Semantic Encryption Transformation Scheme*

Willard Thompson
Florida State University
267 Love Building
Tallahassee, FL 32306, USA
wthompso@cs.fsu.edu

Alec Yasinsac
Florida State University
262 Love Building
Tallahassee, FL 32306, USA
yasinsac@cs.fsu.edu

Todd McDonald**
Florida State University
267 Love Building
Tallahassee, FL 32306, USA
mcdonald@cs.fsu.edu

Abstract

We present a scheme to protect mobile code from malicious hosts. We propose a transformation scheme that changes the semantics of a program using pseudo-random I/O scrambling, conditional elimination, and encryption using numeric variables for changing programs into encrypted but executable form that yields a recoverable result. The goal of our transformation process is to prevent an attacker from knowing the purpose of a program in order to reduce tampering.

Keywords*: semantically transform program, mobile code protection.*

1 INTRODUCTION

There are various areas of software applications that would benefit from encrypted computations such as mobile code, mobile agents, electronic voting, watermarking, and hiding intellectual property. It is difficult to protect remote computations, since they can be visually inspected, statically analyzed and dynamically tested by its executing environment [4, 5, 8].

The goal of this paper is to present a program encryption transformation scheme that will minimize the impact of Black Box and White Box analysis of a malicious host. Our proposed program transformation scheme will transform an original program into a semantically different but executable program such that the result of the transformed program is recoverable. Thus, our aim is to prevent an adversary from knowing the real intentions of the original program so that his ability to tamper with it is reduced to blind disruption, allowing greater program survivability.

The remainder of this paper is organized as follows: We identify some important related work in section 2. In section 3 we discuss the goals of our proposed Semantic Encryption Transformation Scheme (SETS). In section 4, we go over the transformation methodology, and finally, in section 5 we conclude.

*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.

2 RELATED WORK

We briefly discuss some of the important points of the two main thrusts of mobile code protection, mobile cryptography and code obfuscation. Most of the related work for protecting mobile code has been in defending against White Box analysis that is, protecting against identifying the actual instructions of a program. There has also been related work for defending against Black Box analysis, since given enough I/O pairs, the function may be deduced without analyzing the code.

2.1 Mobile Cryptography

Sander and Tschudin [13, 14] have coined the phrase Mobile Cryptography, which aims to provide provable security for mobile code. They also provide a summary on computing with encrypted functions (CEF) [13, 14]. CEF refers to a process where a program is transformed into a different program that protects the original intent, yet still produces the desired result. Currently, there are no known practical CEF schemes.

The homomorphic encryption scheme (HES) is another form of mobile cryptography. HES provides a mapping of data elements of one group or ring to another unequal but congruent group or ring of data elements. HES allows for computing with encrypted data and ensuring privacy of the input. HES is not reasonably practical because large numbers are required in order to thwart an attacker from being able to brute force a computation in any reasonable time and HES encryption mechanisms suffer from information leakage [13, 14].

Loureiro and Molva [11] present a function hiding technique based on error correcting codes (ECC). Loureiro and Molva [11] make use of the McEliece public key cryptosystem in conjunction with Goppa codes to encrypt a function. However, the practicality of using ECC is also limited.

Another area that is closely related to mobile cryptography is the integration of known data encryption mechanisms, such as AES (Advanced Encryption Standard) and DES (Data Encryption Standard), into a program [9, 10]. Chow, et. al. give a direction into providing White Box protection using AES [9] and thwarting extraction of secret keys from a program using

DES [10]. However, some of the drawbacks of their methods are that it is not provably secure and that it significantly degrades performance.

2.2 Code Obfuscation

Code obfuscation scrambles the syntax of a program into some chaotic form that is actually another representation of the same functionality [1, 2, 3, 7, 9]. The goal of obfuscation is to increase the cost for an adversary during reverse engineering. Unfortunately, obfuscation by itself does not provide a mathematical basis of security, making it difficult to measure its effectiveness [1, 2]. If only humans analyzed programs, then obfuscation may provide enough time complexity for security. However, the more damaging attacks are automated [8].

Most obfuscation techniques are applied to the decompilation phase of reverse engineering. Among the few who focus on the disassembly phase is Linn and Debray [3], who discuss thwarting static disassembly algorithms. Additional work on thwarting static analysis is that of Wang, et. al. [4], who use a compiler based technique for obstructing static analysis of programs using control and data flow transformations.

One of the seminal works for code obfuscation was that of Collberg, et. al. [2], who 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.

Ng and Cheung [12], use "intention spreading" to strategically insert dummy code into a program. Ng and Cheung [12] attempt to maximize entropy by transforming a program's original intention to a large number of indistinguishable intentions. This inundates the remote host with multiple, equiprobable intentions via noisy coding, thus reducing the adversary's ability to correctly guess the original intention of the code.

Hohl [6] presents a time-limited Black Box protection mechanism for mobile agents. The idea is to construct a Black Box agent with the same functionality of the original agent but with an obfuscated structure. Hohl's claim is that this obfuscated structure provides enough time complexity to prevent an adversary from learning the meaning of the code. Upon the expiration of the allotted time, the agent becomes invalid.

Finally, another related work is that of Aucsmith [5], which implements mechanisms for verifying the integrity of operations. The main code segment is the Integrity Verification Kernel, which provides unique installation and can be self-modifying and self-decrypting [5].

3 SETS

We now discuss the goals of the Semantic Encryption Transformation Scheme (SETS).

3.1 Objective

The goal of SETS is for Alice to make it difficult for a malicious Bob to comprehend the semantics of the program, by altering the operational semantics of the code and by hiding the I/O relationship of the program[1]. An original program p is transformed into a nonequivalent encrypted program p' as conceptually shown in Figure 1. Our notion of nonequivalence between p and p' is that when given polynomially many distinct inputs, where the same input is used for both p and p', it is computationally infeasible to find two outputs that are equal, as also conceptually shown in Figure 1. Thus, the objective of this paper is to address the following question:

> Can Alice transform an original program p into a secure program p' such that when Bob has possession of p', he is unable to efficiently identify the semantics of p, and the output of p' will enable Alice to efficiently recover the intended result?

3.2 Conceptual Model of SETS

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 the 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.

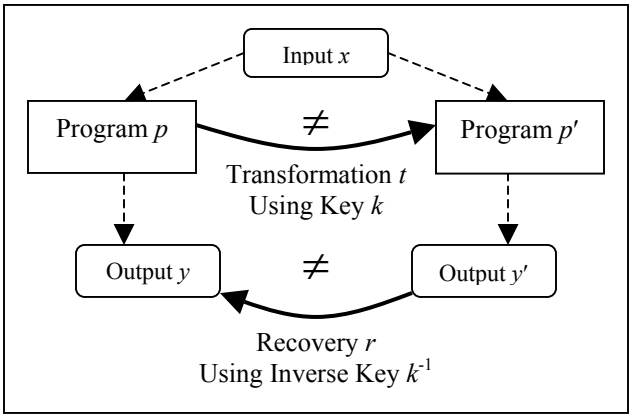

***Figure 1**, Semantic Encryption Transformation Scheme.*

[1] By convention we let Alice be the originator of the mobile code and let Bob be the remote host that executes the mobile code.

3.3 Knowing vs. Not Knowing a Function

The concept of knowing a function is fundamental to our notion of program encryption[2]. In order to measure the security effectiveness of protecting programs we must formally specify the conditions for program knowledge, with respect to the transformed program p'. We formally define knowing a function as:

Definition 1: For every distinct input each corresponding output can be predicted in polynomial time.

In contrast to knowing a function, we define NOT knowing a function as:

Definition 2: For every distinct input each corresponding output can be predicted with only greater than polynomial time.

Since these definitions represent the extreme cases, an adversary may only need to predict a percentage of outputs to any set of corresponding distinct inputs for p' to obtain enough clues about the non-encrypted, original program p. We argue that the adversary's effectiveness increases with more outputs that can be efficiently predicted, as conceptually shown in Figure 2.

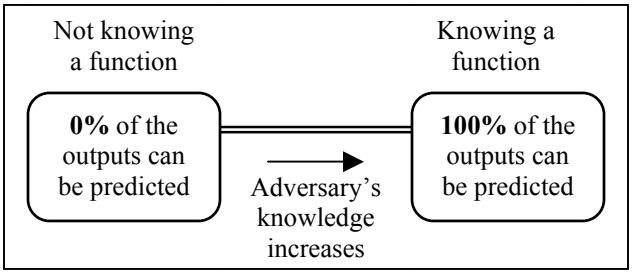

***Figure 2**, Knowing vs. Not Knowing a Function.*

We now give a Black Box, program, encryption, transformation means for which it is computationally infeasible to determine the function given a polynomial number of I/O pairs.

***Theorem 1**: If a program P: $X \rightarrow Y$ is concatenated with a cryptographically strong data encryption algorithm E to form another program P′: $X \rightarrow Y$, that is $P' = P \mid E$, such that the data result of P is encrypted with E, then it is computationally infeasible to determine the I/O relationship of P′.*[3]

> **Proof**: The properties of strong data encryption are multiple rounds of substitutions and permutations in conjunction with a large key-space, randomness, uniform distribution, independence, and the inability to determine the next bit after a given sequence of bits with a probability greater than 0.5 in polynomial time. As a result, it is computationally infeasible to deduce the plaintext of a corresponding ciphertext, or key in polynomial time.
>
> Any outputs produced by the execution of p' are computationally indistinguishable, that is they are pseudo-randomly generated. After knowing any n I/O pairs, $\{(x_1, p'(x_1)), \ldots, (x_{n-2}, p'(x_{n-2})), (x_{n-1}, p'(x_{n-1})), (x_n, p'(x_n))\}$, when given the next input x_{n+1}, the corresponding output $p'(x_{n+1})$ cannot be correctly predicted in polynomial time, other than by a random guess or by executing the program. Moreover, by executing the program a polynomial number of times, the entropy of the program remains constant, that is the probability of deducing a pattern from any number of I/O pairs remains negligible.
>
> Therefore, when a program $p \in P$ is concatenated with a data encryption algorithm $e \in E$ to form a new program $p' \in P'$, the I/O relationship of p' is computationally infeasible to determine. □

We can now formally describe our notion of an adversary being able to deduce the semantics of the original program. Given an algorithm A, consider an infinite set of programs $P = \{p_1, p_2, \ldots, p_\infty\}$, comprised of all programs that implement algorithm A, such that each p is syntactically distinct, i.e. no p is a copy of another, and the corresponding infinite set of encrypted programs $P' = \{p'_1, p'_2, \ldots, p'_\infty\}$, such that $p'_i = t(p_i, k)$, for any $i \geq 1$. An adversary's knowledge of A increases if and only if, given p'_i, his ability to predict $y = p_i(x)$ in polynomial time for each distinct x, increases.

3.4 Black Box and White Box Security Levels

We describe the security strength of program encryption in four levels, from weak (1) to strong (4). The intuition of these levels is that White Box security is stronger than Black Box security, and protection against predicting encrypted results (y') is stronger than protection only against predicting decrypted results (y). Note that each definition below begins with the following common part: 'Given an encrypted program p'_i, n known I/O (x_i, y'_i) pairs of p'_i, and any number of subsequent distinct inputs, $\{x_k, x_{k+1}, x_{k+2}, \ldots\}$ where $k > n$, …'[4]

[2] Since a program implements an algorithm, we use the popular notion that an algorithm is a relationship or mapping between a set of inputs and a set of outputs.

[3] Note that Theorem 1 is exclusively in terms of Black Box security, that is, independent from White Box security.

[4] These definitions are independent from executing the program or via a random guess.

3.4.1 Weak Black Box (1)

Definition 3: …an adversary cannot deduce *any* corresponding decrypted output y_k in polynomial time.

3.4.2 Weak White Box (2)

Definition 4: …while having access to the code of p'_i and understanding the logic of the p'_i, an adversary cannot deduce *any* corresponding decrypted output y_k in polynomial time.

3.4.3 Strong Black Box (3)

Definition 5: …in addition to Weak Black Box, an adversary cannot deduce *any* corresponding encrypted output y'_k in polynomial time.

3.4.4 Strong White Box (4)

Definition 6: …in addition to Weak White Box, an adversary cannot deduce *any* corresponding encrypted output y'_k in polynomial time.

Notice that each definition describes how much information can be gleaned from the transformed program p'. The goal is to achieve Weak Black Box security at a minimum, that is, to ensure that an adversary cannot predict the output of a program p by examining its encrypted version p'. The other three protection levels give us greater confidence in the transformation security.

3.5 Formal Definition of SETS

We now define SETS, as conceptually shown in Figure 1 above. SETS is a 9-tuple (X, P, P′, T, R, K, K^{-1}, Y, Y′) such that:

1. X is the set of possible inputs to both programs $p_i \in P$ and $p_i' \in P'$.
2. P is the set of original, non-encrypted programs.
3. P′ is the set of transformed programs derived from P.
4. T_K: P $\rightarrow$ P′, is the set of program transformation processes.
5. R_{K-1}: Y′ $\rightarrow$ Y, is the set of output recovery processes.
6. K is the set of computation keys for transformation T.
7. K^{-1} is the set of inverse computation keys for the recovery R.
8. Y is the set of final results from R.
9. Y′ is the set of intermediate results from P′.

We note that the computation keys, k and k^{-1}, are secret information only known to Alice. To illustrate this notion, suppose an original program is y = x + 3 and the transformed program is y′ = 2x – 5, then, in this case, k would be the addition of (x – 8) to y, and k^{-1} would be the recovery computation of [((y′ + 5) / 2) + 3], yielding the originally intended result. For instance, if x = 7, the original result would be 10 and the intermediate result would be 2(7) – 5 = 9. Thus, Alice would compute ((9 + 5) / 2) + 3 to recover the original result, 10.

3.6 Minimum Program Encryption Characteristics

To summarize our goals and to ensure that our transformations are usable, we require three properties for SETS:

1. p' cannot equal p, semantically nor syntactically.
2. p' yields a recoverable result of p by Alice.
3. Given p', Bob is unable to know p.

4 TRANSFORMATION METHODOLOGY

In section 2 we described other attempts to encrypt programs. In this section, we present our proposed transformation methodology of SETS. We aim to construct White Box security while building on our basis for Black Box security.

4.1 Black Box Transformation

Considering Theorem 1, producing a pseudo-random result ensures that the adversary is unable to feasibly deduce any I/O correlations through exclusive I/O analysis. The result of the encrypted program undergoes a final data encryption transformation on any of the outputs that are returned to Alice, as conceptually shown in Figure 3.

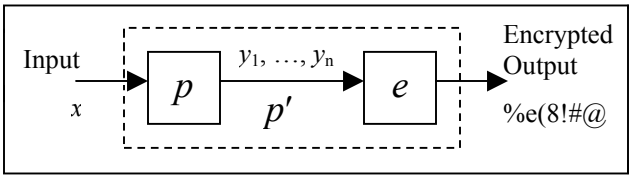

***Figure 3**, Producing a Pseudo-random Result.*

4.2 White Box Transformation

We ultimately aim to disguise the operations of the original program. When an encrypted program receives an input the resulting output is in expanded form. In other words, regardless of the I/O mapping of the original, non-encrypted program, that output is expanded, and we have a one-to-many I/O relationship as conceptually shown in Figure 4.

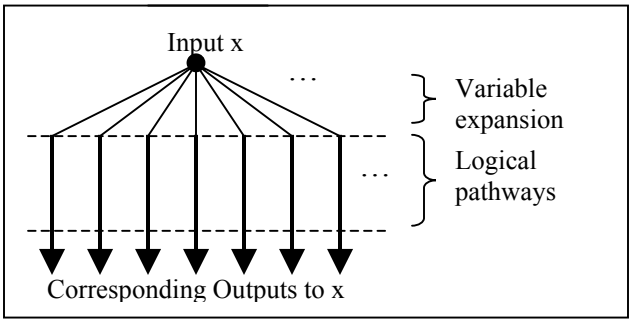

***Figure 4**, Multiple Pathway Executions.*

Moreover, since in an original program the logical pathway that is to be executed is dependent upon the input, we attempt to minimize this dependency by

implicitly requiring multiple logical pathways to be executed in order to fulfill the requirement of multiple results, for expanded outputs. This is also conveyed above in Figure 4.

4.2.1 Interleaving Data Encryption

We acknowledge that an adversary could quickly recognize the data encryption mechanism *e* in section 4.1 through White Box code analysis. We utilize the data encryption property of transpositions by pseudo-randomly permuting the expanded structures after each operation on those structures, as conceptually described in Figure 5.

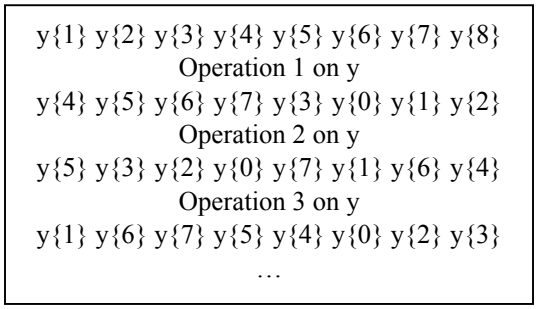

***Figure 5**, Transposition of Expanded Variables.*

Notice, in Figure 5, before each (set of) operation(s), the elements of y are rearranged. Since only Alice knows which element(s) is a part of her transformation/recovery key, she knows which element(s) to decrypt.

4.2.2 Conditional Elimination

Conditional elimination is a White Box transformation technique that we propose. Conditional elimination has the opposite effect of opaque predicates [2], by reducing the number of pathways through a program by eliminating conditions in the original program. Conditional elimination changes the program from single pathway executions to multiple simultaneous pathway executions. Conditional elimination is viable if adequate information within the Boolean expression, making up the condition, exists in the subsequent non-conditional operation(s) and the non-conditional operations do not cause a conflict during execution. Lets look at a toy example to clarify this concept:

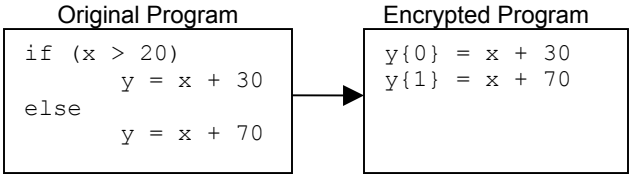

***Figure 6**, Conditional Elimination Toy Example.*

Figure 6 consists of pseudo-code for an original program in the left box that gets transformed into an encrypted program in the right box. Given that x and y are integers, the two non-conditional statements y = x + 30 and y = x + 70 from the original program transform into y{0} = x + 30 and y{1} = x + 70 respectfully, shown in the encrypted program, and the if-else condition is removed. Notice how we expanded y into array y{}. As long as the non-conditional statements contain some information about the condition such as the variable x, then Alice would know which statement is the correct one. For instance, if x = 21, we know, from the original program, that the first non-conditional statement would be executed resulting in y = 51. This would also be the value of y{0} in the encrypted program, which would indicate to Alice to use y{0} instead of y{1} as shown below for the recovery procedure in Figure 7. Also, if y{0} = 50 (or less), then Alice would know to use y{1}.

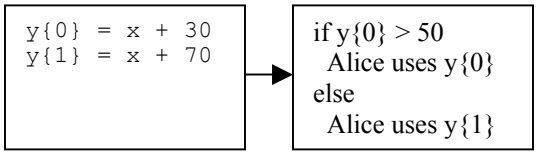

***Figure 7**, Recovery for Conditional Elimination.*

4.2.3 Encryption Using Numeric Variables

Encryption using numeric variables is another White Box transformation technique that we propose. It consists of changing mathematical operations using numeric variables or constants. This serves two purposes: disguising the real computation, and allowing for an easily reversible computation for Alice during decryption. As it stands, the encrypted toy program in Figure 6 would expose x + 30 and x + 70 to the adversary. Alice can disguise those two computations by semantically changing them. Additionally, Alice further expands the array y{} to disguise the number of logical pathways of the original program. Our toy program, after performing encryption using numeric variables, is shown in Figure 8:

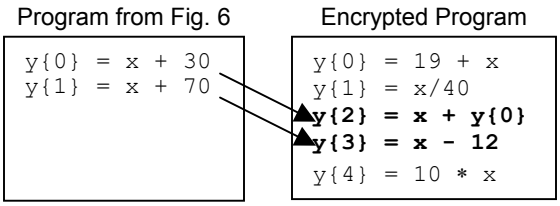

***Figure 8**, Encryption Using Numeric Variables[5].*

For the recovery procedure, conceptually shown in Figure 9 below, Alice can randomly choose any number of elements of the array as part of her recovery key that she keeps secret. For instance, Alice randomly designates y{2} and y{3} as part of the computations that will eventually yield a result that Alice decrypts. We can statically trace the result of y{2} as 2x + 19 from the right box above in Figure 8. If this expression evaluates to an

[5] We can assign numeric constants to variables, but in light of brevity we simply use the actual numeric values themselves instead of variables that represent those constants.

encrypted output of 61 or greater, Alice knows to subtract 19, and then divide by 2. After that Alice can then simply add 30 to get the final intended result of 51. If the result of y{2} is less than 61, Alice can use the expression of y{3} to decrypt the result. Since the difference of 70 and –12, from the second line in the left box in Figure 8, respectfully is 82, Alice would add 82 to y{3} to get the final intended result. Finally, the other elements of the array y{}, namely y{0}, y{1}, and y{4} are discarded.

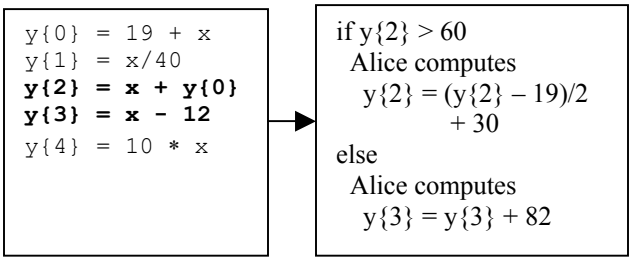

***Figure 9**, Recovery for Numeric Variables Encryption.*

Finally, considering the interleaving of the data encryption algorithm in section 4.2.1 above, after each (non-conditional) operation, we can permute the array y{}, and perform conditional elimination and encryption via numeric variables repeatedly.

5 CONCLUSION

We provide a means for a key-based transformation that semantically changes a program while retaining the ability to efficiently retrieve the computed result. We also give a framework for defining program encryption and give toy examples of how our techniques can be applied. With this framework in place, we can consider more comprehensive obfuscation techniques.

SETS provides a hybrid approach between Mobile Cryptography and obfuscation by allowing for the decryption of the intermediate result and scrambling code. Our approach builds upon our notion of Black Box security with the goal of achieving White Box security. Our method provides insight into defending against reverse engineering, deobfuscation and decompilation via the additional step an adversary would need to take to deduce the original program from the semantically transformed program.

Finally, as our research evolves, the true test of our techniques will be determined empirically, and may add more generality to our solution.

REFERENCES

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

[2] 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.

[3] C. Linn, S, Debray, “Obfuscation of Executable Code to Improve Resistance to Static Disassembly”, Proceedings of the 10th ACM Conference on Computer and Communication Security, pp. 290 - 299, 2003.

[4] C. Wang, et. al., “Protection of Software-Based Survivability Mechanisms”, Foundations of Intrusion Tolerant Systems (OASIS'03), p. 273, 2003.

[5] D. Aucsmith, “Tamper Resistant Software: An Implemenation”, Information Hiding: First International Workshop, Cambridge, U.K., pp. 317-333, 2001.

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

[7] G. Wroblewski, "General Method of Program Code Obfuscation", PhD Dissertation, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[8] J. Vitek, G. Castagna, “Mobile Computations and Hostile Hosts”, Journ'ees Francophones des Langages Applicatifs JFLA99, pp. 113-132, 1999.

[9] S. Chow, et. al., “White-Box Cryptography and an AES Implementation”, LCNS, 9th Annual Workshop on Selected Areas in Cryptography, pp. 250-270, 2002.

[10] S. Chow, et. al., “A White-Box DES Implementation for DRM Applications”, ACM CCS-9 Workshop DRM 2002 – 2nd ACM Workshop on Digital Rights Management, Springer-Verlag LNCS, pp.1-15, 2002.

[11] S. Loureiro and R. Molva, “Function Hiding Based on Error Correcting Codes”, Proceedings of the International Workshop on Cryptographic Techniques and Electronic Commerce, 1999.

[12] S. Ng, K. Cheung, “Protecting Mobile Agents Against Malicious Hosts by Intention Spreading”, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99), pp. 725-729, 1999.

[13] T. Sander, C. Tschudin, “On Software Protection Via Function Hiding”, Lecture Notes in Computer Science, Volume 1525, pp. 111-123, 1998.

[14] T. Sander, C. Tschudin, “Protecting Mobile Agents Against Malicious Hosts”, Lecture Notes in Computer Science, Volume 1419, 1997.