Cryptomorphic Programming: Towards White-box Security*

W. Thompson and J. Todd McDonald**
{wthompso, mcdonald}@cs.fsu.edu
Department of Computer Science
Florida State University
Tallahassee, FL 32306

Abstract *We provide a theoretical notion of white-box security for protecting the integrity and privacy of mobile programs. We assume that protecting the program's semantic privacy will improve the integrity protection of a program, thereby improving tamper resistance and the protection of secret algorithms. We introduce this notion using a Random Oracle Model and describe our important concept of a random program via a transformation to further clarify our notion of white-box security. It is this theoretical notion that is required to achieve provable security for program protection, with respect to tamper reduction.*

Key Words *Program Integrity Protection, Tamper Resistance, White-box Security*

1 Introduction

There are a number of reasons why the integrity and privacy of programs need to be protected. For mobile code the integrity of the program needs to be protected to ensure functionality and survivability within a distributed environment. Mobile agents would certainly benefit from some kind of mobile code integrity protection. Likewise, the privacy of a program is important for proprietary reasons. If the privacy of a program can be protected then secret algorithms and intellectual property can be hidden to ensure digital rights management.

We focus our work on software-only solutions for addressing the malicious host problem. The malicious host problem is a remote execution host that is not trusted, having total control and visibility of a visiting program, which can tamper with the program. For instance, 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. If Bob behaves maliciously he could in the worst case alter the program for his own benefit. We call this worst-case scenario *effective tampering*.

Our ultimate goal is to reduce effective tampering of the program to blind tampering. The benefit of blind tampering over effective tampering is that the damage potentiality is

* 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.

reduced. For instance, if a mobile agent has been blindly tampered with by a remote host A it may crash its execution on a subsequent host B. In this case, host B can easily recover with only a loss of some time, where as if host A effectively tampered with the mobile agent, host B could execute erroneous code and commit to something it does not want to commit to or not be able to commit to something it wants to commit to.

Our specific focus within this paper is to provide a theoretical notion for white-box security to complement our notion of black-box security in [14]. Currently, the notion of white-box security is vague and there has been little work only within the past few years for white-box security[1] of programs. In fact, there has been little work in theoretical foundations for mobile program protection overall.

1.1 Related Work

We categorize related work into the following general areas, code obfuscation, mobile cryptography, and white-box security.

1.1.1 Code Obfuscation

Code obfuscation is the altering of the syntax of a program into a less readable format, while maintaining the black-box semantics of the program [1, 2, 3, 4, 9]. A good example of code obfuscation is something a software developer uses often, that is a compiler. The compiler translates high-level source code into machine code so that the program can easily run on a computer. In fact, there are a myriad of obfuscation techniques that are similar to compiler optimization techniques such as variable splitting, interleaving methods and reducing flow-graphs to name a few [2]. Opaque predicates have been the main contribution of [2].

The machine code generated by a compiler is usually a binary string of which is tedious and time consuming to read and understand, in general, to say the least. However, knowing the architecture in which the program runs on and knowing the program's I/O relationship, the program can be successfully reversed engineered into more readable code such as source code. Moreover, code obfuscation is heuristic-based, and upon learning the techniques used to obfuscate a program, an adversary can easily reverse the operations and construct a de-obfuscator. Ultimately, although code obfuscation is practical it is nowhere near being provably secure [1]. However, there has been some work to counter the claims of [1], namely that of N. Varnovsky and V. Zakharov [9] who attempt to provide a formal information-theoretic definition of code obfuscation.

Currently, the most advanced work in code obfuscation research is to address static analysis [9]. M. Mambo, et. al., [8] compare properties or conditions of a program that make it easy and hard to analyze a program. It is argued that static analysis of a program is easier for an adversary than doing dynamic analysis, since the text of a program can be easily entered into a static analyzer resulting in viewable properties of the program [3]. Moreover, code obfuscation research is meandering towards NP-hard reducibility.

[1] Our notion of white-box security includes provable security as opposed to just code obfuscation.

Ideally, one would find an NP-hard problem that can be reduced to a certain class of program structures [9], for instance, some analysis techniques such as analyzing function pointers is shown to be NP-hard [13].

1.1.2 Mobile Cryptography

Mobile cryptography attempts to protect the privacy of functions by using mathematical constructs, in hopes of providing a general means for provably securing programs [7, 12]. Since multi-party computations and secret sharing provide a basis, mobile cryptography has been around longer than code obfuscation. The most well known mobile cryptographic scheme is Computing with Encrypted Functions (CEF) using homomorphism. 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. 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 is another work by K. Cartrysse and J.C.A. van der Lubbe [6] that discuss perfect secrecy for mobile code, that can arguably fall under the mobile cryptography category. Claude Shannon's notion of a secrecy system for static data is used for mobile programs. K. Cartrysse and J.C.A. van der Lubbe describe their perfect secrecy notion using three main components: key space, function space and the output space, with the caveat that the key space has to be at least as large as the function and output space. Their perfect secrecy notion is theoretical and they express their concept with an example using one-time pad for polynomials. Lastly, their notion falls short of the full transformation process, in that they have no procedure for decrypting the encrypted output.

1.1.3 White-box Security

There is some overlap between code obfuscation and white-box security. We provide a separation solely based on the notion that code obfuscation is currently not provably secure, whereas our aim with white-box security is to achieve a more robust means of formally measuring security. As mentioned above, white-box security is a relatively new area of program protection. Chow, et. al., [10, 11] were the first to attempt to prevent extraction of secret keys from an encrypted symmetric cipher algorithm such as DES and AES. For DES they interleave affine transformations with s-boxes, which attempts to delinearize permutation operations and xors that make up an affine transformation. For AES they encode key-independent tables with random bijections.

They argue that their methods will prevent an adversary from easily obtaining the secret key being used. Unfortunately, there are some attacks that can exploit the nonlinear properties of s-boxes, such as the statistical bucketing attack and the differential fault injection attack. Link and Neumann [5] follow up on Chow, et. al.'s work by improving resistance against the statistical bucketing attack by increasing the size of the matrices making up DES, and intermixing the s-box input with the left and right input halves for each round.

1.1.4 Black-box, White-box and Gray-box Security

The following definitions for black-box, white-box and gray-box security have been informally used within the papers throughout the related work. We not only use them but also simply reiterate them for clarification:

> *Black-box security* is protecting the privacy and integrity of a program by not allowing any visibility into the instructions of a program. Since the goal of black-box cryptanalysis is to gain an understanding of the program from the program's I/O, black-box security serves to obscure the I/O relationship of the program as well.
>
> *White-box security* protects the integrity of a program by not allowing the adversary to understand the program when given the internal instructions of that program[2].
>
> *Gray-box security* is the combination of black-box and white-box security, that is, providing protection against static and dynamic analysis when used together.

1.2 Objective

Our objective within this paper is to provide a basis of white-box security that will ensure provable security with respect to the reduction of effective tampering to blind tampering. We aim to describe this notion by using a Random Oracle Model. We partially justify and motivate our Random Oracle Model of white-box security via the transformation of a primitive program using the random properties of a data cipher as well.

We note that most of the related work regarding program encryption address protecting against automated static analysis. In our work, however, we want to also address protecting against non-automated static techniques such as manually inspecting code. This situation can arise in practice. For instance, mobile code especially in the form of mobile agents, are not very large for efficiency reasons. An adversary, after successfully reverse engineering a mobile agent, can visually inspect any part of the code he feels is important. In addition to each of these attacks individually, in most cases the adversary uses a combination of static analysis and dynamic analysis. Therefore, we aim to generalize our protection scheme by addressing as many vulnerabilities as possible.

The remainder of this paper is organized in the following way: We discuss our model, which we call the Cryptomorphic Programming Model (CPM) in section 2. In section 3 we go over our main notion of White-box security along with a transformation. In section 4 and section 5 we discuss future work and conclude respectively.

[2] Since the instructions can be viewed, the privacy of the program can be debated.

2 Cryptomorphic Programming Model

We build upon our *Cryptoprogramming* notion [15] and *Semantic Encryption Transformation Scheme* (SETS) model [14] by formally evolving our concept of program encryption transformation. We show an algebraic equivalence between two cryptomorphic pathways, which go from ***x***, the program's input, to ***y***, the program's truly intended output as shown in Figure 1.

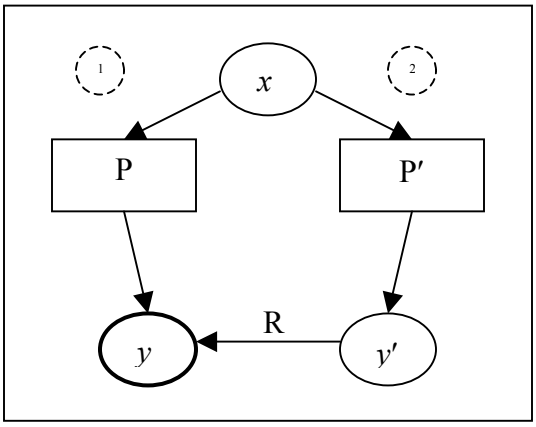

Figure 1*, Cryptomorphic Programming Model (CPM).*

The CPM in Figure 1 consists of two sets of pathways leading from the initial input ***x*** to the final output ***y***. The first pathway is on the left side of Figure 1, which is a program **P** that takes an input ***x*** and outputs ***y***. The second pathway, which is predominately on the right side of Figure 1, consists of taking the same ***x*** as input into a transformed program **P′** producing output ***y′***, which then is decrypted into ***y*** via recovery procedure **R**.

2.1 Pathway Equivalence

We now formally describe our two pathways in a cryptomorphic way. For the sake of clarity we note that since **P′** is some transformation of **P** (rather an encrypted version of **P**), there exists an arrow **T**, representing the transformation from **P** to **P′**. We omit this arrow from the diagram in Figure 1 on purpose to emphasize the individual pathways from ***x*** to ***y***. We show that by going route 2 above in Figure 1, we cryptomorphically achieve the equivalence to route 1, however, with the help of **T**.

So, by letting $\boldsymbol{x}\mathbf{P} = \boldsymbol{y}$, $\boldsymbol{x}\mathbf{P'} = \boldsymbol{y'}$, $\mathbf{TP} = \mathbf{P'}$, and $\boldsymbol{y'}\mathbf{R} = \boldsymbol{y}$, we have in Listing 1:

$$\boldsymbol{x}\mathbf{P'} = \boldsymbol{y'}$$
$$\boldsymbol{x}\mathbf{PT} = \boldsymbol{y'}$$
$$\boldsymbol{x}\mathbf{PTR} = \boldsymbol{y'}\mathbf{R}$$
$$\boldsymbol{x}\mathbf{P} = \boldsymbol{y}$$

Listing 1*, Algebraic Representation of Cryptomorphic Programmatic Equivalence.*

2.2 Cryptosystem Analogy

In addition to our cryptomorphic pathway equivalence, we show the equivalence of CPM to a data cryptosystem. As with any data cryptosystem there exists an easy route, namely that of knowing or having the secret (decryption) key, and there exists a difficult route for decrypting plaintext into ciphertext. Below in Figures 2 and 3, we describe both of these routes.

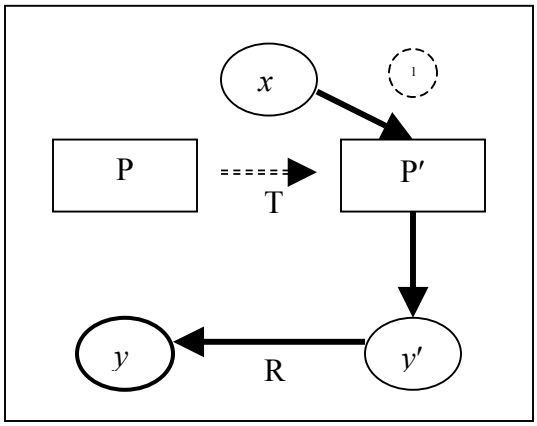

***Figure 2**, Route 1 (Easy).*

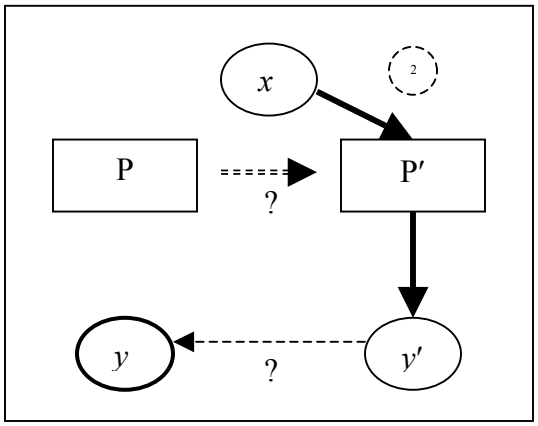

***Figure 3**, Route 2 (Hard).*

We can surmise that **T** is a cryptomorphic transformer of **P** if there exists a mapping **R** such that **TR** is what we call a cryptomorphic identity operation for ***x*P**. Although this is not a pure identity mapping it is a special kind of mapping that uses the notion of ***x*P** and ***x*P′** as conceptually shown below in Figure 4.

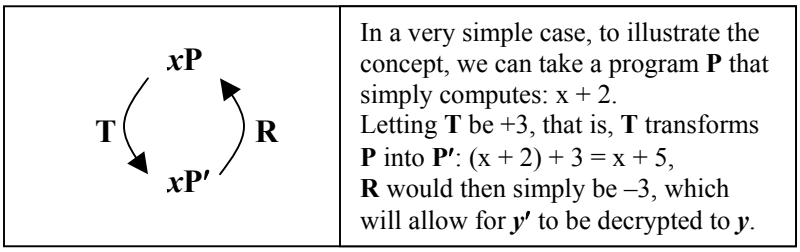

***Figure 4**, TR, Cryptomorphic Identity Concept.*

3 White-box Security

In addition to our black-box security model in [14], we now attempt to add another dimension, white-box security, to our evolving CPM. The following are our assumptions: The adversary can observe the operation of the encrypted program **P′**. Since, our aim is to protect the original non-encrypted program **P**, when given the encrypted program **P′**. Thus, when the adversary has **P′**, he should not be able to deduce any part of **P** that would gain him greater benefit.

3.1 Random Oracle Model

The Random Oracle Model for encrypting programs describes the highest-level of white-box security that we seek for protecting the integrity and privacy of a program. We have an oracle $\mathcal{O}$ that encrypts programs, i.e. performs the function of **T**. The adversary sends an original non-encrypted program **P** to the oracle $\mathcal{O}$ to be encrypted and the oracle returns the corresponding encrypted program **P′**, as shown below in Figure 4 via the top two arrows labeled 1 and 2. Upon a polynomial many encryptions, the oracle then sends to the adversary an encrypted program $\hat{\mathbf{P}}'$, along with its non-encrypted counterpart **P** as shown by label 3. The adversary would then have to determine whether or not the encrypted program $\hat{\mathbf{P}}'$ is the real encrypted program **P′** corresponding to **P** or if it is a random program $\mathbf{P_R}$ as shown by his decision in label 4.

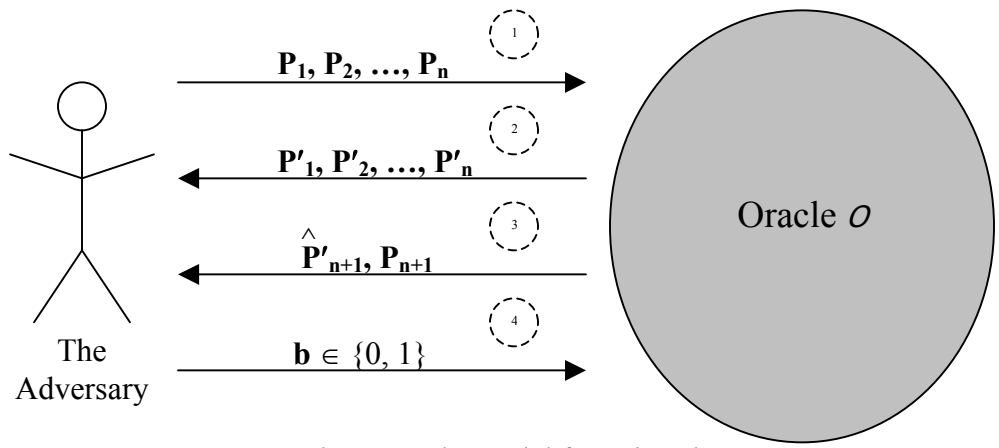

***Figure 4**, Random Oracle Model for White-box Security.*

The adversary attempts to make his prediction by returning a bit $\mathbf{b} \in \{0, 1\}$ corresponding to his guess of either $\mathbf{P_R}$ or $\mathbf{P'_{n+1}}$., as shown in label 4 in Figure 4 and below in Listing 2. The probability that the adversary is able to predict either the real encrypted program from a random encrypted program should be less than or equal to ½ + ε, as shown in Listing 3 to ensure provable security. We let ε be the negligible error probability. We are now left with the notion of $\mathbf{P_R}$. Thus we achieve a computational indistinguishability.

$$\hat{P}'_{n+1} = \begin{cases} P_R \\ P'_{n+1} \end{cases}$$

***Listing 2**.*

$$\Pr[\hat{P}'_{n+1} = P_R] \leq \tfrac{1}{2} + \varepsilon$$

$$\Pr[\hat{P}'_{n+1} = P'_{n+1}] \leq \tfrac{1}{2} + \varepsilon$$

Listing 3.

3.2 Definition of $\mathbf{P_R}$

Since we know what **P** and **P′** are, we only need to define $\mathbf{P_R}$. Thus, the following is the definition of $\mathbf{P_R}$:

> ***Defintion 1: Random Program*** $\mathbf{P}_R$
>
> *A random program* $\mathbf{P}_R$ *is a hypothetical program that performs a meaningless operation. In other words,* $\mathbf{P}_R$ *is an executable program where its operations and data are not associated with any context.*[3]

3.3 White-box Transformation

We take the definition of $\mathbf{P_R}$ one step further by showing a transformation of a primitive program. This transformation serves to illustrate our notion of $\mathbf{P_R}$. We start with a program **P** that performs some set of operations. For our purpose, **P** is abstract. The main assumptions that we use for the program transformation are the following:

- We do not consider loops per se. We instead reduce our notion of a loop to a series of conditional statements, since a loop is really made up of one or more contiguous conditional statements. We then only need to deal with a unidirectional instruction flow.

- Another assumption that we make is not considering the recoverability procedure **R** of the final result as shown above in Figure 1. Since recoverability is clearly defined by our black-box model in [14], we exclude this property of cryptoprogramming because we are focusing only on white-box security, namely that of describing $\mathbf{P_R}$.

- Given that our computer or instruction set is abstract or logical, that is, not tied to any particular architecture, we also do not consider hardware or low-level issues such as memory faults.

Moving on to our primitive program transformation. Suppose our hypothetical computer only has the following three types of instructions, where each instruction type consists of a mnemonic along with its numeric opcode.

- ADD:1 [Adds two numbers]
- MOVE: 2 [Moves contents from one cell[4] to another]
- BIZ: 3 [“Branch if zero”, Check to see if a cell is zero]

---

[3] We acknowledge that our definition of P_R is informal. However, we build upon this axiomatic notion in the following sections. Although, in trying to be less ambiguous, envision bits in a computer’s memory space changing and moving from location to location in a pseudo-random way.

[4] A cell represents a memory location. For our purposes it is an abstraction for a memory block and register within a real computer.

We use the following instruction format:

Opcode	*Destination Cell*	*Source Cell*

The opcode can either be 1, 2, or 3 corresponding to our three instructions above. We restrict the number of cells to eight, C[0], C[1], …, C[7], where cell C[0] = 0, is a constant. So, given the following program **P** in Figure 5, we randomly alter its operations and cells to produce the encrypted program **P′** in Figure 6:

```
00: // Assign (random) values
01: // into the last seven C's

10: 1 2 1 // ADD C[1] + C[2] → C[2]
11: 1 3 1 // ADD C[1] + C[3] → C[3]
12: 1 3 2 // ADD C[2] + C[3] → C[3]
13: 2 4 3 // MOVE C[3] to C[4]
14: 1 3 1 // ADD C[1] + C[3] → C[3]
15: 3 3 0 // BIZ (C[3] == 0)
16: 1 5 4 // ADD C[4] + C[5] → C[5]
17: 1 2 3 // ADD C[3] + C[2] → C[2]
18: 2 6 4 // MOVE C[4] to C[6]

19: 0 0 0 // HALT
```

***Figure 5**, Original program P.*

```
00:
01:

10: 1 5 2
11: 3 3 0
12: 1 1 4
13: 2 2 7
14: 2 5 2
15: 1 1 4
16: 2 5 4
17: 1 7 0
18: 1 2 4


19: 0 0 0
```

***Figure 6**, Encrypted program P′.*

The program **P′** in Figure 6 clearly executes. Henceforth, **P′** achieves white-box randomization illustrating our notion of $\mathbf{P_R}$.

4 Future Work

We acknowledge that our formulations of white-box security need more research in order to reach satisfying levels of practicality. For instance, our main concern regarding our current notion is being able to encrypt or hide a loop within a program. In briefly examining the following simple and general loop structure written in a programming language such as C++:

```
for (int i = 0; i < n; i++)
{
      // Do something
}
```

***Listing 4**, Simple Loop.*

We can see in Listing 4, that this loop has an iterator `i`, which allows the loop to iterate `n` times. Our current scope relegates us to heuristic methods for hiding the true intentions of a loop. For instance, we can alter `i` by changing the way the loop increments. We can also change `n` and increase or decrease the number of loop iterations. Other heuristic methods consist of obfuscation-like techniques such as inserting dummy code either by

convoluting the loop body or by adding more loops. However, all of these heuristic methods may increase the complexity or difficulty in decrypting the intermediate result $\mathbf{y'}$ via recovery process **R**. Ultimately, the overall goal of systematizing CPM for encrypting programs in general still needs more work.

Lastly, another issue that we would like to address in future work is the convergence of black and white-box security, that is, ultimately a gray-box solution. Current research trends address thwarting static-analysis while negating dynamic analysis due to the difficulty of deducing a program only by unique I/O pairs. However, as mentioned above, static and dynamic analyses can be combined as well to effectively understand a program. Given our current notion of black-box security [14] along with a robust notion of white-box security, we can begin taking steps towards practical solutions to encrypt programs for improving tamper resistance.

5 Conclusion

We have provided a notion for white-box security via the Random Oracle Model for encrypting programs. In the Random Oracle Model for white-box program security it is imperative to define our random program $\mathbf{P_R}$. We defined $\mathbf{P_R}$ as a meaningless program in that it consists of a random set of operations and data, which has no context association. We acknowledge that $\mathbf{P_R}$ is informally defined but we expressed our notion of $\mathbf{P_R}$ via a transformation of a primitive program, with the goal of showing the highest theoretical level of program integrity security.

6 Acknowledgement

We would like to acknowledge Dr. Alec Yasinsac and Dr. Mike Burmester for their contributions and insightful discussions regarding program protection.

Bibliography

[1] B. Barak, et. al., *On the (Im)possibility of Obfuscating Programs*, Electronic Colloquium 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 and S. Debray, *Obfuscation of Executable Code to Improve Resistance to Static Disassembly*, Proceedings of the 10th ACM conference on Computer and communications security, pp. 290-299, 2003.

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

[5] H. Link and W. Neumann, *Clarifying Obfuscation: Improving the Security of White-Box Encoding*, Cryptology ePrint Archive: Report 2004/025, 2004

[6] K. Cartrysse and J.C.A. van der Lubbe, *Secrecy in Mobile Code*, 25th Symposium on Information Theory in the Benelux, pp. 161-168, 2004.

[7] 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.

[8] M. Mambo et. al., *A Tentative Approach to Constructing Tamper-Resistant Software*, Proceedings of the 1997 Workshop on New Security Paradigms, pp. 23-33, 1997.

[9] N. Varnovsky and V. Zakharov, *On the Possibility of Provably Secure Obfuscating Programs*, LNCS, Vol. 2890, p. 91, 2003.

[10] S. Chow, P. Eisen, H. Johnson, P.C. van Oorschot, *White-Box Cryptography and an AES Implementation*, 9th Annual Workshop on Selected Areas in Cryptography, pp.250-270, Springer LNCS 2595, 2002.
[11] S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot, *A White-Box DES Implementation for DRM Applications*, Proceedings of ACM CCS-9 Workshop DRM 2002, Vol. 2595 of LNCS, pp. 1 – 15, Springer-Verlag, 2003.
[12] T. Sander and C. Tschudin, *Towards Mobile Cryptography*, Proceedings of the IEEE Symposium on Security and Privacy, 1998.
[13] T. Ogiso, et. al., *Software Tamper Resistance Based on the Difficulty of Interprocedural Analysis*, 3rd International Workshop on Information Security Applications, pp. 437-452, 2002.
[14] W. Thompson, et. al., *Semantic Encryption Transformation Scheme*, International Workshop on Security in Parallel and Distributed Systems, 2004.
[15] W. Thompson, et. al., *Cryptoprogramming: A Software Tamper Resistant Mechanism Using Runtime Pathway Mappings*, Florida State University, C.S. Dept., Technical Report #TR-041021, 2004.