Cryptoprogramming:
A Software Tamper Resistant Mechanism Using Runtime Pathway Mappings*

Willard Thompson, Alec Yasinsac, J. Todd McDonald†
Department of Computer Science
Florida State University
Tallahassee, FL 32306, U.S.A.
{wthompso, yasinsac, mcdonald} @cs.fsu.edu

Abstract *Mobile code suffers from the malicious host problem. When an adversary receives code he is able to effectively tamper with the code if he is able to relate the operations of the program with the appropriate context, that is, understanding the semantics of the program. In order to thwart an adversary from effectively tampering with a program he must be given an encrypted version such that he may be able to observe its operations but not understand why those operations are performed. Our notion of encrypting a program in such a way is to semantically alter it. In this paper, we add a White-box dimension called Cryptoprogramming to the Black-box notion of our Semantic Encryption Transformation Scheme. We construct an encrypted program by transforming the runtime logical pathways of the original program into a nonequivalent set of corresponding runtime logical pathways, and yet still allow for easy recoverability of the output of the program.*

1 Introduction

Mobile code plays a key integral part with advancing the growth of distributed systems. Not only is data a valuable commodity in a distributed system, but also programs, that can traverse a network performing specific operations, have significantly added to the distributed computing paradigm. However, the usage of mobile code has been hampered by the malicious host problem, which has stunted the progress of the utilization of mobile agents and other forms of mobile code. The malicious host problem emanates from the fact that the execution environment of the mobile code has total control, and can, not only invade the privacy of mobile code but affect its integrity as well. Ideally, Alice constructs a mobile program that she wants to run on a remote host Bob, with the intention of attaining the output of that program. It is required that Bob cannot know how the meaning of the program relates with its supposed context in which it runs. However, if Bob is malicious, he could effectively tamper[1] with the code, since Bob is able to visually inspect, statically analyze and dynamically test the code[2]. In order for Alice's mobile code to survive it needs to be protected, even if Bob understands what the program does via White-box and Black-box exploits.

* This work was supported in part by the U.S. Army Research Laboratory and U.S. Army Research Office under grant number DAAD19-02-1-0235.
† The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.
[1] By effectively tamper (as opposed to blind tamper) we mean that Bob is able to maliciously alter the code for some known benefit. Furthermore, assuming only software mechanisms, it should be intuitive that it is impossible to protect against blind tampering since the execution environment contains the program in its own memory space.
[2] In practice, these operations are commonly used in reverse engineering, however, our basis for Crytpoprogramming does not address securing directly against reverse engineering.

Our goal is, given the same set of inputs we provide a transformation that maps the outputs of an original program p to a pseudo-randomly scrambled and nonequivalent set of outputs of its dual encrypted program p'. Since upon a given input, outputs are derived from a particular sequence of instructions within a program during runtime, we provide a mapping of the runtime logical pathways in p to p'. We argue that this mapping will minimize the impact of White-box analysis by a malicious host, since the answers a malicious host looks for would be held secret by the originator of the mobile code. Building upon the intuitive notion that an algorithm is a solution that maps inputs to outputs and likewise, a program is an implementation of an algorithm, if the set of outputs of p' are pseudo-randomly scrambled and nonequivalent to the set of outputs for p, for each corresponding input, then it would be computationally infeasible to find a collision that is, an I/O pair of p that equals an I/O pair of p'. As a result, we argue without proof that it is then computationally infeasible for an adversary to be able to deduce the semantics of the original program from its dual, thus not allowing the adversary to be able to effectively tamper with the code.

The remainder of this paper is organized in the following way: We briefly discuss related work in section 2. Section 3 identifies our objective. In section 4 we briefly go over the Semantic Encryption Transformation Scheme (SETS). In section 5, we discuss Cryptoprogramming and finally in section 7 we conclude.

2 Related Work

We briefly summarize the main aspects of the related work for semantically encrypting programs. The most relevant work is that of [WT04], which describes our notion of Black-box program protection. We show that it is computationally infeasible for an adversary to deduce the semantics of the program, when the outputs of the program are sent through a data encryption cipher, such as 3DES or AES.

Mobile Cryptography is another area of related work. Mobile Cryptography coined by Sander and Tschudin [TC98, TS98, TS97], aims to provide provable security for mobile code. Sander and Tschudin have attempted to generalize Mobile Cryptography into what is called Computing with Encrypted Functions (CEF) [TC98, TS98, TS97]. CEF refers to a process where a function is transformed into a different function that is embedded into a program, protecting the original intent, yet still produces a result that can be decrypted only by the originator. CEF can be realized via homomorphism and functional composition. Although Mobile Cryptography is ideal on the surface, it is not practical for a general program, primarily since it is limited to some mathematical computations.

There are other types of program encryption mechanisms related to Mobile Cryptography, that incorporate computational models such as circuits and Turing machines. For instance, Adadi and Feigenbaum [MA90] have used Boolean circuits for describing Computing with Encrypted Data (CED). CED is process by which data, which is encrypted using provably secure means, can be sent to a remote host for computation. Also, Brekne [TB01], who calls encrypted programs "functional ciphertext," has begun to explore possibilities in encrypted representations of automata using functional compositions and has arrived at some interesting challenges for universal Turing transformations.

Code obfuscation, although does not aim to semantically alter the program, is another related area for protecting programs. Code obfuscation alters the syntax of a program into some confused form that is actually another representation of the same functionality [BB01, CC98]. The goal of obfuscation is to increase the adversary's cost during reverse engineering. Collberg, et. al. [CC98] provide seminal work for code obfuscation. They describe control-flow transformations with respect to resilience, stealth, potency and cost. Collberg, et. al. describes resilience in terms of inserting opaque predicates into code. Opaque predicates are Boolean expressions whose values are difficult to ascertain during automatic deobfuscation, but are known to the obfuscator. Other forms of code obfuscation consist of Ng and Cheung's [SN99] "intention spreading", which increases the amount of dummy code in a program, and time-limited Black-box protection mechanism [FH98], which allows for a time period for which a program is protected based on the strength of obfuscation, where upon the expiration of the allotted time, the agent becomes invalid.

Finally, another means for protecting programs is the utilization of declarative programming languages such as the Seal-Calculus [JV99, JV98]. However, these languages are limited in that they normally require trusted computing environments and it is difficult to apply these concepts with their restrictions for general imperative programs.

3 Objective

Since an adversary has at his arsenal the transformed program p' and the general context in which p' is to be executed, he may be able to find out the purpose of the program. Hence, it is our aim to keep the purpose of the program confidential in order to thwart effective tampering. We have shown in [WT04] that from exclusive Black-box protection it is computationally infeasible for an adversary to be able to deduce the semantics of the original program. However, in order to achieve solid program confidentiality, not only are Black-box protection mechanisms required, but also robust White-box protection measures should be applied. Thus, our objective is to fulfill our notion of Black-box security for programs via a generalized White-box transformation approach. In particular, we provide a mapping between the I/O relationships of p and p', that is, the sequence of program instructions that are executed during runtime for p, for a given input, are mapped to a semantically different sequence of instructions of p' for the same input. Furthermore, we address the following cryptanalytic and exploitation operations:

(1) Deducing the semantics of the original program
(2) Effectively tampering with the code

We assert that the second operation is dependent upon the first. The intuition is that as the adversary learns more about the semantics of the original program, that is, making the connection of the program with the context, he knows where and how to exploit the code to his benefit. Because of this relationship, we relate the effectiveness of code tampering by how much the adversary can learn from the encrypted program p'.

Finally, we focus on mobile code that is non-interactive to not only Alice but also to Bob. In other words, the mobile code, while executing at the remote host, merely receives an input and computes an output that is returned to Alice. In contrast to, a program having multiple interactions with the remote host, such that the remote host uses the output(s) during a single execution.

4 SETS Concept

Before going over the foundation of Cryptoprogramming, we briefly go over the concept of SETS [WT04], so as to provide the practical context of our notion of program encryption.

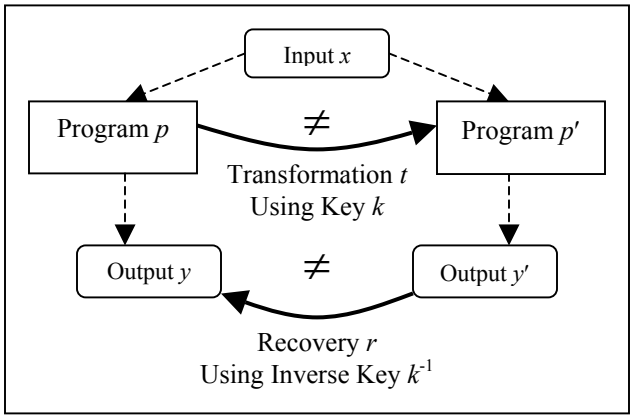

Figure 1. *Semantic Encryption Transformation Model.*

Figure 1 reflects the SETS approach. The dotted arrowed lines represent data flow and the solid arrowed lines represent program transformation and data recovery. A program is encrypted by Alice ($p' = t(p, k)$), and sent to Bob, with the result, $y' = p'(x)$, being returned to Alice and decrypted ($y = r(y', k^{-1})$). Note that the adversary only has in his possession x, p' and y'. His goal is to deduce p, from p', and key k to determine how p and p' are related. Thus, our goal is to minimize the amount of information of p' that the adversary can use to deduce p or k.

5 Cryptoprogramming

We now discuss the notion of Cryptoprogramming. We have shown in [WT04] that it would be computationally infeasible for an adversary to be able to deduce the semantics of the original program if the outputs of the original program were concatenated with a data encryption cipher all encompassed in a Black-box. Since an adversary can circumvent Black-box protection via code observation, we need to consider (its corresponding) White-box protection approaches. Although, since observing the algorithm for operations of a data cipher process such as DES with respect to Xor and substitution operations (and even more so for a public key cipher process such as RSA, considering its modular and exponential arithmetic), generally, an adversary could be able to spot those characteristics within a program, unless either the program is reduced to these data encryption operations where they are confluent and conform, or these data encryption operations are altered in such a fashion that they are transparent to the constructs and characteristics of the program and likewise conform. Since either of these operations are not known to exist, we employ a scaled-down version of data encryption to primarily demonstrate our notion of runtime pathway mapping.

Our transformation consists of mapping runtime logical pathways[3] of the original program to the encrypted program. We first state an important restriction of a program's input in section 5.1.

[3] By a runtime logical pathway we mean the sequence of instructions that are executed during runtime mapping a given input to its corresponding output.

We then discuss our notion of semantic nonequivalence in section 5.2. In section 5.3 we discuss the basis of our logical pathway mappings for semantically transforming programs, and finally, in section 5.4 we give a toy example to further describe our logical pathway transformation concept.

5.1 Program Inputs

We require that the inputs to a program are known a priori, that is, we restrict our programs to only a certain well-known possible input set. We consider aspects of computability theory, namely that in principle if the set of inputs is effectively enumerable, then we can compute or rather construct a program that can account for each of those inputs [MS97]. However, if the set of inputs are not enumerable, such as the set of real numbers, then our constructed program must contain restrictions on the input space, such as only allowing inputs to contain five decimal places[4].

Another view on inputs that we claim is that it is generally impossible to hide Bob's input from Alice. Some aspects of Mobile Cryptography not only aims to hide the decrypted result from a remote Bob, but also aims to hide the remote input from Alice [MA90, TC98, TS98, TS97]. However, we can undermine the notion of protecting inputs from Alice, by allowing Alice to compute $f(x) = x$. In this case Bob is unable to keep his input secret, since $f(x)$ would be encrypted and Bob would not know exactly what $f(x)$ does. Since it is impossible to effectively secure input x, we also allow for Alice to know Bob's particular input after runtime. At best Bob may choose to not compute the encrypted version of $f(x) = x$, if he wants to hide his input.

5.2 Semantic Program (Non)Equivalence

In order to discuss the semantics of p and p', we must first define the semantics of a program. We take on the accepted definition of operational semantics, where the semantics of a program is the set of sequences of computational steps producing a certain output from a specific input. We first describe what it means when two syntactically distinct programs are semantically equivalent and then convey what is required for semantic nonequivalence between p and p'.

When are two programs considered semantically equivalent? Given a set of programs P, such that any program in P implements the same algorithm A, we define two programs p_i and p_j as semantically equivalent, such that p_i and $p_j \in P$ and $p_i(x) = p_j(x) \ \forall \ x \in X$. In making our definition more precise, we may note that when $p_i(x) \neq p_j(x)$, for one or more distinct inputs $x \in X$, then p_i and p_j are not semantically equivalent.

We require that p and p' be semantically nonequivalent. However, our current definition of nonequivalence, as it stands, is not strong enough for p and p' to avoid collisions. For instance, if only one out of a number of outputs of p' do not equal the output of p for the same set of inputs, then if the adversary decides to conclude that he has deduced the original program from the encrypted program, his conclusion could be close enough to being correct for his purpose, even though it is not correct by our current definition of semantic equivalence. Hence, we require that

[4] Even though the set of natural numbers are considered effectively enumerable, we would still have to restrict this set if it was the input space. Moreover, in practice we don't often see situations allowing an infinite amount of inputs into a program.

a greater than polynomial number of inputs produce non-equivalent outputs of p and p' to avoid collisions.

5.3 Logical Runtime Pathway Mapping

We now describe our notion of logic runtime pathway mapping[5]. We first discuss the I/O relationships of the original program p and its corresponding encrypted dual p' a priori. We use the intuitive definition of a logical pathway through a program as a sequence of instructions of the program, executed during runtime, taking an input to its corresponding output[6]. We use the symbol δ_i to designate the i^{th} logical runtime pathway through a program. We have that, given a program p and its encrypted dual p', there exists two sets of nonequivalent I/O pairs, δ and δ', corresponding to p and p', respectively. Given $x_i \in X$, $y_i \in Y$, and $y_i' \in Y'$, such that it is computationally difficult to find a collision, that is, $y_i = y_i'$, we can conceptually view these two sets, δ and δ', within a table as follows[7]:

δ			δ'		
δ_0: x_0	$\rightarrow$	y_0	δ_0': x_0	$\rightarrow$	y_0'
δ_1: x_1	$\rightarrow$	y_1	δ_1': x_1	$\rightarrow$	y_1'
δ_2: x_2	$\rightarrow$	y_2	δ_2': x_2	$\rightarrow$	y_2'
δ_3: x_3	$\rightarrow$	y_3	δ_3': x_3	$\rightarrow$	y_3'
.		.	.		.
.		.	.		.
.		.	.		.
δ_n: x_n	$\rightarrow$	y_n	δ_n': x_n	$\rightarrow$	y_n'

***Table 1**, I/O Pairs for Both p and p'.*

We note that since the inputs are the same for δ and δ', and that the corresponding outputs are nonequivalent, these outputs can be mapped by simply going across the table in each row, for instance, y_0' is mapped to y_0, via x_0, and so forth. So, when Alice receives an encrypted output y_i', she can simply view her table to see the corresponding non-encrypted output y_i. Thus, for the recovery process, Alice can easily retrieve the truly intended result by examining this relationship. After discussing the relationship of p and p', we are now left with how to construct[8] p' such that:

(1) The outputs of δ' and δ are nonequivalent (Semantic nonequivalence)
(2) The outputs of δ' are pseudo-random (Black-box effect)

[5] We use the term mapping loosely, that is, not in the mathematical sense but representative of the correspondence between the elements of δ and δ'.
[6] We use the phrases 'logical pathway' and 'I/O pair' interchangeably.
[7] We envisage that the execution of both programs, with the same inputs, would produce such a table.
[8] As long as the input set and its corresponding output set are effectively enumerable then our transformation is computable [MS97]. Even if the input set is theoretically infinite, we can still arrange the elements of the input set into a countably infinite set, as long as there exists a mapping from each input element to a unique natural number.

5.3.1 Intermediate Form of *p*

We breakdown the original program *p* into individual runtime pathways that correspond to each input, that is, we list the actual instructions that execute during runtime for each input. The graph in Figure 6 describes the conceptual relationship between the collection of inputs in X and the possible outputs in Y and Y′. This graph is conceptually similar to the SETS conceptual model above in Figure 1 above. Just as edges are defined by two vertices within a graph, we let the edges δ and δ′, represent the relationship between X and Y, and X and Y′, respectively. The cardinality of the number of logical pathways through the transformed program *p*′ must be equal to or greater than the cardinality of the logical pathways of the original program *p*, that is $|\delta'| \geq |\delta|$ in order to provide a mapping for all possible outputs of *p*.

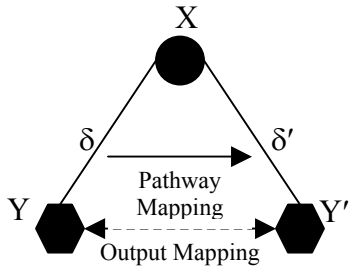

Figure 2. *Conceptual View of the I/O Relationship between p and p′.*

We define each component of Figure 2 in the following way:

X : Set of input values.

δ : The collection of logical pathways through the original program *p*.

δ′ : The collection of logical pathways through the transformed program *p*′.

Y : The collection of outputs of the original program *p*.

Y′ : The collection of outputs of the transformed program *p*′.

5.3.2 Semantic Nonequivalence and Black-box Effect

After listing each logical pathway of the intermediate representation of the original program *p*, we semantically alter its computation achieving semantic nonequivalence, as described in section 5.2, which addresses our first goal above in section 3. The logical pathway alterations are constructed in such a way that the outputs are produced pseudo-randomly. These semantic alterations consider White-box scrambling mechanisms resulting in pseudo-random output that is, interleaving data encryption scrambling mechanisms throughout the logical pathways that will ensure random outputs. We attempt to employ the data encryption properties of transpositions and substitutions by illustrating this process with a toy example.

5.4 Toy Example

We illustrate the concept of logic runtime pathway mapping along with the notion of interleaving a data cipher to produce pseudo-random results by using a toy example. We accomplish this goal

by providing an example of a toy program that manipulates strings[9]. In particular, we arbitrarily allow our toy program to simply reverse the input string and append one or more terminating markers. If the input string length (the number of characters in the input string) is greater than four then we append two terminating markers. A terminating marker is designated by the # sign. For the sake of brevity our toy program is only allowed to accept the following five input strings, that is, upon receiving another input string the program will not compute it:

Input Set X
"hello"
"abcd"
"xyz"
"12345"
"abaabbbaaba"

Table 2. *Possible Inputs for Programs p and p′.*

The pseudo-code of the original program *p* for our toy example is as follows:

```
1.     Operation(input string)
2.            Precondition: Accept only those strings from the list above.
3.            output returnString = ""
4.            Reverse string
5.            Assign string to returnString
6.            if (string length < 5) then
7.                   Append # to returnString
8.            else
9.                   Append ## to returnString
10.           return outputString
11.    End Operation
```

Listing 1. *Program p.*

From the toy program above, we can count two possible logical pathways through the code[10]. We note that the number of intermediate mappings may be equal to or greater than the number of logical pathways of the original program. As a result of each pathway, we get the following I/O mappings:

[9] We often see computations that manipulate strings when working with computational models such as automata and Turing machines.
[10] Measuring the number of logical pathways through code is formally known as the McCabe's Cyclomatic Complexity.

Input X	Output Y
1. hello	olleh##
2. abcd	dcba#
3. xyz	zyx#
4. 12345	54321##
5. abaabbbaaba	abaabbbaaba##

Table 3. *Logical Pathway Mapping I/O for p, δ.*

For brevity we do not list all of the actual logical pathways here, instead they are implied via the I/O pairs in Table 3. For instance, with the first input word, `hello`, we execute instructions 1, 2, 3, 4, 5, 6, 8, 9, 10, and 11, as shown in Listing 1, resulting in output `olleh##`. Thus, after enumerating the logical pathways through program *p* for each input, we semantically alter *p* by hand to produce the following encrypted program *p′*:[11]

```
1.     Operation(input string)
2.           Precondition: Accept only those strings from the list above.
3.           output returnString = “”
4.           if (string = “hello” or string = “abcd”) then
5.                 Eliminate letter in position 2 (hllo, acd)
6.                 Append “dxutyz” (hllodxutyz, acddxutyz)
7.                 Assign to returnString
8.           else if (string = “xyz”) then
9.                 loop 3 times
10.                      Swap first letter with the second
11.                      Truncate first letter
12.                      Append letter “t1”
13.                (for xyz we eventually get: x1t1t1)
14.                Assign to outputString
15.          else if (string = “12345”) then
16.                Replace every even positioned character with y
17.                Replace every odd positioned character with x (xyxyx)
18.                Assign to outputString
19.          else if (string = “abaabbbaaba”) then
20.                loop 3 times
21.                      Eliminate letter in position 2
22.                      Reverse string
23.                      Swap third letter with fourth letter
24.                (we eventually get: aababababa)
25.                Assign to outputString
26.          return outputString
28.    End Operation
```

Listing 2. *Program p′.*

[11] At minimum our heuristics based approach satisfies requirements of semantic nonequivalence with respect to producing nonequivalent I/O pairs between *p* and *p′*.

This encrypted program *p′* in Listing 2, thus, yields a new set of I/O pairs as shown in Table 4:

Input X	Output Y′
1. hello	hllodxutyz
2. abcd	acddxutyz
3. xyz	x1t1t1
4. 12345	xyxyx
5. abaabbbaaba	aababaaba

Table 4. *Logical Pathway Mapping I/O for p′, δ′.*

Putting the two sets of pairs, δ and δ′, together into a table we have the following:

Output Y′	Output Y	Input X
1. hllodxutyz	olleh##	(via hello)
2. acddxutyz	dcba#	(via abcd)
3. x1t1t1	zyx#	(via xyz)
4. xyxyx	54321##	(via 12345)
5. aababaaba	abaabbbaaba##	(via abaabbbaaba)

Table 5. *Showing sets δ and δ′.*

As we can see from Table 5, the encrypted results under Y′ map to the final results under Y. This table conceptually expresses our logical runtime pathway mapping from program *p* to its encrypted dual *p′*. So, for instance, we map the instruction sequence from Table 3 above for input word "hello", namely the instructions 1, 2, 3, 4, 5, 6, 8, 9, 10, and 11 of *p*, to its corresponding encrypted version from Table 4 for the same input word "hello", namely the instructions 1, 2, 3, 4, 5, 6, 7, 26, and 28, from *p′*. The benefit of this kind of mapping allows us to formalize and explicitly show the nonequivalent relationship of I/O pairs between programs *p* and *p′*.

6 Reducing Effective Tampering to Blind Disruption

We finally make a connection with the second issue of our objective, reducing effective tampering to blind disruption. After describing the concept of logical runtime pathway mapping and providing a toy example, we are led to the following assertion:

> *If an adversary is unable to efficiently deduce the correct semantics of the original program p, then he is unable to effectively tamper with the encrypted program p′.*

Since we construct an encrypted program p' using the White-box transformation of logical runtime pathway mapping as described above, we have that many output elements of δ' are semantically nonequivalent to their corresponding output elements of δ. Moreover, assuming that each output element of δ' is computationally indistinguishable from any other output element of δ', when only given p', an adversary is unable to deduce a pattern and hence the semantics of p efficiently. Therefore, by not knowing the semantics of the (original) program, we claim that the adversary is unable to effectively tamper with the program.

7 Conclusion

We have provided another starting point for protecting programs by transforming them into semantically different programs via logical runtime pathway mappings, allowing the originator to easily recover the final intended result. Even after successful reverse engineering, the adversary still has to attempt to understand how the program relates to the context for which it is supposed to run in. In other words, following de-compilation and de-obfuscation, the adversary can clearly observe the instructions of the program, however, by our notion of semantic transformation the adversary is still not able to effectively tamper with it, since he does not understand the purpose of the program. We have used a toy example to further illustrate our notion of White-box transformation of runtime pathway mappings.

We emphasize that we are taking this one step at a time, and we acknowledge that despite the idea of runtime pathway mapping being a start on a potentially new technique, a systematic approach is desirable. Moreover, our notion of Cryptoprogramming in this paper is more of an a priori account of the concept of semantically altering a program. In other words, given our aim to construct an encrypted program as we have shown, we have provided another conceptual representation of our goal that may augment our research into solving the malicious host problem. Thus, with more empirical data the Cryptoprogramming concept may evolve with time.

Bibliography

[BB01] B. Barak, et. al., "On the (Im)possibility of Obfuscating Programs", Electronic Colloguium on Computational Complexity, Report No. 57, 2001.

[CC98] C. Collberg, et. al., "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", In Proc. 25th ACM Symposium on Principles of Programming Languages, pp. 184-196, 1998.

[FH98] F. Hohl, "Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts", LNCS, Springer Verlag, pp. 92-113, 1998.

[JV99] J. Vitek and G. Castagna, "Mobile Computations and Hostile Hosts", Journ'ees Francophones des Langages Applicatifs, pp. 113-132, 1999.

[JV98] J. Vitek and G. Castagna, "Towards a Calculus of Secure Mobile Computations", In Proceedings Workshop on Internet Programming Languages, LNCS, 1686, Springer, 1998.

[MA90] M. Abadi and J. Feigenbaum, "Secure Circuit Evaluation: A Protocol Based on Hiding Information from an Oracle", Journal of Cryptology, Vol. 2, No. 1, pp. 1-12, 1990.

[MS97] M. Sipser, *Introduction to the Theory of Computation*, United States: PWS Publishing Company, 1997.

[SN99] S. Ng and K. Cheung, "Protecting Mobile Agents Against Malicious Hosts by Intention Spreading", International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 725-729, 1999.

[TB01] T. Brekne, “Encrypted Computation”, Department of Telematics, Ph.D. Dissertation, Norwegian University of Science and Technology, 2001.

[TS98] T. Sander and C. Tschudin, “On Software Protection Via Function Hiding”, LNCS, Vol. 1525, pp. 111-123, 1998.

[TS97] T. Sander and C. Tschudin, “Protecting Mobile Agents Against Malicious Hosts”, LNCS, Vol. 1419, 1997.

[TC98] T. Sander and C. Tschudin, “Towards Mobile Cryptography”, Proceedings of the IEEE Symposium on Security and Privacy, 1998.

[WT04] W. Thompson, A. Yasinsac, T. McDonald, “Semantic Encryption Transformation Scheme”, International Workshop on Security in Parallel and Distributed Systems, 2004.