OF UNICORNS AND RANDOM PROGRAMS

ABSTRACT
We provide a theoretical and practical notion of white-box security for protecting integrity and privacy of software. This notion provides a useful framework to analyze and implement software encryption mechanisms. We relate strength of program encryption to properties of *random programs* and take a purposefully different view of security than the traditionally cited virtual black-box method of Barek *et al.* [1]. We pose and answer several questions of interest: what are random programs, do they exist, and how can they be used to evaluate effectiveness of proposed algorithms. Further, a theoretical foundation for program security based on the random oracle model is defined using our definition of random programs.

KEY WORDS
Software agents, program encryption, mobile agent security, tamper proof software

1. Introduction

Securing the intellectual rights of software (in general) and protecting the integrity and privacy of mobile programs (in particular) are critical for success of distributed computing environments. The malicious host problem in mobile agent settings provides an interesting case for examining such requirements. In particular, a remote host has full and complete control over any code that it executes—leaving open the possibility of undetected program alteration. Methods for preserving code privacy in such environments have included multi-party secure computation, encrypted functions, tamper-proof hardware, and obfuscation.

Obfuscation is the attempt to hide or blur the semantic knowledge of a program via heuristic means, increasing tamper resistance or protecting proprietary software rights—without the need for expensive trusted hardware. Even though many obfuscation and tamper-proofing techniques have been devised and catalogued [2,3,4,5], measuring the effectiveness and strength of such techniques has been an elusive task for the research community.

Early researchers remained skeptical towards the use of obfuscation as a provably secure means of program protection [1,6,7,8], but deriving a theoretical basis for analyzing obfuscation security has been the subject of renewed interest [9,10,11,12,13]. We distinguish between obfuscation (which does not assume underlying cryptographic properties) and program encryption (which has properties of traditional cryptographic ciphers but with a view towards fully executable programs).

In this paper, we further our notions of black-box security [14] and program recognition [15] to lay theoretical foundations based on *random programs* for analyzing obfuscation, tamper-proofing, and piracy prevention measures. We also examine the parallels that are naturally drawn between the security properties of software protection mechanisms and those of cryptographic data protection algorithms. In order to understand properties of a random bit stream program, we first consider the properties of random bit stream data.

1.1. Data Protection

Data ciphers can be defined as algorithms which receive plaintext and produce encrypted ciphertext. Strong ciphers do not reveal or leak non-trivial information about the plaintext: ciphertext must be non-distinguishable from a stream of random bits. Proving security properties of asymmetric and symmetric ciphers is accomplished from an information theoretic viewpoint (secure regardless of computational resources) or complexity viewpoint (secure based on limited resources). By nature of their generality and complexity, information theoretic proofs are harder to produce.

Absolute proofs of security for encryption schemes imply that complexity classes $P \neq NP$. Data protection strength is therefore often stated by properties such as whether breaks are reducible to known hard problems (i.e., factoring or finding a discrete logarithm). Asymmetric ciphers such as RSA use trapdoor one-way functions based on groups or rings to provide the necessary encryption engine. Asymmetric proofs are therefore based on number theory and are distinctly mathematical in nature.

Provable security for symmetric ciphers, on the other hand, does not rely on a strict number theoretic foundation. Symmetric schemes such as DES, AES, and RC4 are based on the use of three basic operations: confusion, diffusion, and composition. No *easy* attacks on symmetric schemes like DES have been found despite voluminous research efforts over the years[1] and there are no known proofs, mathematical attacks, or reductions to known hard problems. Symmetric cryptosystems instead rely on brute force exhaustive search as their strength (computational complexity), yet are considered viable for data protection despite absence of mathematical proof formulations. The security properties of both symmetric and asymmetric cryptosystems are evaluated in decidedly <u>different</u> paradigms

1.2. Software Protection

Program encryption, tamper-proofing, watermarking, and obfuscation schemes can be likened to symmetric data ciphers in many ways. Chief among these similarities are that we need a decidedly <u>different</u> paradigm to reason about and measure software protection. We suggest such a

[1] Observation from RSA Security, http://www.rsasecurity.com

framework in this paper by posing and answering the question of whether random programs exist. Based on our positive assertion, we show how random programs can be used to evaluate white-box and black-box security properties of protected programs and generalized obfuscation methods.

Considering a second similarity between data protection and program protection, the evaluation of data ciphers assumes that mechanisms exist which simulate random bit strings. We can compare encrypted data to the output of a pseudo-random number generator—which is assumed to mimic a truly random number generator given an appropriate seed. Program ciphers, likewise, need to have a baseline for comparison; we refer to this baseline as the "random program". Encrypted programs, unlike encrypted data, must be intelligible to some underlying interpreter or execution engine.

The methods, frameworks, and results for analyzing software protection schemes have varied greatly. For example, the security characterization of obfuscation has been described as non-existent [1], NP-easy [7], derivable in limited contexts [9,11], and proven to be NP-hard [16,17] / PSPACE-hard [18] based on a specific protection mechanisms. We strongly agree with [1] that not all classes of programs *can* be obfuscated but postulate that practical program protection is still feasible short of number theoretic proofs of security.

Comparing data and program ciphers again, obfuscation techniques tend to be spurious, heuristic, and limited; data encryption algorithms tend to be systematic, provably secure, and general in nature. Furthermore, obfuscation techniques do not have the goal of trying to produce predefined properties of *randomness* in the transformed program and they typically do not rely on the key as the basis for transformation in the algorithm (Aucsmith's approach [19] utilizes a key to generated pseudorandom blocks of encrypted code that are decrypted just prior to execution).

Data ciphers rely often on Kerckhoff's principal of security: the cryptosystem is secure given full knowledge of the system except for the key. We define *program encryption* mechanisms as those which are general purpose in nature, have appropriate recovery mechanisms, and are based on a key. These algorithms produce programs which have definable properties of randomness.

Just like symmetric data schemes, program encryption techniques do not have to be seen from a purely number theoretic viewpoint in order to be useful or to offer strong software protection. Algorithms which rely on confusion, diffusion, and composition strategies are not necessarily *weaker* than mathematically based function transformations such as homomorphic encryption schemes [20]. However, a new measurement framework is needed to allow researchers to frame and consider security properties of their techniques.

The remainder of our paper flows accordingly: section 2 defines the notion of random programs and presents questions for consideration. Section 3 discusses the notion of confusion and diffusion as it applies to *executably encrypted programs*—as opposed to programs encrypted by data ciphers that become non-executable. Section 4 defines our theoretical foundation for security based on random programs using a random oracle model. Section 5 points the reader to related works and section 6 summarizes contributions of this work.

2. Random Programs

Like a random bit stream, the purpose of a random program may not be obvious at first glance. We suggest several notions to define "random program" properties.

First of all, random programs are legal program. By definition, a legal program is syntactically and grammatically correct. Random programs also must terminate on all input. Termination may be dependent on the underlying interpreter or environment and can range from reaching the last program statement, executing a HALT instruction, reaching a final state, and so on.

In order to utilize random program notions in protecting programs, we must answer three important theoretical questions:

> Do random programs exist?
>
> Can we generate random programs from non random programs?
>
> Can random programs preserve functionality of the original program?

2.1. Random Bit Stream Programs

We address first the most important question: d*o random programs exits?* One notion of a random program is that the digitized program is indistinguishable from a random bit stream, i.e. that it has no discernable bit patterns, each bit is equally likely to be zero or one, and any sub-string of any reasonable length has approximately the same number of zero's as it has ones.

We illustrate this notion with an abstract machine with a saturated instruction space, e.g. a machine with four operations, sixteen four bit registers and where all operations have two four bit operands and ten bit instructions (table 1). Program output is reflected in the contents of the registers upon program termination. Then any bit stream whose length is divisible by ten represents a valid program in this architecture.

Theorem 1. Random programs exist in the Ten Bit Instruction Architecture (TBIA).

Proof: Generate p', a random bit stream of length 10c, where c is an arbitrarily large constant. Then with instructions interpreted serially from beginning to end, p' is a random program in TBIA.

1. p' is a legal program.
2. It has a meaning

Moreover, p' is random in every reasonable sense of the term in that p' has no patterns in its:

3. static representation, else it would not have been a random bit stream.
4. data representations
5. control flow

QED

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

Table 1. Ten Bit Instruction Machine

2.2. Composition

We would (of course) like to extend these results to more functional architectures in the future. In order to do so, it may be possible to use the important notion of composition. Here we ask a second question: *can random programs be created from other random programs?* It may seem that the composition (catenation) of two random programs is a random program, but we point out that, recursive composition leaves a clear pattern (repeated segment(s)). However, the program resulting from concatenation of atomic (independent) random segments is random.

2.3. Random Instruction Selection Programs

TBIA clearly illustrates that random programs exist. We now extend this notion to a more complex machine were the instruction space is not saturated. For example, extend TBIA to include a fifth operator, say the shift operator that shifts left one bit the value in Rega and stores the result in Regb:

Op	Opnd 1	Opnd 2	Description
SFT	Rega	Regb	Shft Rega left 1 bit-> Store Regb

To accommodate the additional instruction, we may increase the operator length to three bits. Thus, a random bit stream interpreted as a program in TBIA may contain illegal instructions. To address architectures where the operator space is not saturated, we may think of a random program as having the operators equally distributed across the program.

In this scenario, we generate a random program by randomly selecting each operator from all possible operators and similarly selecting the operands. Programs generated in this way have random properties similar to those in TBIA, such as having a similar count of each instruction type, no patterns among operands, and no observable patterns between instructions. During execution, the data and control flow reflect the random properties of the instructions.

The examples in TBIA and its extension clearly illustrate that our model need not be complex or sophisticated to allow random programs. We next consider more sophisticated random programs. Random selection could be used in TBIA architecture to produce random programs. Its added value is that random selection allows a systematic way to generate random programs that avoid illegal instructions. We again rely on random selection to advance the notion of random programs to more sophisticated architectures.

2.4. Random Function Selection Programs (RFSP)

To extend the notion of random programs beyond the simple architecture of TBIA, think of a random program as a collection of higher level structures, composed with no discernable pattern or plan. For example, consider a large library of random program segments i.e. random programs that may be incorporated into another program without modification. We can compose[2] selected segments to create another, larger program, but it is not clear which randomness properties we preserve in the composition.

In a simple architecture like TBIA, it may be possible to recognize usable patterns in the segments, even though they are randomly created. For example, in a one-bit architecture, functionality of every segment is defined as one of the following:

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

Thus, by purposefully selecting segments, the composition may not be random or may even have a usable function with obvious pattern. This concern diminishes rapidly as the architectural complexity increases, since randomly generated segments are less likely to have a usable, recognizable function.

Still, we may also increase the confusion of generated RFSPs by governing segment selection. We retain reference to TBIA because it is sufficiently simple to illustrate our concepts, yet complex enough to give a flavor of its strength.

Given a large constant *cl* (e.g. > 100) a small constant *cs* (say 10< *cs* <30), and an integer *l*, the following algorithm will generate RFSPs of length *l*cs* statements.

1. Generate *cl* * *cs* random statements.
2. Partition the statements into *cl* random segments of length *cs*. Number the segments from one to cl.
3. Create a program *p* by randomly selecting *l* segments (without replacement) and concatenate them.
4. *p* is an RFSP.

[2] In TBIA composition, consists of concatenation. We recognize the administrative actions necessary in higher level languages and posit that these are well understood and that segment compatibility issues can be overcome reasonably easily.

Clearly, *p* is a random program. Were replacement allowed, there would be a possibility of including the same segment more than once, resulting in a discernable pattern and diluting *p's* randomness. However, we observe that, because of the random construction, these patterns reveal very little about the program. This is easily seen if we think of each segment as a named subroutine and replace each segment with its name and arguments to create *p'*. Then *p'* is a random program, since the repeated subroutines are randomly placed.

A final extension of this notion is to consider randomly composing non-random segments. Clearly, this injects patterns into the code. Again, if each of the segments are named and are replaced in *p* with their names, *p* is a random program.

2.5. Random Turing Machines

Finally, we consider random programs as Turing machines. Consider a Turing machine $T = \{Q, \Gamma, S, b, F, \delta\}$ where:

Q is a finite set of states

Γ is a finite set of the tape alphabet

$S \in Q$ is the initial state

$b \in \Gamma$ is the blank symbol

$F \subseteq Q$ is the set of final or accepting states

δ transition function: $Q \times \Gamma^k \rightarrow Q \times (\Gamma \times \{L, R, S\})^k$

Following our pattern, we construct a random Turing machine *t* using the following algorithm.

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

Then *t* is a random Turing machine.

2.6. Section Summary

The consistent theme is that while there may be several ways to think about random programs, each type of randomness has discernable properties, just like random bit streams. The more we know about random program properties, the more likely we will be able to generate intentioned programs that reflect random program properties; this is our goal.

3. Program Encryption

Program encryption requires the encryptor to systematically transform and confuse *p* into *p'* so that an adversary cannot learn anything about program intent by analyzing the static code structure or by observing program execution in great detail. The confusion must make the code and all possible execution paths that it produces display properties of random programs.

3.1. Confusion

Any program encryption approach must confuse and diffuse the original program statements. As with data encryption, program encryption techniques must aggressively scramble or confuse the original program statements, systematically manipulating them with randomizing substitutions.

3.2. Diffusion

This confusing must be distributed across the original program with operations that move confused code unpredictably, based in part on the confused program itself so that the manipulations create diffusion.

3.3. Blind vs. Effective Tampering

The worst case mobile agent risk is a host that performs strict denial of service where the agent is starved for resources, provided the wrong information, or destroyed without execution or migration. No known methods other than tamper-proof hardware can effectively *prevent* these actions by the remote host, but *detection* is possible given proper security mechanisms such as trusted parties and timed execution limits. We refer to this form of alteration as blind tampering because at most an adversarial host can prevent or circumvent correct program execution.

Effective tampering, on the other hand, deals with adversaries who execute remote agents but intend to alter the normal execution of the code to gain some benefit. Such threats include alteration of the agent itinerary, code replay attacks, changing execution pathways, and discovery of proprietary algorithms. Though blind and effective tampering attacks are hard to deal with, the ultimate goal for agent security is to reduce effective tampering to blind tampering.

Consider non-Byzantine faults which do not terminate a program: we would prefer the program to crash instead of continuing to execute with erroneous or possibly corrupted results. The error in this case is at least *identifiable* as a failure. For Byzantine program errors in mobile agents, strict denial of service is easier to detect and recover from than partial denials of service where adversaries effectively alter mobile code for their own malicious intent without detection.

Prevention of effective tampering is the desired goal not only for mobile agents but for general software protection mechanisms. Encrypted programs because of their random properties would reduce denial of service attacks to blind tampering. We consider next the definition of black-box and white-box security in terms of random programs.

4. Evaluating Program Confusion: Random Oracle

In order to evaluate the effectiveness of software protection mechanisms, baseline security characteristics must be established in terms of random programs. In [14], we elaborate the SETS program encryption mechanism that provides black-box perfect secrecy; we reintroduce in definition 1 our statement of strong black-box security. SETS defines how program p can be effectively transformed into a semantically different version p' that produces pseudo-random data output given normal program input. Such a technique disallows an adversary from using any set of observed I/O pairs of program p' for the purpose of black-box algorithm analysis, but allows the original program owner to transform any result y' back to its intended y.

Definition 1: Strong Black-box security

Given program p'

1. implements program encryption algorithm $\mathcal{E}$ $p' = \mathcal{E}(p)$
2. takes input x
3. produces output y = p(x)

After knowing any n I/O pairs $\{(x_1, p'(x_1), (x_2, p'(x_2), \ldots, (x_{n-1}, p'(x_{n-1}), (x_n, p'(x_n))\}$, an adversary that supplies any set of subsequent input $\{x_k, x_{k+1}, x_{k+2}, \ldots\}$ cannot correctly predict in polynomial time either the correct outputs $\{p'(x_k), p'(x_{k+1}), p'(x_{k+2}), \ldots\}$ of the obfuscated program p' or the correct outputs $\{p(x_k), p(x_{k+1}), p(x_{k+q}), \ldots\}$ of the original program p.

In order to effectively measure the protective qualities of a program encryption, we must assume that an adversary has full access to the executable program. Although white-box cryptography is defined in certain contexts as the ability to protect secret keys in untrusted host environments [21,22], we refer to white-box security in the general sense as the ability to shield pertinent program information from an adversary. This shielding assumes the adversary can generate any n set of I/O pairs via execution of the obfuscated program p' AND assumes the adversary has full access to the executable code of p' itself.

We define white-box strength of program encryption based on the existence of a random program oracle. We have demonstrated the existence of random programs with TBIA and described our goal to create intentioned programs that reflect random program properties. We assume the existence of a random program oracle because of the general existence of any random oracle that simulates creation of random data strings. If we can simulate the creation of random data strings, then we can simulate the creation of random programs, as discussed in section 2.

Given the assumption that we want to protect a program p from an adversary who can observe the actions of an encrypted program p', we say that under white-box security the adversary who has possession of p' should not be able to deduce any part of p that would give him greater benefit. Figure 1 illustrates our model where the oracle performs two functions: when given a program, it can generate an encrypted version of that program based on a predefined algorithm $\mathcal{E}(p)$ OR it can generate a random program from the set of all random programs, defined in section 2.

The adversary sends an original program p to the oracle to be encrypted based on an underlying algorithm, $\mathcal{E}(p)$. The algorithm can be any tamper proofing, obfuscation, and piracy prevention mechanisms. The oracle returns the corresponding encrypted program $p' = \mathcal{E}(p)$. as shown below in Figure 1 via the top two arrows labeled 1 and 2.

After the adversary builds some polynomial history of pairs, he generates and sends the oracle a program p_{n+1} as shown by Figure 1, circle 3. The oracle then returns $\hat{p}_{n+1}$ (Figure 1, circle 4) and the adversary is given a decision to make. The question is whether the program $\hat{p}_{n+1}$ given by the oracle is the encrypted version of program $p_{n+1}' = \mathcal{E}(p_{n+1})$. or if it is a random program.

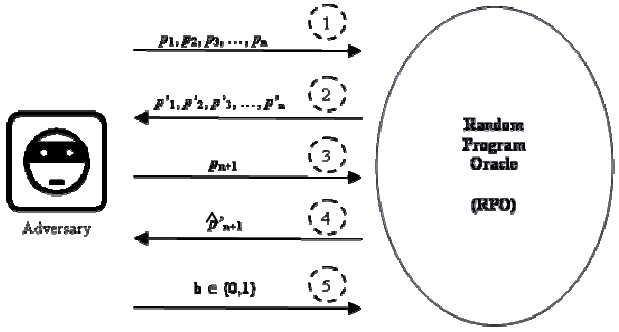

Figure 1: Random Program Oracle

The adversary attempts to make his prediction by returning bit b ∈ {0, 1} corresponding to the guess of either P_R or p'_{n+1} as shown by Figure 1, circle 5 and described in definition 2 of white-box security below.

Definition 2: White-box security

Given access to the random program oracle which transforms programs based on some underlying algorithm $\mathcal{E}(p)$ into an encrypted version p', and given full access to any encrypted program p'_x:

After knowing any n pairs of original and encrypted programs $\{(p_1, p'_1), (p_2, p'_2), \ldots, (p_{n-1}, p'_{n-1}), (p_n, p'_n))\}$, an adversary that supplies a subsequent program p_{n+1} will receive $\hat{p}_{n+1}$ from the oracle which is either: a random program (P_R) or the encrypted version of the program $p_{n+1}' = \mathcal{E}(p_{n+1})$.,

$$\hat{P}'_{n+1} = \begin{cases} P_R & \Pr[\hat{p}'_{n+1} = P_R] \le \tfrac{1}{2} + \varepsilon \\ p'_{n+1} & \Pr[\hat{p}'_{n+1} = p'_{n+1}] \le \tfrac{1}{2} + \varepsilon \end{cases}$$

To ensure provable security, the probability that the adversary is able to predict either the real encrypted program (p'_{n+1}) from a random program (P_R) is less than or equal to ½ + ε, where ε is the negligible error probability.

Having defined the properties of random programs in the previous sections, we thus achieve a computational indistinguishability for proposed encryption algorithms based on random programs and the random oracle model.

5. Related Works

Random programs are an idealized tool that can be used to reason about security properties of software protection mechanisms. A growing body of work exists already on both theoretical and practical application of software protection techniques [1,2,3]. Protecting embedded keys, program integrity, proprietary algorithms, and digital rights are among the most common uses to date [13]. Protection mechanisms can be loosely categorized as piracy prevention, tamperproofing, obfuscation, watermarking, and trusted computing [2].

Though software protection mechanisms have different goals (copy prevention, license enforcement, copy detection, trade secret protection, etc.), random programs can be used as baseline tool for comparing relative strengths and weaknesses of various techniques. We focus on techniques more specific to mobile coding paradigms: code obfuscation, mobile cryptography, and secure multiparty computations.

Obfuscation is the altering of the syntax of a program into a less readable form, while maintaining the same I/O relationships and function of the program [4,5,10,19]. The goal is to reduce the cognitive understanding of a given program. In [2], Naumovich and Memon describe obfuscation as the means to disguise or hide the presence of *other* protection mechanisms. Several obfuscation techniques have been developed that are similar to compiler optimization techniques: variable splitting, interleaving methods, opaque predicates, and reducing flow-graphs to name a few [11,16,17,18].

Mobile cryptography [20] is a form of secure function evaluation designed to provide provably secure code privacy in the general case. Unfortunately, its mathematical foundation on homomorphic properties limits implementation to classes of rational functions. Work by Lee *et al.* [23] has produced a hybrid approach that utilizes function composition and homomorphic properties to protect code privacy and integrity.

When multiple agent parties are involved in a secure computation, several schemes can be used to effectively create white box protection (see [24] for a brief survey). Other implementation specific white-box cryptographic approaches have been posed in [21,22] to prevent extraction of keys within a program implementing data ciphers such as AES and DES.

Several notions have been propose define the cognitive understanding a programmer has concerning a specific piece of software. Zero-knowledge [8], random variables [9], entropy/perfect secrecy [25], virtual black-box [1], and the "knowability" of a function [14] are all various methods used to characterize this knowledge. Our random program model provides a practical framework for defining the semantic meaning of software that correlates to the traditional notions of a data encryption cipher.

6. Conclusion

The notion of a random program as a tool to measure strength of program ciphers and software protection is a novel and useful concept. We have presented in this paper our framework for modeling, viewing, and analyzing program encryption based on random programs. We have proved by demonstration that random programs exist and have given several initial properties that characterize random programs. We have given a definition of program encryption based upon random programs and then provided a model by which white-box and black-box security attributes can be evaluated for a given encrypted program using a random oracle model.

We plan to continue development of a formalized mathematical framework defining characteristics of random programs, produce and show empirical evidence that generalized methods for randomizing programs exist, and expand the architectures by which random programs can be created and verified.

References

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

[2] G. Naumovich and N. Memon, "Preventing piracy, reverse engineering, and tampering," *Computer*, vol. 36, pp. 64-71, 2003.

[3] M. J. Atallah, E. D. Bryant, and M. R. Stytz, "A survey of anti-tamper technologies," *Crosstalk*, 2004.

[4] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, & obfuscation - tools for software protection," *IEEE Trans. on Software Engin.*, vol. 28, pp. 735-746, 2002.

[5] 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.

[6] R. Anderson and M. Kuhn, "Tamper resistance — A cautionary note," in *Proceedings of Second Usenix Workshop on Electronic Commerce*, Oakland, CA, 1996.

[7] A. Appel, "Deobfuscation is in NP," unpublished manuscsript, preprint available from http://www.cs.princeton.edu/~appel/papers/deobfus.pdf, 2002.

[8] S. Hada, "Zero-knowledge and code obfuscation," *ASIACRYPT'2000 - Advances in Cryptology,* 2000.

[9] N. Varnovsky and V. Zakharov, "On the possibility of provably secure obfuscating programs," *Perspectives of System Informatics, LNCS 2890*, pp. 91-102, 2003.

[10] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, "Software obfuscation on a theoretical basis and its implementation," *IEEE Transactions on Fundamentals*, vol. E86-A, 2003.

[11] B. Lynn, M. Prabhakaran, and A. Sahai, "Positive results and techniques for obfuscation," *Eurocrypt'04,* 2004.

[12] A. Yasinsac, "Program Obfuscation: [Im] Possibility Revisited", *Electronic Letters* (submitted June 2005).
[13] P. C. v. Oorschot, "Revisiting software protection," *Proceedings of 6th International Information Security Conference (ISC 2003)*, *LNCS 2851*, Bristol, UK, 2003.
[14] W. Thompson, A. Yasinsac, and J. T. McDonald, "Semantic encryption transformation scheme," in *Proceedings of the International Workshop on Security in Parallel and Dist. Systems (PDCS 2004)*, San Francisco, CA, 2004.
[15] A. Yasinsac, J. T. McDonald, and W. C. Thompson, "Recognizable Programs", *Seventh International Conference on Information and Communications Security*, Dec. 10-13, 2005, Springer (Decision Aug 25).
[16] C. S. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," *Proceedings of the 25th ACM Symposium on Principles of Programming (POPL1998)*, 1998.
[17] C. Wang, "A security architecture for survivability mechanisms," PhD thesis, Department of Computer Science, University of Virginia, 2000.
[18] S. Chow, Y. Gu, H. Johnson and V. A. Zakharov, "An approach to the obfuscation of control-flow of sequential computer programs," *Proceedings of the 4th International Conference on Information Security (ISC 2001), LNCS 2200*, pp. 144-155, 2001.
[19] D. Aucsmith, "Tamper-resistant software: an implementation," *Information Hiding, Proceedings of the 1st International Workshop, LNCS 1174*, 1996
[20] T. Sander and C. F. Tschudin, "On software protection via function hiding," *LNCS 1525*, pp. 111-123, 1998.
[21] S. Chow, P. Eisen, H. Johnson, and P. C. van Oorschot, "White-box cryptography and an AES implementation," *Selected Areas in Cryptography (SAC 2002), LNCS*, 2003.
[22] S. Chow, P. Eisen, H. Johnson, and P. C. van Oorschot, "A white-box DES implementation for DRM applications," *Proceedings of the 2nd ACM Workshop on Digital Rights Management (DRM 2002), LNCS 2696*, pp. 1-15, 2003.
[23] H. Lee, J. Alves-Fos, and S. Harrison, "The construction of secure mobile agents via evaluating encrypted functions," *Web Intelligence and Agent System*, vol. 2, pp. 1-19, 2004.
[24] J. McDonald. "Hybrid approach for secure mobile agent computations," to appear, *Secure Mobile Ad-hoc Networks and Sensors Workshop, MADNES 2005*, Sept. 2005.
[25] 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.