

    FD_SET(gkuSock, &master);
    FD_SET(authSock, &master);

maxFD = authSock;
if (gkuSock>authSock)
   maxFD = gkuSock;

    ////////////////////
    //Servicing request
    while (1){

       read_fds = master;
    //arg 5 timeout NULL, blocking select
       readyFD = select(maxFD+1, &read_fds, NULL, NULL, NULL);

       if(FD_ISSET(authSock, &read_fds)){ /* New Client KRB5 Auth
          */

          memset(&cliaddr,0,sizeof(cliaddr));
          clientlen = sizeof(cliaddr);

          if ((connfd = accept(authSock ,(SA *)&cliaddr,&clientlen
             )) < 0){
            printf("server accept error");
            continue;
            }

          /*store client IP address*/
          memset(clientIP,'\0',INET_ADDRSTRLEN);
          inet_ntop(AF_INET,&cliaddr.sin_addr,clientIP,
             INET_ADDRSTRLEN);
          printf("GKS(AS) accepted connection from %s\n",clientIP)
             ;

          char *clientPrincStr;
          UCHR cliTGSSharedKey[P4KEYSIZE];
          memset(cliTGSSharedKey,'\0',P4KEYSIZE);
          int sharedKeyLen = 0;
```

```
if (P4KRB){

        retval = krb5_recvauth(context, &auth_context, (
            krb5_pointer)&connfd,
                        KRB_GKS_VERSION, server,
                        0,    /* no flags */
                        keytab,       /* default keytab
                            is NULL */
                        &ticket);

        if (retval) {
              printf( "recvauth failed. go home man.
                  its over.[%ld]%s\n",
               (long) retval, error_message(retval));
              return 1;
           }

        retval = krb5_unparse_name(context, ticket->
            enc_part2->client, &clientPrincStr);
        retval = krb5_auth_con_getkey(context,
            auth_context, &sKey);

        printf("GKS: client principal name: %s\n",
            clientPrincStr);
        printf("Kerberos key length: %d bytes\n",sKey->
            length);

        printf("key[");
        for(i = 0; i < sKey->length; ++i){
            printf("\\x%X",(UCHR) sKey->contents[i]);
            }
        printf("]\n");


        krb5_free_ticket(context, ticket);
        krb5_auth_con_free(context, auth_context);

        sharedKeyLen = sKey->length;
        memset(cliTGSSharedKey,'\0',P4KEYSIZE);
        memcpy(cliTGSSharedKey,sKey->contents,sKey->
            length);

        //addClient(tgsClientPrincStr, finalSharedKey,
            clientList,&clientCount,clientIP);
```

```
            krb5_free_keyblock( context, sKey);
        }
        else {
            memset(recvline,'\0',MSGLEN);
            strncpy(cliTGSSharedKey,k_Dummy,P4KEYSIZE);
        }

        close(connfd);

        //Add the client info to the GKS *client* records
        //need principalnm, key from KRB TODO
        //addClient(clientPrincStr,finalSharedKey,clientList,&
            clientCount,clientIP);

        ////////////////////////////////////
        //Seal long term client/[KDC/GKS] shared key with TPM?
        //Encrypted blob written to disk

        //char blobname[P4KEYBLOBNAMELEN];
        //memset(blobname,'\0',P4KEYBLOBNAMELEN);
        //sprintf(blobname,"P4GKS.%s.ltsharedkey",cname);

        //p4TpmSeal(blobname,finalSharedKey,sKey->length);
        ////////////////////////////////////
}



if(FD_ISSET(gkuSock, &read_fds)){ /* Group Key Update REQ */
        ////////////////
        //"KRB_TGS_REQ" <grpname>, where grpname="<grpstring>"
            or
        //grpname="P4GRP" <princname1> <princname2> ... <
            princnamen>

        memset(&cliaddr,0,sizeof(cliaddr));
        clientlen = sizeof(cliaddr);

        if ((connfd = accept(gkuSock, (SA *) &cliaddr,&
            clientlen)) < 0){
            printf("server accept error\n");
            break;
            }
```

```
                /*store client IP address*/
                inet_ntop(AF_INET,&cliaddr.sin_addr,clientIP,
                    INET_ADDRSTRLEN);
                printf("GKS(TGS) accepted connection from %s\n",
                    clientIP);

                char *tgsAbsGrpName = (char *)malloc(sizeof(char)*
                    P4GRPNAMELEN);
                char *tgsStrictGrpName = (char *)malloc(sizeof(char)*
                    P4GRPNAMELEN);
                char *tgsClientPrincStr = (char *)malloc(sizeof(char)*
                    P4PRINCIPALNMLEN);
                char *tgsPrincListStr = (char *)malloc(sizeof(char)*
                    MSGLEN);
                char *tgsRecvLine = (char *)malloc(sizeof(char)*MSGLEN)
                    ;
                memset(tgsRecvLine,'\0',MSGLEN);
                memset(tgsAbsGrpName,'\0',P4GRPNAMELEN);memset(
                    tgsStrictGrpName,'\0',P4GRPNAMELEN);
                memset(tgsPrincListStr,'\0',MSGLEN);
                int grpSize = 0;

/*              if (P4KRB){
*/
               ////////////////////////////////////////////////
               //Krb5 sendauth server endpoint ////////////////
               //server NULL to force krb to allow P4 servername
               //ticket holds client request info
/*               if (retval = krb5_recvauth(context, &auth_context,
               (krb5_pointer)&connfd, KRB_GKS_VERSION, NULL,
               0,                          // no flags */
//               keytab,          /* default keytab is NULL */
//               &ticket))
/*                   printf("GKS(TGS): err recvauth \n");

               retval = krb5_unparse_name(context, ticket->server, &
                   sname);
               retval = krb5_unparse_name(context, ticket->enc_part2->
                   client, &cname);

               memset(tgsAbsGrpName,'\0',P4GRPNAMELEN);
               memset(tgsClientPrincStr,'\0',P4PRINCIPALNMLEN);

               strncpy(tgsAbsGrpName,sname,strlen(sname));
               strncpy(tgsClientPrincStr,cname,strlen(cname));
```

```
                krb5_free_unparsed_name(context,sname);
                krb5_free_unparsed_name(context,cname);

                //krb5 auth cleanup
                krb5_free_ticket(context, ticket);
                krb5_auth_con_free(context, auth_context);
                }
               else {
*/
                srv_Recv_TGS_REQ(connfd,tgsRecvLine,tgsStrictGrpName,&
                    grpSize,
                    tgsPrincListStr,tgsClientPrincStr,tgsAbsGrpName);

                //Hack: client should have AUTHd before now, but AUTH is
                     disabled
                addClient(tgsClientPrincStr,k_Dummy,clientList,&
                    clientCount,clientIP);

//                }

                ////////////////////////////////////
                //verify sig

                ////////////////////////////////////

            //TODO check if the client exists in records (should)


            ////////////////////////////////////////
            //Generate symmetric session key/////////
            //256bit AES TODO good randomness source for key input?
            UCHR grpsesskey[P4KEYSIZE];memset(grpsesskey,'\0',P4KEYSIZE);
            UCHR grpsessiv[P4KEYSIZE];memset(grpsessiv,'\0',P4KEYSIZE);
            openSSLGenKey(grpsessiv,NULL,grpsesskey);

            ////////////////////////////////////////
            ////////////////////////////////////////


            char grpKeyBlobName[P4KEYBLOBNAMELEN];
            memset(grpKeyBlobName,'\0',P4KEYBLOBNAMELEN);
            int newGrpIndex,requestorClientID;
```

```
requestorClientID = getClientID(tgsClientPrincStr,clientList,
    clientCount);

////////////////////////////////////////////////
//Check if GRP exists. If so, get members//////////
//pass in ENTIRE server string. If new group, will fail and
//grpname = <grpsize> <p1> <p2> ... <pn>

if ((requestedGroupIndex = mcastGroupExists(-1,tgsAbsGrpName))
    >-1){

        printf("GKS case 1\n");

        sprintf(grpKeyBlobName,"P4GKS.%s.%s.sharedgroupkey",
            tgsAbsGrpName,tgsClientPrincStr);

        //TGS_REP k_tgs/i{grpkey} k_tgs/i{grpname} k_oldgrp{
            grpkey}
        gks_TGS_REP(connfd,grpsesskey,requestorClientID,
            clientList,MCGrpList[requestedGroupIndex].grpid);

        //TODO seal new LTGK blob for this GRP+requestor
        //p4TpmSeal(grpKeyBlobName,grpsesskey,P4KEYSIZE);
        ////////////////////////////////////////////

        // add new group GKS storage
        newGrpIndex = addMcastGroup(groupIndex,tgsAbsGrpName,&
            groupIndex,grpKeyBlobName);

         //add the requestor to the mcast group records for new
             group
         addMcastGrpMember(newGrpIndex,tgsClientPrincStr,
             requestorClientID,
           clientList[requestorClientID].ipaddr);
         }
  else {//new group
        //"grpname" = <grpsize> <p1> <p2> ... <pn>

        if (grpSize < 1)
            break; //bad request

        sprintf(grpKeyBlobName,"P4GKS.%s.sharedgroupkey",
            tgsStrictGrpName);

        //Add new requested group to GKS mcastgroup records
```

```
 newGrpIndex = addMcastGroup(groupIndex,tgsStrictGrpName
     ,
                        &groupIndex,grpKeyBlobName);
 addMcastGrpMembersByPrinc(newGrpIndex,grpSize,
     tgsPrincListStr,
    clientList,clientCount);

 //add the requestor to the mcast group records for new
     group
 addMcastGrpMember(newGrpIndex,tgsClientPrincStr,
     requestorClientID,
   clientList[requestorClientID].ipaddr);

//TODO Perhaps wait to seal until after the distribution
    ?
//p4TpmSeal(grpKeyBlobName,grpsesskey,P4KEYSIZE);
////////////////////////////////////////////////

int tmpClientID;
UCHR encGrpKey[P4KEYSIZE+AES_BLOCK_SIZE];
UCHR encTimeStr[32+AES_BLOCK_SIZE];
memset(encTimeStr,'\0',(32+AES_BLOCK_SIZE));
memset(encGrpKey,'\0', (P4KEYSIZE+AES_BLOCK_SIZE));

p4OpensslEncAES(grpsesskey,P4KEYSIZE,clientList[
    requestorClientID].shkey,iv_Dummy,encGrpKey);

//AP exchange with requestor (member of new group)
gks_Send_AP_REQ(connfd,tgsAbsGrpName,grpsesskey,
    requestorClientID,clientList);

gks_Recv_AP_REP(connfd,encTimeStr);


////////////////////////////////////////////////
//Send "AP_REQ" <princlist> K_TGS/si[grpkey] AND recieve
    AP_REP
//to each proposed group server si who is a reciever (
    p4clientR must be running)

for (i = 0;i < MCGrpList[newGrpIndex].size ;i++) {
    if ((tmpClientID = getClientID(MCGrpList[newGrpIndex
        ].
        clients[i].princString,clientList,clientCount)) >
            -1){
```

```
            gks_AP_REQ__REP(grpsesskey,tmpClientID,
                clientList,tgsPrincListStr,
                MCGrpList[newGrpIndex].grpName);
        }
    }

//add the requestor to the mcast group records for new
    group
addMcastGrpMember(newGrpIndex,tgsClientPrincStr,
    requestorClientID,clientList[requestorClientID].
    ipaddr);

//send final TGS_REP to client (group established)
gks_TGS_REP(connfd,grpsesskey,requestorClientID,
    clientList,newGrpIndex);
}
////////////////////////////////////


        ////////////////////////////////
        //Authenticate TKT using requester/kdc shared
            key (sealed in TPM?)
        //N.B. memory allocation here intentional (TSS
            precondition)
        //BYTE *msghash = (BYTE *)malloc(sizeof(BYTE)*
            P4HASHSIZE);

        //UCHR msgTKT[MSGLEN];
        //UCHR str2[MSGLEN];

        //memset(msgTKT,'\0',MSGLEN);
        //memset(str2,'\0',MSGLEN);
        //memset(sendline,'\0',MSGLEN);
        //sprintf(msgTKT,"%d %s",req_grpid,keystr);

        //hash TKT, append to P4_MDIST
        //p4TpmHash(msgTKT, strlen(msgTKT),msghash);
        //p4TpmHMAC((BYTE *)k_Dummy,P4KEYSIZE,(BYTE *)
            msgTKT, strlen(msgTKT),msghash);
        //printf("p4server auth hash (hmac-sha1): %s\n
            ",getHexStr(msghash,P4HASHSIZE, str2));
        ////////////////////////////////


        ////////////////////////////////
```

```
                        close(connfd);
                }


        }

        return 0;
}
```

## C.7 Server Header

Listing C.7: Server Header

```
//#define WELLKNOWNPORT 5501 //defined elsewhere by Sean

/*
int getclientbyfd(int fd,int numclients,struct client *list)
{
int i;

for (i = 0;i < numclients;i++)
    if (list[i].sockfd == fd)
        return i;

return -1;
}

int getclientbypid(int pid,int numclients,struct client *list)
{
int i;

for (i = 0;i < numclients;i++)
    if (list[i].handlerpid == pid)
        return i;

return -1;
}

int pathtofile(char *path,char *file){
char *token,*tmptoken,*pathcopy;

pathcopy = (char *)malloc(sizeof(char)*300);
memset(pathcopy,'\0',300);
```

```
strcpy(pathcopy,path);

if (strstr(path,"/") != NULL){
   if ((token = strtok(pathcopy,"/")) != NULL){
     while ((tmptoken = strtok(NULL,"/")) != NULL){
       token = tmptoken;
       }

     strcpy(file,token);
     return 0;
     }
   }
else {
   return 1;
   }

return 0;
}
*/

int sendn(int sockfd,char *buf,int bytesToSend){
int bytesleft = bytesToSend,numsent = 0;
char *ptr;

ptr = buf;

while (bytesleft > 0)
  {
  if ((numsent = send(sockfd,ptr,bytesleft,0)) <= 0){
     if ((numsent < 0) && (errno == EINTR)){
       numsent = 0;
       }
     else {
       printf("client: sendn error\n");
       return -1;
       }
     }

  bytesleft -= numsent;
  ptr += numsent;
  }
return (bytesToSend-bytesleft);
}

int sendnf2(int sockfd,int n,FILE *fptr){
```

```
int chunk = 0;
int numsent = 1,totalbytesleft = n;
int totalsent;
unsigned char tmpbuf[MSGLEN];

totalsent = 0;

fread(tmpbuf,chunk,1,fptr);

while (totalbytesleft > 0){

  if ((numsent = send(sockfd,tmpbuf,chunk,0)) <= 0){
     if ((numsent < 0) && (errno == EINTR)){
        numsent = 0;
        }
     else {
       /*printf("server: sendnf error, numsent: %d\n",numsent);*/
       /*printf("buf: %s\n",tmpbuf);*/
       return -1;
       }
   }
  else if (numsent == chunk){
    chunk = MSGLEN;
    if (totalbytesleft < MSGLEN)
      chunk = totalbytesleft;
    memset(tmpbuf,'\0',MSGLEN);
    fread(tmpbuf,chunk,1,fptr);
    }
  else
    chunk -= numsent;

    totalbytesleft -= numsent;
    totalsent += numsent;
    }

return (n-totalbytesleft);
}

int sendnf(int sockfd,int n,FILE *fptr){
int chunk = 0;
int bytesleft = 0,numsent = 1,totalbytesleft = n;
int totalsent;
unsigned char tmpbuf[MSGLEN];

while (totalbytesleft > 0){
```

```
  memset(tmpbuf, '\0',MSGLEN);
  chunk = MSGLEN;
  if (totalbytesleft < MSGLEN)
     chunk = totalbytesleft;
  if (numsent <= 0)
     chunk = 0;

  fread(tmpbuf,chunk,1,fptr);
  bytesleft = chunk;

  totalsent = 0;
  while (bytesleft > 0){
    if ((numsent = send(sockfd,tmpbuf,chunk,0)) <= 0){
       if ((numsent < 0) && (errno == EINTR)){
          numsent = 0;
          }
       else {
         printf("server: sendnf error, numsent: %d\n",numsent);
         /*printf("buf: %s\n",tmpbuf);*/
         return -1;
         }
       }

    bytesleft -= numsent;
    totalsent += numsent;
    }

  totalbytesleft -= totalsent;
  }

return (n-totalbytesleft);
}

int recvn(int sockfd,char *dest,int bytesToGet){
int chunk = 0;
int bytesleft = bytesToGet,numrecvd = 0;
char *ptr;

ptr = dest;

while (bytesleft > 0){
  chunk = MSGLEN;
  if (bytesleft < MSGLEN)
     {chunk = bytesleft;}
```

```
  if ((numrecvd = recv(sockfd,ptr,chunk,0)) < 0){
     if (errno == EINTR)
       numrecvd = 0;
     else {
       printf("client: recvn recieve error\n");
       return -1;
       }
     }
  else if (numrecvd == 0)
     break;

  ptr += numrecvd;
  bytesleft -= numrecvd;
  }

return (bytesToGet-bytesleft);
}
```

## C.8 Utility Functions

Listing C.8: Utility Functions

```
//util.c common functions


//Pre: dest must have sufficient memory
char *getHexStr(UCHR dig[],int len,UCHR *dest){
        int i = 0;
        UCHR c[5];

        for (;i<len;i++){
                memset(c,'\0',5);
                sprintf(c,"%x",dig[i]);
                strcat(dest,c);
        }

        return dest;
}
```

# BIBLIOGRAPHY


[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. principles, implementations, and applications. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[2] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[3] George Berg and Sanjay Goel. Security Threats: Network Based Attacks. `http://www.cs.albany.edu/~berg/risk_analysis/Lectures/Security_Threats.pdf`.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corp., April 1977.

[5] Beth E. Binde, Russ McRhee, and Terrence J. O'Connor. Assessing outbound traffic to uncover advanced persistent threat. https://www.sans.edu/student-files/projects/JWP-Binde-McRee-OConnor.pdf, May 2011.

[6] S. Blake, D. Black, Davies Carlson, M., Z. E., Wang, and W. Weiss. An Architecture for Differentiated Services. *RFC 2475*, December 1998.

[7] Mihai Budiu, Ulfar Erlingsson, and Martin Abadi. Architectural support for software-based protection. Technical report, In Workshop on Architectural and System Support for Improving Software Dependability (ASID), October 2006.

[8] Mike Burmester, Joshua Lawrence, David Guidry, Sean Easton, Sereyvathana Ty, Xiuwen Liu, Xin Yuan, and Jonathan Jenkins. Towards a Secure Electricity Grid. In *Proceedings of the IEEE Eighth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP 2013)*. IEEE, 2013.

[9] Mike Burmester, Emmanouil Magkos, and Vassilis Chrissikopoulos. Modeling security in cyber-physical systems. *International Journal of Critical Infrastructure Protection*, 5:118–126, 2012.

[10] Ran Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science*, FOCS '01, pages 136–145, Washington, DC, USA, 2001. IEEE Computer Society.

[11] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, August 2014. USENIX Association.

[12] Microsoft Windows Dev Center. PAE/NX/SSE2 support requirement guide for windows 8. `http://msdn.microsoft.com/en-us/library/windows/hardware/hh975398.aspx`, June 2012.

[13] CERT. CERT vulnerability notes database: Advisory and mitigation information about software vulnerabilities. `http://www.kb.cert.org/vuls`.

[14] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41:3, July 2009.

[15] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *ACM Conference on Computer and Communications Security*, pages 39–50, 2008.

[16] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. A Protocol for Property-Based Attestation. In *STC '06: Proceedings of the First ACM Workshop on Scalable Trusted Computing*, pages 7–16, New York, NY, USA, 2006. ACM.

[17] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *In USENIX Security Symposium*, pages 177–192, 2005.

[18] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Computers and Communications, 2006. ISCC '06. Proceedings. 11th IEEE Symposium on*, pages 749 – 754, June 2006.

[19] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proc. of 13th Usenix Security Symposium*, 2004.

[20] David D. Clark and David R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, 1987.

[21] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting systems from stack smashing attacks with StackGuard. In *Linux Expo*, 1999.

[22] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Usenix Security Symposium*, 2003.

[23] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[24] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 119–129 vol.2, 2000.

[25] Jedidiah Crandall and F.T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*, 2004.

[26] Terry Cutler. The anatomy of an advanced persistent threat. `http://www.securityweek.com/anatomy-advanced-persistent-threat`, December 2010.

[27] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association.

[28] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, STC '09, pages 49–54, New York, NY, USA, 2009. ACM.

[29] Davi, Lucas and Sadeghi, Ahmad-Reza and Winandy, Marcel. Ropdefender: a detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM.

[30] Department of Homeland Security and Centre for the Protection of National Infrastructure. Cyber security assessments of industrial control systems, *control systems security program, national cyber security division*, November, 2010. `https://ics-cert.us-cert.gov/sites/default/files/documents/Cyber_Security_Assessments_of_Industrial_Control_Systems.pdf`.

[31] Solar Designer. Non-executable stack. http://www.openwall.com/linux/.

[32] Danny Dolev, Cynthia Dwork, Orli Waarts, and Moti Yung. Perfectly secure message transmission. *J. ACM*, 40(1):17–47, January 1993.

[33] Andrew Edwards, Hoi Vo, Amitabh Srivastava, and Amitabh Srivastava. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[34] Derek Bruening et al. DynamoRIO: Dynamic Instrumentation Tool Platform. http://dynamorio.org, February 2009.

[35] Hiroaki Etoh. Gcc extension for protecting applications from stack-smashing attacks. `http://www.research.ibm.com/trl/projects/security/ssp/`, June 2000.

[36] Simon N. Foley. A Non-Functional Approach to System Integrity. *IEEE Journal on Selected Areas in Communications*, 21(1):36–43, January 2003.

[37] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 575–589, Washington, DC, USA, 2014. IEEE Computer Society.

[38] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: A call graph execution profiler. *SIGPLAN Notices*, 39(4):49–57, April 2004.

[39] Trusted Computing Group. TPM specification architecture overview, revision 1.4. `http://www.trustedcomputinggroup.org/resources/tcg_architecture_overview_version_14`, August 2007.

[40] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd my gadgets go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 571–585, Washington, DC, USA, 2012. IEEE Computer Society.

[41] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 29–41, New York, NY, USA, 2006. ACM.

[42] International Electrotechnical Commission. IEC/TR 61850-1, First Edition, 2003-04, Technical Report, Communication networks and systems in substations. `http://webstore.iec.ch/preview/info_iec61850-1{ed1.0}en.pdf`, 2003.

[43] International Electrotechnical Commission. IEC/TR 61850-90-5, Edition 1.0 2012-05, Technical Report, Power systems management and associated information exchange – Data and communications security. `http://webstore.iec.ch/preview/info_iec61850-90-5{ed1.0}en.pdf`, May 2012.

[44] Jeremy L. Jacob. The basic integrity theorem. In *4th IEEE Computer Security Foundations Workshop (CSFW '91)*, pages 89–97. IEEE Computer Society Press, 1991.

[45] Jonathan Jenkins and Mike Burmester. Trusted Computing for Critical Infrastructure Protection Against Real-time and Run-time Threats. In *Proceedings of the Seventh Annual IFIP Working Group 11.10 Conference on Critical Infrastructure Protection*. Springer, 2013.

[46] Jonathan Jenkins, Sean Easton, David Guidry, Mike Burmester, Xiuwen Liu, Xin Yuan, Joshua Lawrence, and Sereyvathana Ty. Trusted Group Key Management For Real-Time Critical Infrastructure Protection. In *Proceedings of the Thirty-Second Annual Military Communications Conference (MILCOM 2013)*. Springer, 2013.

[47] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. `http://www.cs.columbia.edu/~vpk/research/libdft/`.

[48] Chongkyung Kil, E.C. Sezer, A.M. Azab, Peng Ning, and Xiaolan Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *IEEE/IFIP International Conference on Dependable Systems Networks, 2009. DSN '09*, pages 115–124, June 2009.

[49] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Softw. Pract. Exper.*, 24(2):197–218, February 1994.

[50] M. Lelarge and J. Bolot. Economic incentives to increase security in the internet: The case for insurance. *INFOCOM 2009*, pages 1494–1502, April 2009.

[51] Yehuda Lindell. General composition and universal composability insecure multiparty computation. *Journal of Cryptology*, 22(3):395–428, 2009.

[52] John Lambert Michael Howard, Matt Miller and Matt Thomlinson. Windows ISV software security defenses. `http://msdn.microsoft.com/en-us/library/bb430720.aspx`, December 2010.

[53] Vishwath Mohan and Kevin W. Hamlen. Frankenstein: Stitching malware from benign binaries. In *WOOT*, pages 77–84, 2012.

[54] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[55] C. Neuman, T. Yu, Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). *RFC 4120*, July 2005.

[56] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium*, 2005.

[57] National Institute of Standards and Technology. National vulnerability database: automating vulnerability management, security measurement, and compliance checking. http://web.nvd.nist.gov.

[58] Aleph One. Smashing the stack for fun and profit. `http://www.phrack.org/archives/49/p49_0x0e_SmashingTheStackForFunAndProfit_by_Aleph1.txt`, 1996.

[59] Nick L. Petroni, Jr., Timothy Fraser, Aaron Walters, and William A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[60] Perl 5 Porters. perlsec - perldoc.perl.org. http://perldoc.perl.org/perlsec.html#Taint-mode.

[61] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks. In *in Proceedings of the ACM Special Interest Group on Operating Systems*, 2006.

[62] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. *45th Annual IEEE/ACM International Symposium on Microarchitecture*, 0:135–148, 2006.

[63] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1):2:1–2:34, March 2012.

[64] scut/team teso. Exploiting format string vulnerabilities. `http://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf`.

[65] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[66] S. Shenker, R. Braden, and D. Clark. Integrated Services in the Internet Architecture: An Overview. *IETF Request for Comments (RFC)*, 1633, 1994.

[67] SISCO. CISCO and SISCO Collaborate on Open Source Synchrophasor Framework, Press Release. `http://www.sisconet.com/downloads/90-5_Cisco_SISCO.pdf`, 2011.

[68] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[69] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 85–96, New York, NY, USA, 2004. ACM.

[70] TCG. TPM main specification. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, March 2011.

[71] TCG. Trusted Network Connect Architecture for Interoperability, Specification 1.5, Revision 3. http://www.trustedcomputinggroup.org/resources/tnc_architecture_for_interoperability_specification, May 2012.

[72] The PaX Team. Homepage of the pax team. http://pax.grsecurity.net/.

[73] Trusted Computing Group (TCG). http://www.trustedcomputinggroup.org/.

[74] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254. IEEE Computer Society, 2004.

[75] Vendicator. Stack shield a "stack smashing" technique protection tool for linux. http://www.angelfire.com/sk/stackshield/info.html.

[76] David W. Wall. Systems for late code modification. In Robert Giegerich and Susan L. Graham, editors, *Code Generation Concepts, Tools, Techniques*, Workshops in Computing, pages 275–293. Springer London, 1992.

[77] Ben Wun. Survey of software monitoring and profiling tools. http://www.cse.wustl.edu/~jain/cse567-06/ftp/sw_monitors2/index.html.

[78] Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis. Taint-exchange: A generic system for cross-process and cross-host taint tracking. In *In Proceedings of the 6th IWSEC*, pages 113–128, 2011.

[79] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, Washington, DC, USA, 2013. IEEE Computer Society.

[80] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 129–140, New York, NY, USA, 2014. ACM.

# BIOGRAPHICAL SKETCH

Jonathan Jenkins

Computer Science

Jonathan Jenkins earned his Bachelor of Fine Arts degree from the University of Central Florida in 2004. He received his Master of Science degree from Florida State University in 2008. He joined the doctoral program at Florida State University in 2009.

Jonathan has been the recipient of a College Teaching Fellowship, an Information Systems Security Professionals certificate, and a Teaching Assistant award.

While pursuing his degree, Jonathan worked as a Teaching Assistant and as an intern at General Dynamics Corp.

Jonathan has presented his research at department conferences and in conferences such as the IEEE Military Communication Conference and the IFIP Working Group 11.10 on Critical Infrastructure Protection.

Jonathan's dissertation, *Building Trusted Computer Systems via Unified Software Integrity Protection*, was supervised by Dr. Mike Burmester.