Tamper Resistant Software Through Intent Protection

Alec Yasinsac[1] and J. Todd McDonald[2*]
(Corresponding author: Alec Yasinsac)
Department of Computer Science, Florida State University, Tallahassee, FL 32306, USA[1]
Department of Electrical and Computer Engineering,[2]
Air Force Institute of Technology Wright Patterson AFB, OH 45433, USA
(Email: yasinsac@cs.fsu.edu, jmcdonal@afit.edu)
(Received Oct. 18, 2006; revised and accepted June 12, 2007)

Abstract

One approach to protect distributed systems implemented with mobile code is through program obfuscation. Disguising program intent is a form of information hiding that facilitates tamper proofing. By hiding program intent, adversaries are reduced to non-semantics attacks such as blind disruption or operating system level attacks (e.g. buffer overflows). In this paper, we amplify the Barak result to observe that the Virtual Black Box (VBB) program obfuscation model is fundamentally flawed for useful analysis. We provide an alternative framework for establishing and evaluating program intent protection mechanisms to impede software tampering. Our model reflects more modest goals than VBB. Rather than considering a comprehensive obfuscation view, we detail broad threat classes and propose mechanisms to counter those threats. We then illustrate our model with a protection proof and outline extensions to our results.

Keywords: Mobile agent security, obfuscation, program encryption, software protection, tamper proof software

1 Introduction

Protecting programs from illegitimate use is a classic problem in computer science. To date, there is no comprehensive approach to control distributed software use. Registration, water marking, and copy protection have each achieved some success in preventing and detecting software tampering. We focus on program obfuscation and the stronger concept of program encryption as they can be used to control unauthorized or unintended software use and itemize our primary contributions as:

- Point out limitations in the current standard obfuscation model for defining intent protection while still acknowledging large classes of unobfuscatable programs exist.
- Offer a new obfuscation model with the more modest objectives of characterizing threats and the mechanisms for countering those threats.
- Exercise the new obfuscation model by proposing a mechanism and proving its security properties.
- Introduce the notion of random programs and how it can protect against tampering.
- Introduce the field of program encryption.

The goal of program obfuscation is to disguise programs so that a user can execute them but cannot determine their intent; essentially, it entails constructing programs so that they are unrecognizable in some sense [9]. The goal is that if an adversary does not know what the program is trying to do (in some sense), it does no good to copy it, nor can an adversary change the program in any meaningful way.

Consider a military application where the enemy may capture a hand held device. If an adversary captures the device with a session open, the enemy can observe some number of input and output relationships. If the program is protected against black box analysis, the enemy cannot determine the function of the device from an arbitrary number of input-output pairs. However, a sophisticated adversary may be able to analyze the device from a white box perspective; that is, they can watch the execution of arbitrary input and analyze the control flow and data manipulations as they occur.

Programs that are protected from white box analysis prevent the enemy from learning the program's intent by watching its execution. Program intent protection supports many other security capabilities, including tamper resistance, data protection, even the opportunity to leverage symmetric encryption techniques for pubic key functions.

*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.

One of the primary driving applications for obfuscation is to protect mobile code execution protection and to provide tamper protection in [mobile and non-mobile] agent applications. Mobile agent applications have motivated much of the research in code protection overall and our work specifically.

1.1 Tamper-Resistant Software

Tamper-resistant[1] software is a classic problem in computer science [1, 19, 20], both as a matter of security and of digital rights management. It is difficult to protect program execution, manipulation, and copying in an environment that a sophisticated adversary controls. Normally simple security functions, like protecting an encryption key, are complicated in a hostile execution environment. The adversary's ability to copy the code to attack in parallel combined with the canonical benchmark of guaranteeing security in the worst case paints a bleak picture.

Still, mobile code is an important approach to distributed computing. Sending the execution environment to the data rather than the reverse, can have positive properties, including, but not limited to: (1) reduce network latency (2) encapsulate protocols, (3) execute asynchronously, (4) adapt dynamically, (5) leverage heterogeneity. Additionally, code migration allows process autonomy that mirrors and leverages the highly dynamic execution environment that is characteristic of current and future networks. Similarly, in mobile computing environments, sophisticated intruders may acquire (or capture) for rigorous offline analysis.

We propose to protect against program tampering by hiding their intent; essentially preventing intruders from understanding mobile code in some sense. The implication is that if malicious parties do not know what the program is trying to do, they cannot perpetrate attacks that achieve a predictable result. Thus, their interference is limited to blind intrusion, or at least to a subset of well-known, non-application specific attacks (e.g. buffer overflow attacks that have no semantic application relationship).

There is significant program obfuscation research documented in the literature [9, 10, 17, 18, 33]. Some research focuses on a narrow application domain [14, 16, 29], while other results consider very broad application [35]. We propose a protection approach that evolved from obfuscation research, called program encryption. Possibly the major contribution of this paper is our framework for measuring and categorizing program obfuscation and encryption techniques. This framework allows us to precisely discuss whether or not a given program is recognizable.

[1] We do not claim [or seek] tamper proof programs. A contribution of our paper is to recognize that there are many programs that cannot be obfuscated, though our mechanisms can inhibit a large class of tampering attacks.

1.2 The Threat Model

The Virtual Black Box (VBB) paradigm is a classic approach to analyzing program obfuscation. The notion is that to be effective, obfuscation must prevent information from leaking through the source code. In provable security lingo, this (loosely) means that an adversary should not be able to prove anything when in possession of the obfuscated code that they could not prove with only access to an oracle for the original program. Unfortunately, the VBB model is fundamentally flawed.

First, VBB does not capture the notion of black box analysis to determine program intent [17]. If a sophisticated adversary can use input-output pairs acquired from the oracle to equate the obfuscated program to some known program, the VBB premises are met, but the conclusion is not achieved. Clearly, if an adversary can determine program intent by observing and arbitrary number of input-output pairs, the program is not obfuscated in any meaningful sense.

Additionally, the operational characteristics (size, performance, etc.) of an obfuscated program always give away hints about the original program. For example, simply by looking at program size, an adversary can determine if the program can be intended for constrained computing environment implementation. Beyond this, operational characteristics may convey program semantics. Events such as execution timing or event sequencing, as demonstrated through the canonical covert channel of disk head movement reflecting data encoding [31] reflect embedded functional intent that VBB does not capture. We do not contend that VBB is useless in analyzing obfuscation, only that it is limited in what it captures, and thus, other models are necessary to demonstrate effective hiding of program properties for security.

The VBB flaws result from the breadth the approach seeks, essentially to be a comprehensive model for program obfuscation. Our goal is comparatively modest. As a first step toward a new model, we reduce the objective from general obfuscation to protecting program intent, under a more narrow definition and against specific attacks. For our purposes, we consider intent protection a game between an originator and an adversary or intruder (we use these terms interchangeably). We consider that intruders only desire to recognize programs for three purposes:

1) To manipulate the code in order to attain a known output effect.

2) To manipulate input to attain a known output effect.

3) To understand the input/output correlation for use with contextual information.

We illustrate the first two of these by considering an Internet purchase application where a mobile agent gathers bids for a product or service. If the adversary residing on a visited host recognizes the program, they may manipulate the response to the agent or they may locally

modify the agent code in order to falsely elevate their opportunity to win the bid. Intent hiding does not prevent an intruder from changing input or code, but reduces this type of tampering to blind disruption by preventing the intruder from being able to predict the effect of an input or code change.

The third objective considers an adversarial environment where parties gather information or intelligence, about one another. Here we anticipate that the adversary may gain important information, not about the specific transaction, but about the underlying business practice or strategy that the agent executes. If an adversary is able to understand the program's intent, they may infer fundamental business information from the transaction. Conversely, if we protect program intent, an intruder is unable to gather information about the dispatcher's activity.

The rest of this paper is organized as follows. In the next section, we define program recognizability and illustrate the concept in three separate formalisms. In Section 3 we discuss program encryption and lay out the aspects and techniques that are important to our approach. We end the paper with a summary and conclusions.

2 Understanding Programs

Software engineering experts pride themselves in being able to produce programs whose purpose is clear from the software artifacts alone (source code and documentation). There is substantial value in understanding what a program is supposed to do for maintenance purposes and producing understandable programs has been a software engineering research goal for more than forty years. Program understanding [3] has also been an area of research to analyze or capture mental processes associated with code learning (for the purpose of re-writing or modifying efficiently).

While our work is motivated by using program prediction for malicious purposes and obfuscation for security, the notion of program clarity for maintenance directly applies. A maintenance programmer must be able to understand program intent in order to make purposeful changes, e.g. to fix bugs, improve performance, port to a different environment, etc. In the same sense, a malicious host must understand what a program is doing (in some sense) to effectively copy, modify, run, or forward the program to accomplish a semantics-oriented purpose.

Side effects are an example of an unintended outcome of a program, segment, or construct, or at least an outcome that is not clearly intended. Some programmers consider their code elegant because of their stylistic use of obscure approaches to accomplish intended function in ways that are not obvious. When programs with obscure mechanisms are changed, the maintenance programmer is unlikely to recognize the all impacts of the change. This is good news for security researchers that utilize obfuscation to protect programs since it suggests that understanding programs precisely is a naturally hard problem.

On the other hand, research has produced mixed results on this question. For example, we know that certain control structures are provably difficult to analyze [26]. Conversely, other investigation tells us that obfuscation is impossible in the general contexts [2, 12]. Considerable work has been done to show that obfuscation is possible for point functions (functions that output 1 for only a specific input value and output 0 otherwise) in different security models [5, 6, 22, 34].

The primary impossibility result of [2] formally demonstrates VBB limitations by contriving a family of programs that cannot be VBB obfuscated, illustrating that general, efficient, provably secure, VBB obfuscators do not exist. Other researchers such as [14] are currently moving towards other models or definitions of obfuscation to allow greater freedom for useful results. In this vein, we put forth a model which allows us to leverage other program intent protection properties that avoid the major limitations of the impossibility results. For example, attackers may not require precision; i.e. they may only need a high level understanding or recognition of a functionality subset in order to accomplish their intended malice. Once again, there is little in the literature that quantifies the understanding necessary to maintain, or attack, a program. Before we give such formalizations, we first offer our intuition.

The foundation for our approach is that an adversary only understands a program if they are able to predict its operation in one of two ways. First, an adversary that understands a program can predict a program's output with any given input. For example, for the program that computes the simple function given in Equation (1), an adversary given a small number of input-output pairs need not run the program to strongly suspect that its output is 7 on input 2. As a more complex example, consider a program P that implements a small degree polynomial. Even if an adversary is unable to expose P itself, but can plot a graph based on gathered input-output pairs, they may be able to guess output for a given, arbitrary input without running P.

$$y := x + 5. \tag{1}$$

The second program understanding notion is that an adversary that understands a program can reason about the input required to produce a desired semantic result. For the program P that implements Equation (1), an adversary that understands P and desires that P produce an output of, say 19, knows to feed 14 into the program. This "one-way" property captures an important intent quality.

A common threat to mobile code is that the adversary desires the query to produce a favorable result from their perspective. Accordingly, their goal is to modify the input or code to produce a result with these properties. With intent-protected mobile code, the adversary can only guess the input to produce the desired result with low probability.

We earlier illustrated, adversaries may use intuition and graphing to try to understand a program's intent, but there are many other ways. For example, an adversary may be able to guess the output of P by determining that P is equivalent to another program[2], P_i that the adversary recognizes (i.e. understands its intent). Essentially, the adversary could run P_i as their "prediction process" as long as they are confident that for any relevant input x, $P(x) = P_i(x)$.

We now formalize our definition of program understanding as an entity's ability to derive the input corresponding to an arbitrary output based on their program understanding. While we speak in terms of functional response, we recognize the broader notion of any persistent state change or information transfer to another process or device as output[3].

Definition 1. *Given polynomial-time Turing Machines (TM) $\boldsymbol{B}$ and $\boldsymbol{P}$ and given $\boldsymbol{B}$ has access to the following input:*

1) an arbitrarily large set of I/O pairs from $\boldsymbol{P}$;

2) access to $\boldsymbol{P}$'s code for static analysis;

3) access to $\boldsymbol{P}$'s code for dynamic analysis;

4) a combination of (1), (2), and (3);

5) arbitrary output $y \in \mathrm{Y}$,

***B understands P**: $X \longrightarrow Y$ if and only if*

$$Pr[\boldsymbol{B}(y) = x | \boldsymbol{P}(x) = y, x \in \mathrm{X}] > |X|^{-1} + \epsilon,$$

*where ϵ is a small constant. If no such $\boldsymbol{B}$ exists, we say that $\boldsymbol{P}$ is **intent protected**.*

2.1 Programs and Context

A major challenge to protecting a program's intent is the role that contextual information plays. In most mobile applications, it is impossible to protect all contextual information from the executing host. Items such as program size, execution time, controlled input performance and resource use variations, response to injected errors, and many other operational program aspects are under the executor's control. Application-domain analysis in fact aids in non-malicious reverse engineering efforts rather well [4, 8]. It is a prerequisite for protecting a program's intent that the adversary has limited contextual information available. Thus, there are many programs that inherently cannot be obfuscated.

Consider a program that comes from a vendor known to provide travel plans, and the target host contains only flight information and pricing for a known airline with limited availability dates (e.g. last minute flights). In this case, even a casual observer may infer that the program is gathering flight information to prepare imminent travel plans for the dispatcher's client. Thus, available contextual information and intent protection opportunity are inversely proportional.

A main contribution of our work is that we limit the program obfuscation model's goal by recognizing that there are programs that cannot be obfuscated. In addition, there are programs that can be obfuscated, but our approach is not appropriate to obfuscate them. We do not claim that intent protected programs cannot be attacked; only that they are secure against semantic-specific attacks. For example, many current attacks leverage standard code patterns and known interactions between applications and operating systems. Our mechanisms are not intended to prevent such exploits, though they offer some protection against common cookie-cutter attacks. Additionally, we do not claim to protect against input data manipulation or denial of service attacks except that we prevent such attacks that target program intent, e.g. an executor can filter the input fed into the obfuscated code, but they cannot predict the impact on the program output.

There are three primary approaches to context-independent program intent detection: (1) Input-output (black box) analysis (2) Static analysis (3) Dynamic or Run-time analysis. The later two collectively define white box analysis. Program Recognizability (PR) is a classic concept in computer science and is related to the program understandability notion. Classic PR refers to the context-free notion of being able to determine whether or not a string is a member of a particular language. This is a form of static analysis. Compiler optimization techniques refine the recognizable language classes, allowing program segment identification through signature analysis. Combined with reverse engineering techniques, compiler optimization techniques complicate hiding program intent.

Program intent may become evident through repeated execution and input-output pair analysis, so programs that hide their intent must protect against black box analysis. In preliminary results, we propose general mechanisms to accomplish black box protection [36], propose a white box protection technique based on circuit randomization [23], and demonstrate an end-to-end perfectly secure protection algorithm in limited program contexts [24]. We formalize these notions in this work and extend them to include combinations of both black and white box protection. Malicious parties that acquire code or can corrupt hardware may be able to examine executing code with automated tools such as debuggers. We propose approaches to prevent run-time, dynamic analysis based on the concept, extended from obfuscation research, of *program encryption*. Finally, we recognize that malicious parties are likely to attack intention protection using hybrid methods of combinations of static analysis, black box testing, and dynamic analysis. We also propose mechanisms to protect against these hybrid attacks.

[2] We consider issues of program equivalence in a later section.

[3] Obviously, programs that do not have output in this sense are not necessarily suitable to our obfuscation approach.

2.2 Understandable Programs

We consider Program Understandability (**PU**) to be Boolean. That is, given an arbitrary program p, there may be a function *understand(p)* that returns either true or false. Of course it is possible that **PU()** is Boolean, yet that no program exists that distinguishes between programs that are understandable and those that are not. It is also possible that the Boolean viewpoint is too narrow, for example there may be programs that have no notion of understandability, i.e. programs that have NO overriding intention[4] or pattern (possibly created with that in mind to confound potential intruders[5]).

If **PU()** is Boolean, we can use this to reason about what it means to understand a program. Consider the set of all programs, P. We can partition P into two subsets, the set of all understandable programs (R) and the set of all non-understandable programs (U), where $P = R \cup U$. We observe that many functions are fundamentally understandable, so cannot be obfuscated. For example, any program that implements the function $y = x^2$ is black box recognizable. No matter how random the code implementing this function may be, an adversary need not look at the code to know what the program is doing. It need only conduct black box analysis. The essence of numerous impossibility results for obfuscation [2, 12, 13] all support this thesis as an inherent weakness of general program protection.

Since P is infinite, either R, U, or both are infinite. It is also reasonable to ask if either R or U is empty. In the former, we may argue that ALL programs have unintended impacts at some level of abstraction, or even that our ability to articulate intentions precludes any program from comprehensively meeting them. In the latter, we may point to the Barak result [2] as sufficient to argue that U is empty. We know that simple polynomials are not good candidates for intent protection, but what about strong encryption functions, since we know that these are not susceptible to black box analysis? However, all well-known encryption algorithms have well understood program structures that can be recognized by a sophisticated white box intruder. We offer additional insights on this later.

We have a strong intuition regarding what it means to understand a program from Definition 1. However, we have not formalized what it means for a program to be *understandable*. A program is non-understandable (*obfuscated*) only if it leaks no intention-relative information, for example, if it is indistinguishable from a random program. We argue that this notion is sufficiently strong to preclude intentioned attacks, though we recognize that weaker formalizations may prevent some (or even most) intentioned attacks. Thus, a conservative program encryption goal is to generate *executably-encrypted* code that is indistinguishable from random programs, which we define in the next section.

2.3 Random Programs

When we refer to *random data*, we envision data (e.g. bit streams) produced from a well defined population via unbiased selection. The following properties loosely characterize random bit streams:

- No discernible patterns;
- Each bit is equally likely to be zero as one;
- Any reasonable length sub-string has about the same number of 0s and 1s.

We now suggest several similar "random program" properties. Since programs can be digitized, we could utilize the same characterization as is applied to random data; i.e. that there are no discernable bit patterns, each bit is equally likely to be zero or one, and any sub-string of reasonable length has about the same number of zero's as it has ones. If we do not intend to execute random programs, this characterization works fine; in fact, programs are routinely [data] encrypted for transmission or storage, but these encrypted programs must be decrypted before execution.

It is less obvious whether or not we can retain the random bit stream characterization if we intend for the encrypted program itself to execute. Computers require structure in machine instructions and this structure may inject discernable patterns into programs. Instruction structure is analogous to lexical representation in textual data. We can overcome this limitation if we assume an abstract machine with architecture such that the instruction space is saturated. This means that if instructions are n bits long, the instruction set has 2^n valid instructions and that every (random or other) bit stream of length $x * n$ bits contains x valid instructions. We may think of the instruction length as analogous to block length of data encryption. Unfortunately, here is where the analogy to data encryption diverges.

In data encryption, cipher text has no operational expectation. Its value is exclusively in its randomness and only decryption reveals the potential information that it possesses. Conversely, encrypted programs carry out their purpose through execution. Thus, we should extend the notion of program randomness to include its dynamic, runtime behavior. Clearly, executably-encrypted code constructed via random bit generation must ensure that instructions are not related; though [it appears] such programs offer no intentional opportunity.

We can also think of a random program as having the code equally distributed over the instruction space. We could generate a random program by randomly selecting the operator from all possible operators and similarly selecting the operands. Programs generated in this way would have random properties such as having similar count of each instruction type, no patterns among

[4] Random programs or programs that have no impact on the environment.

[5] We presently ignore the self contradiction of having programs whose purpose is to have no purpose.

operands, and no observable patterns between instructions. Again, it is difficult to envision how such a program could execute successfully, let alone accomplish meaningful intent.

Finally, we could think of a random program as a composition of higher level structures, composed with no discernable pattern or plan. We may construct such a random program by randomly selecting subroutines from a large subroutine library built for this purpose. These programs would display yet different random properties from the first two approaches.

The consistent theme is that while there may be several ways to think about how to select random programs, each type of selection process has discernable randomness properties, analogous to 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.

2.4 Approaches to Understanding Programs

Only imagination and resources limit the number of methods that a motivated and sophisticated adversary can employ to reveal a program's protected intent. Nonetheless, the literature reveals black box and white box analysis as the classical approaches for defeating program obfuscation. Without loss of generality, we address these attacks as if they are accomplished off line, where the adversary has copied the software to a computer with large, but polynomially bounded resources. In practice, the adversary may only be able to employ on line attacks. In any case, off line attacks reflect the stronger adversarial model.

Input-output mappings naturally encapsulate program functionality. Thus, a natural way to try to identify a program's intent is to analyze known input/output pairings. This approach is widely known as Black Box Analysis since this technique treats the program as an oracle (black box) without any insight into its internal workings. While an adversary may capture some I/O pairings during normal system operation, the adversary's ability to execute an intent-protected program off line to generate an arbitrarily large set of I/O pairings is the greater concern.

Definition 2. *Any TM* $\boldsymbol{P}$*:* $X \longrightarrow Y$ *is* ***black box understandable*** *if and only if there exists TM* $\boldsymbol{B}$ *such that given an arbitrarily large set of input-output pairs from* $\boldsymbol{P}$ *and arbitrary output* $y \in Y$*,*

$$Pr[\boldsymbol{B}(y) = x | \boldsymbol{P}(x) = y, x \in X] > |X|^{-1} + \epsilon,$$

where ϵ *is a small constant. If no such* $\boldsymbol{B}$ *exists, we say that* $\boldsymbol{P}$ *is* ***black box intent protected****.*

While black box obfuscation can be helpful in itself, if an adversary can attain the source code of an intent-sensitive program via theft, reverse engineering, etc., they can subject the program to off line scrutiny and automated analysis. These code-based mechanisms look into the clear [or white] box and study program construction and instruction characteristics. The two primary types of white box investigation are static and dynamic analysis.

Static analysis involves actions that an adversary takes without executing the code. Static approaches include inspection, parsing, optimization, pattern matching, etc. These actions can give the adversary hints about the nature of the data, control structures, resources used by the program, etc. Dynamic analysis occurs as the program executes. Run-time tools such as debuggers reveal control flow, data manipulations and evolution, and resource access and consumption. If either static or dynamic analysis or the two applied collaboratively can reveal a program's intent, the program is white box understandable.

Definition 3. *Any TM* $\boldsymbol{P}$*:* $X \longrightarrow Y$ *is* ***static analysis understandable*** *if and only if there exists TM* $\boldsymbol{B}$ *such that given access to* $\boldsymbol{P}$*'s code for static analysis and arbitrary output* $y \in Y$*,*

$$Pr[\boldsymbol{B}(y) = x \in X | \boldsymbol{P}(x) = y] > |X|^{-1} + \epsilon,$$

where ϵ *is a small constant. If no such* $\boldsymbol{B}$ *exists, we say that* $\boldsymbol{P}$ *is* ***static analysis intent protected.***

The third program intent investigative paradigm is dynamic analysis, where the adversary executes the program to observe control flow, variable manipulation, state transition, call sequences, and any other operational program characteristics.

Definition 4. *Any TM* $\boldsymbol{P}$*:* $X \longrightarrow Y$ *is* ***dynamic analysis understandable*** *if and only if there exists TM* $\boldsymbol{B}$ *such that given access to* $\boldsymbol{P}$*'s code for dynamic analysis and arbitrary output* $y \in Y$*,*

$$Pr[\boldsymbol{B}(y) = x | \boldsymbol{P}(x) = y, x \in X] > |X|^{-1} + \epsilon,$$

where ϵ *is a small constant. If no such* $\boldsymbol{B}$ *exists, we say that* $\boldsymbol{P}$ *is* ***dynamic analysis intent protected****.*

Black box, static, and dynamic analysis are separate, related techniques. While they reveal many of the same program properties, they are sufficiently distinct and complementary that their combined application can produce increased effectiveness over their individual operation.

2.5 Related Concepts

2.5.1 Reverse Engineering

We often associate reverse engineering [see Figure 1] with white box analysis for defeating program obfuscation. Reverse engineering produces an abstraction (high level code) from an implementation (low level code) whose goal is to illuminate retain the original code's functionality while also revealing its intent [15]. Our notion of black box/white box understanding is similar to ongoing work [21, 32] aimed at program protection from dynamic or

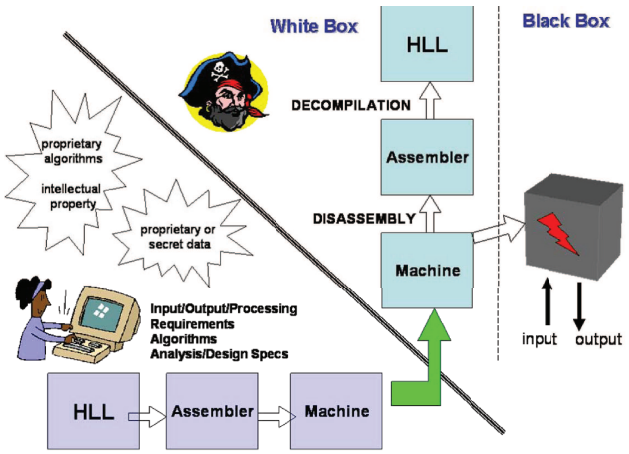

Figure 1: Generic reverse engineering process

static analysis techniques. Program encryption hides algorithmic knowledge without necessarily verifying correct execution or providing data privacy. By breaking the link between how a program functions and its input/output behavior, reverse engineering cannot divulge program intent. Disassembly and decompilation techniques are tools that manipulate code into human-readable form that may reveal program intent.

2.5.2 Program Equivalence

One way to describe **PU()** is as a function of whether or not an adversary can determine if a program q implements functionality A that the adversary is aware of and understands. Thus, if we "understand" functionality A and can show that $q \in P_A$ (where P_A is the set of all programs that implement A) then we can say that we understand program q. One way to show that q implements A is to find or derive a program p_a that is known to implement A, then determine if q is equivalent to p_a. By definition, if q is equivalent to p_a, then q is also an element of P_A.

There is significant research foundation in establishing program equivalence [11, 25]. Much of this work addresses issues of subtle differences in the notion of equivalence. Syntactic equivalence is the strongest form of program equivalence. Two programs that are syntactically identical satisfy all program equivalence notions.

Operational equivalence [27] occurs when two programs can replace one another in any execution environment without negatively impacting the operation. This is a broad form of equivalence that takes into consideration performance, code size, storage efficiency, communication requirements, etc. Finally, two programs are semantically equivalent if they have the same domain and range[6] and if the input/output is identical, i.e. $p_1(a) == p_2(a)$ for all a in the domain of p_1 and p_2. We recognize the weakest equivalence (semantic) as the baseline definition in our model.

[6]We borrow the terms domain and range to reflect the possible input and output sets of programs throughout.

2.5.3 Perfectly Secure Code

Traditional data ciphers are often compared to the strength of the one-time pad–a cipher that reflects perfect secrecy defined by Shannon [30]. Perfect secrecy is achieved when the probability of deriving a plaintext (P) given its enciphered version (C) is the same as deriving the plaintext without any ciphertext to observe. In other words, the ciphertext contains no relevant information that helps in finding the plaintext: $Pr(P|C) = Pr(P)$. When a data cipher is perfectly secure, the probability of deriving the plaintext given the ciphertext becomes the probability of the finding the key itself.

In terms of program ciphers, entropy is a prospective measuring tool. Cartrysse and van der Lubbe [7] define mobile code privacy based on perfect secrecy using the similar notation: $H(p|p', y') = H(p)$. In entropy terms this definition states that the information of a program p, given its encrypted version p' and the output y'(from execution of p') is equivalent to the information of program p itself. This is a restatement of Shannon's notion that the ciphertext (in our case, the encrypted program p' and its output y') provides no additional information about the plaintext (the unencrypted version of p).

Cartrysse and van der Lubbe [7] describe a one-time pad for polynomials (OTTP) that is perfectly secure, but with the stipulation it can only be applied when the set of mobile code is composed of polynomials. Their framework has limits in that the host does not have access to the encrypted output y' for decryption. However, the scheme maintains the notion of an encrypted version p' of the program p by some key K. We develop a framework to reason about program encryption and provide practical implementation techniques for general programs, discussed next.

3 Program Encryption

The notion of data encryption evolved from methods to disguise written or spoken information so that the syntax does not give away the semantics of the message. Early disguises, such as the Caesar Cipher, were simple but effective for their purposes. It took many years for such obfuscation processes to mature into data encryption as we know it today.

Attempts to protect program intentions (semantics) began with modest mechanisms to obfuscate code, e.g., by writing spaghetti code, disguising variable names, casting variables, padding code, etc. Rigorous methods utilizing mathematical approaches emerged more recently. In this section, we propose a model and a mechanism for program intent protection.

3.1 The Model

Thus far we have discussed characteristics of programs as mappings from a pre-image to its image. Traditionally, obfuscation has considered producing different versions

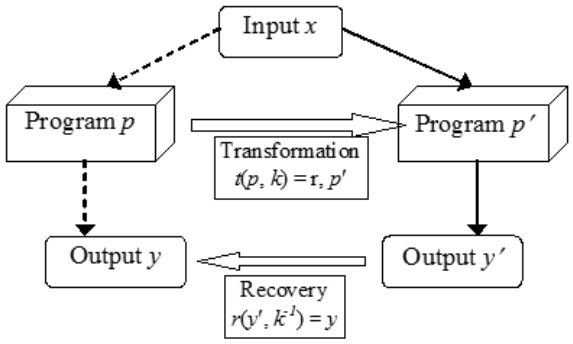

Figure 2: Obfuscation with recovery model

of the same program, where one version is (or likely is) understandable, but the obfuscated version of the same program is not understandable. As we noted earlier, obfuscation in this form was shown to be impossible in the general case [2], offering a bleak outlook for obfuscation-seeking investigators.

A slightly different model renews the hopes of the obfuscation research community. This adjustment allows the investigator to obfuscate a program (say p) by generating a NEW program (p') and a recovery program (r) with the properties that $p(x) = r(p'(x))$ and where r is simple to compute. The fundamental property of the model, shown in Figure 2, is that the output of the cipher code (p') is not equivalent to the output of the original program (p), a property illustrated by Sander and Tschudin [29] and others. This complicates the task of an adversary seeking to reveal the intent of p because discovery of the intent of p' may not reveal the intent of p and there may only be a small percentage of p's intent incorporated into p' by the originator.

The obfuscation process is shown in Figure 2 with a key that may provide security control over the process. Inclusion of the key is a subtle but essential element because it allows us to more nearly mirror the data encryption paradigm. To be cryptographically strong, the method must be public and its strength dependent only on knowledge of the [secret] key.

Our first goal is to produce a transformation process t that generates an unrecognizable program p'. Of course p' must have the functionality reflected in Figure 2 and the transformation process must also produce r with the requisite properties, but the first goal is that t must generate p' that is provably difficult to understand. We introduce one such process in Theorem 1.

3.2 Black Box Protection

We see from Figure 2 that in order to protect the intent of p', the obfuscator must protect p' against black box analysis. We accomplish this by creating p' as the composition of the original program p and a *strong* encryption algorithm e so that for all $x \in X$, $p'(x) = e(p(x))$. Specifically, the transformation process from the original program p is seen in Equation (2):

$$t : \left\{ \begin{array}{l} p' = (p, k) \\ r = d(y', k^{-1}). \end{array} \right\} \tag{2}$$

Cryptographically strong obfuscation results from the nature of strong encryption algorithms. We illustrate this through two the following two lemmas.

Lemma 1. *Any cryptographically strong encryption algorithm is black box intent protected.*

Proof. Arbitrarily select the cryptographically strong encryption algorithm E (i.e., E is a private-key encryption scheme that satisfies indistinguishability under chosen plaintext and nonadaptive chosen ciphertext attacks.) and assume E is black box understandable. Then there exists TM $\mathbf{B}$ such that $\Pr[\mathbf{B}(\mathrm{y}) = \mathrm{x} \mid \mathbf{P}(\mathrm{x}) = \mathrm{y}, \mathrm{x} \in \mathrm{X}] > |X|^{-1} + \epsilon$. This violates cryptographically strong encryption. □

This says that given an arbitrary output, the input to program E cannot be efficiently guessed. Of course, this is an essential property of a strong encryption algorithm. It is also a fundamental property of strong program encryption algorithms.

Lemma 2. *Any program that implements an encryption algorithm with strong semantic security is black box obfuscated.*

Proof Sketch: *Similar to Lemma 1. If an adversary can efficiently guess the cipher text for one plaintext message it can easily distinguish that cipher text from the cipher text of another message. This contradicts the encryption algorithm's strong semantic security.*

We now present a significant contribution of this paper. We give and prove Theorem 1 based on the program intent protection model, where security is guaranteed relative to the specific threat, in this case that threat is black box analysis.

Theorem 1. *Let $t(p, e, k) = (p', r)$ be a process that creates program p' by composing a program p and a black box obfuscated encryption program e. Then p' is black box obfuscated.*

Proof. Follows directly from Lemma 1. If e is black box obfuscated, then p' is also black box obfuscated since the output of p' is [also] the output of e. □

We emphasize that this proof provides the foundation for any further obfuscation. Programs that can be interpreted through black box analysis are not obfuscated. Recall that this is a primary weakness of the VBB paradigm.

This proof effectively overcomes the VBB limitation ensuring us that black box protection is not only possible, but it is fairly easy to accomplish.

Furthermore, it gives us insight into why obfuscation is meaningful. The notion of intentioned manipulation precisely captures an important intrusion category and limits blind disruption to sophisticated intruders. Moreover, it provides a foundation to expand our research into situations where adversaries are able to extract executing code for out of band, white box, analysis.

3.3 White Box Protection

In our model, white box protection requires the originator to systematically confuse p' so that an adversary cannot learn anything about program intent by analyzing the static code structure or by observing program execution. The confusion must make the code and all possible execution paths that it produces display properties of random programs. For example, if a sophisticated adversary can distinguish between the functional program and the composite encryption program, they may be able to extract valuable intent information.

Asymmetric encryption matured slowly until researchers realized a major advance by combining the two well-known, but individually weak approaches of substitution and transposition into the cryptographically strong product cipher that remains the foundation technology of strong asymmetric key cryptography today. We now outline several approaches that can synergistically lead to secure, executably-encrypted code.

3.3.1 Multi-function Programs

By taking the composite of several independent programs, the adversary is challenged not only to recognize many different program intentions, but they may also need to be able to discern which one reflects the originator's the actual intent. Researchers such as Collberg [9] have posed similar mechanisms to introduce multi-functional confusion into the reverse engineering process.

3.3.2 Code Interleaving

One of the factors that lower program understandability is code that accomplishes more than one purpose interwoven together. Researchers have demonstrated the conceptual (not mathematical) hardness of understanding interleaved or multi-functional code [28] and its effect on reverse engineering capability. When programs are (only) concatenated, their code segments are simple for a sophisticated adversary to distinguish. By interleaving code in a way that does not disturb the program's functionality but yet produces measure coalescing, the adversary's workload can be measurably increased.

3.3.3 Module Grouping/Interleaving

Common modules, such as "read", provide natural groupings for deobfuscation analysts. Interleaving unrelated code segments increases deobfuscation complexity.

3.3.4 Systematic Random Statement Insertion

Programs can be made to have random program properties by inserting execution neutral statements with proper characteristics in random and pre-selected places. For example, a program can be made to have the same count of each of several instruction types by inserting no-op instructions of the deficient operator type.

3.3.5 Random Encryption Program Generation

Well known encryption programs have recognizable structures that are fertile targets for deobfuscators. Generating encryption programs based on a systematic approach may provide suitable black box protection while limiting pattern matching and/or control flow attack opportunity.

3.3.6 Randomized Code Blocks (RCB)

Programs can be created to operate in a virtual environment of their own where the original code blocks are randomized and stored. The program's first runtime task would be to re-order these code blocks in memory, based on some key, before continuing execution. This technique would greatly frustrate static analysis tools.

3.3.7 Morphing Code

An extension of RCB, self modifying code offers significant challenges to runtime analysis. Control flow, variable analysis, and other indicators are more difficult to correlate with intent when the program continually changes as it executes.

3.4 Program Confusion and Diffusion

For modern, symmetric key data encryption, cryptographic strength is accomplished through the synergy of two complementary mechanisms: substitution and transposition. Applied alone, neither provides strong content protection, but combined properly, they produce encryption algorithms that have withstood decades of rigorous scrutiny and analysis. Many agree that this synergy comes from the properties of confusion and diffusion that the two approaches provide. Substitution provides rigorous, systematic confusion. Transposition also provides confusion, but provides the added feature of diffusion, spreading each byte of confusion across the cipher text.

Program encryption similarly employs rewriting and interleaving to provide program confusion and diffusion. Rewriting is similar to S-box substitution in data ciphers. Statements and segments are recursively evaluated for

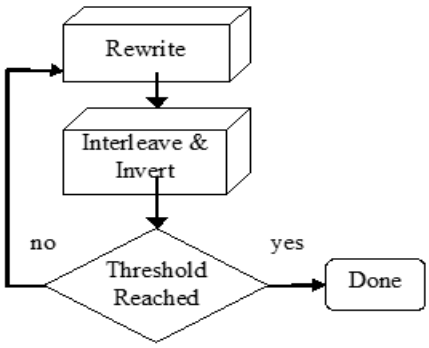

Figure 3: Program encryption

rewriting opportunity and when encountered, they are replaced by a suitable, randomly selected, alternate statement or segment. Conversely, statement/segment interleaving and inversion corresponds to transposition. Suitable statements are randomly selected for interleaving, thus creating different segments available for rewriting. So rewriting and interleaving are performed alternatively and the process recurs for either a fixed number of cycles or until a suitable threshold is reached. Figure 3 illustrates this process.

3.4.1 Rewriting for Program Confusion

S-boxes are powerful mechanisms for implementing strong cryptographic substitution. One lesson of cryptography is that while S-boxes can provide effective confusion, their construction plays a large part in the level of protection they provide. Similarly, statement and segment rewriting rules must be constructed carefully to ensure successful program confusion.

Effective S-box construction is, essentially, guided by randomness. Program randomness must also drive rewriting while also preserving the replaced code's functionality. One approach to meet both objectives is to develop rewrite libraries that provide a large number of potential replacements.

The first requirement of these replacement segments is that they retain the replaced code functionality. They may also employ ruse code insertion that confuses logic flow and statement frequency analysis. These libraries may be automatically created based on well-constructed rules, guided by human interaction with code generators, or manually created by creative software developers. Like S-boxes, effective rewrite libraries will emerge with time and analysis, based on environmental considerations such as user application, source language, etc.

3.4.2 Interleaving and Inversion for Program Transposition

Program transposition is accomplished through statement reordering. Since randomly rearranging statement order is almost assured to change the program's function,

```
proc: Stats(in: grades array; out: mean, median int)
      int: i=0, j=0, sum=0, len=0;
      if empty(grades) {return 0,0; exit;}
1     len := length(grades);            /* begin mean
2     for i=1-len:{sum := sum+grades[i];}end for;
3     mean := round(sum/len);
4     grades:=sort(grades);             /* begin median
5     j:= round_up(len/2);
6     median := grades(j);
      return mean, median;
      end stats;
```

Figure 4: Clear code

```
proc: Days(in: x array; out: day, week int)
      int: i=0, j=0, sum=0, len=0;
      if empty(x) {return 0,0; exit;}
4     x:=sort(x);
1     len := length(x);
2     for i=1-len {sum := sum+x[i];} end for;
5     j:= round_up(len/2);
6     week := x(j);
3     day := round(sum/length(x));
      return day, week;
      end stats;
```

Figure 5: Interleaved code

we must engage other approaches that have randomness properties, but that also retain functionality. Often, independent segments may be merged with statements interleaved with no functional impact. We give an example of code interleaving in Figure 4 and 5, using source code for clarity, though our efforts are largely geared towards lower level code. We number the six reordered statements, again to illustrate how changes may be moved up and down the length of a code segment (Figure 5). We also rename the variables to illustrate how code review may be complicated during code interleaving. In some cases, the name changes are neutral (e.g. changing the recognizable name "grades" to a neutral variable 'x') while others are intentionally misleading (changing "mean" to "day").

While not all segments can be merged, we can detect segments that may be merged using properties such as attribute independence, control flow complexity, and other well-understood principles established in the compiler optimization field. We reemphasize that the intent of segment interleaving is to reorder code to diffuse the confusion created by rewriting. A more direct diffusion method is by statement to statement reordering. We may construct a machine that examines adjacent statement and reorders suitable pairs. Thus, like rewriting, a library may be used to identify allowable exchanges.

The desired synergy is attained by the interactions between rewriting and interleaving/exchanging. Each rewrite produces different segments that may be interleaved and new pairs that may be exchanged. Similarly, each reordering operation creates new segments that may

be rewritten.

4 Conclusions

Tamper-resistant software has many potential uses in information systems, from protecting integrity in E-Commerce applications, to protecting intelligence in combat systems, to protecting intellectual property rights in shrink-wrap software. This paper provides a foundation for building tools that protect program intent from the executing host. Some of the concepts described herein are not new, but are formalizations of loosely constructed notions discussed in other work. These provide the foundation for the novel constructs and proofs that we provide.

Our contribution includes a model complete with definitions, protocols, and proofs that formalize the concept of program intent protection. While our goal is conservative relative to earlier models such as VBB, we show how we can contribute to important security properties with measurable results. We define what it means for a program to be understandable and what a random program is. We also detail notions of program equivalence as a mechanism used to understand intent protected programs. We synthesize these definitions into a model for describing and analyzing properties of program intention protection mechanisms.

This framework enables us to construct mechanisms with provable intent-protection properties and we propose such mechanisms as the first steps in constructing an approach to producing executably-encrypted programs. We give a general mechanism for protecting program intent against black box analysis and prove that the mechanism is strong. Finally, we offer a structure for protecting programs against execution-based analysis based on the proven cryptographic synergy of confusion (through rewriting) and diffusion (through segment interleaving and statement exchanging).

Finally, we introduce the notion of program encryption as the combination of white and black box code protection. Like data encryption, program encryption protects the plaintext's meaning. This is powerful protection that preserves code privacy and prevents semantic attacks against mobile code, essentially reducing the adversary to blind disruption.

5 Acknowledgments

This material is based upon work supported in part by the U.S. Army Research Laboratory (U.S. Army Research Office) under grant numbers DAAD19-02-1-0235 and W911NF-04-1-0415 and sponsorship of the Air Force Research Laboratory, Anti-Tamper Program and Software Protection Initiative (AT-SPI).

References

[1] D. Aucsmith, "Tamper-resistant software: an implementation," *Proceedings of the 1st International Workshop on Information Hiding*, LNCS 1174, pp. 317-333, Springer-Verlag, 1996.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (Im)possibility of obfuscating programs," *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, LNCS 2139, pp. 1-18, Springer-Verlag, 2001.

[3] V. Basili and H. Mills, "Understanding and documenting programs," *IEEE Transactions on Software Engineering*, SE-8, pp. 270-283, 1982.

[4] T. J. Biggerstaff, B. Mitbander, and D. Webster, "Program understanding and the concept assignment problem," *Communications of the ACM*, vol. 37, pp. 72-82, 1994.

[5] R. Canetti, "Towards realizing random oracles: Hash functions that hide all partial information," *Advances in Cryptology – Crypto'97*, LNCS 1294, pp. 455-469, Springer-Verlag, 1997.

[6] R. Canetti, D. Micciancio, and O. Reingold, "Perfectly one-way probabilistic hash functions," *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 131-140, ACM Press, 1998.

[7] K. Cartrysse, and J. C. A Van Der Lubbe, "Privacy in mobile agents," *Proceedings of 1st IEEE Symposium on Multi-Agents Security and Survivability (MASS'04)*, pp. 73–82, Philadelphia, PA, Aug. 2004.

[8] R. Clayton, S. Rugaber, and L. Wills, "On the knowledge required to understand a program," *Proceedings of The Fifth IEEE Working Conference on Reverse Engineering'98*, pp. 69–78, Honolulu, Hawaii, Oct. 1998.

[9] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation," *IEEE Transactions on Software Engineering*, vol. 28, no. 6, pp. 1-13, 2002.

[10] L. D'Anna, B. Matt, A. Reisse, T.V. Vleck, S. Schwab, and P. Leblanc, *Self-Protecting Mobile Agents Obfuscation Report*, Technical Report #03-015, Network Associates Labs, 2003.

[11] R. V. Engelen, D. Whalley, and X. Yuan, "On Automatic validation of code-improving transformations," *Proceedings of ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, pp. 206–210, June 2000.

[12] S. Goldwasser, and Y. Kalai, "On the impossibility obfuscation with auxiliary inputs," *Proceedings of the 46th Annual Symposium on Foundations of Computer Science (FOCS 2005)*, pp. 23-25, Pittsburgh, PA, Oct. 2005.

[13] S. Goldwasser, and G. Rothblum, "On best-possible obfuscation," *Proceedings of the Fourth IACR Theory of Cryptography Conference (TCC 2007)*, LNCS 4392, pp. 194-213, Springer-Verlag, 2007.

[14] D. Hofheinz, J. Malone-Lee, and M. Stam, "Obfuscation for cryptographic purposes," *Proceedings of the Fourth IACR Theory of Cryptography Conference (TCC 2007)*, LNCS 4392, pp. 214-232, Springer-Verlag, 2007.

[15] G. Hoglund, and G. McGraw, *Exploiting Software: How To Break Code*, Addison-Wesley, Boston, 2004.

[16] S. Hohenberger, G. Rothblum, A. Shelat, and V. Vaikuntanathan, "Securely obfuscating re-encryption," *Proceedings of the Fourth IACR Theory of Cryptography Conference (TCC 2007)*, LNCS 4392, pp. 233-252, Springer-Verlag, 2007.

[17] F. Hohl, "Time limited blackbox security: Protecting mobile agents from malicious hosts," *Mobile Agents and Security*, G. Vigna edits, LNCS 1419, pp. 92-113, Springer, 1998.

[18] G. Karjoth, N. Asokan, and C. Gulcu, "Protecting the computation results of freeroaming agents," *Proceedings of the 2nd International Workshop, Mobile Agents 98*, LNCS 1477, pp. 195-207, Springer-Verlag, 1998.

[19] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horowitz, "Specifying and verifying hardware for tamper-resistant software," *Proceedings of IEEE Symposium on Security and Privacy*, pp. 166, Berkeley, CA., May 11-14, 2003.

[20] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *Proceedings of 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOSIX)*, pp. 169-177, Nov. 2000.

[21] C. Linn, and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," *Proceedings of 10th ACM Conference on Computer and Communications Security*, pp. 290-299, 2003.

[22] B. Lynn, M. Prabhakaran, and A. Sahai, "Positive results and techniques for obfuscation," *Advances in Cryptology– Eurocrypt'04*, LNCS 3027, pp. 20-39, Springer-Verlag, 2004.

[23] J. T. McDonald, and A. Yasinsac, "Program intent protection using circuit encryption," *8th International Symposium on System and Information Security*, Sao Jose dos Campos, Sao Paulo, Brazil, Nov. 8-10, 2006.

[24] J. T. McDonald, and A. Yasinsac, "Applications for provably secure intent protection with bounded input-size Programs," *Proceedings of the International Conference on Availability, Reliability and Security (ARES 2007)*, Apr. 10-13, 2007.

[25] G. C. Necula, "Translation validation for an optimizing compiler," *Proceedings of 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 83-94, May 2000.

[26] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, "Software tamper resistance based on the difficulty of interprocedural analysis," *Proceedings of 3rd International Workshop on Information Security Applications (WISA 2002)*, pp. 437-452, Aug. 2002.

[27] A. M. Pitts, "Parametric polymorphism and operational equivalence," *Mathematical Structures in Computer Science*, vol. 10, pp. 321-359, 2000.

[28] S. Rugaber, K. Stirewalt, and L. Wills, "Understanding interleaving code," *Journal of Automated Software Engineering*, vol. 3, pp. 47-76, 1996.

[29] T. Sander, and C. F. Tschudin, "Protecting mobile agents against malicious hosts," *Mobile Agent Security*, G. Vigna edits, LNCS 1648, pp. 44-60, Springer-Verlag, 1998.

[30] C. E. Shannon, "Communication theory of secrecy systems," *Bell Systems Technical Journal*, vol. 28, pp. 656-715, 1949.

[31] G. J. Simmons, "The prisoners' problem and the subliminal channel," *Advances in Cryptology– Proceedings of Crypto '83,* , pp. 51-67, D. Chaum edits, Plenum Press NewYork, 1983.

[32] N. Varnovsky, and V. Zakharov, "On the possibility of provably secure obfuscating programs," *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference (PSI 2003)*, LNCS 2890, pp. 91-102, M. Broy and A. V. Zamulin edit, Springer-Verlag, 2003.

[33] C. Wang, J. Hill, J. Knight, and J. Davidson, *Software Tamper Resistance: Obstructing Static Analysis Of Programs*, Technical Report CS-2000-12, Department of Computer Science, University of Virginia, 2000.

[34] H. Wee, "On obfuscating point functions," *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 523-532, ACM Press, 2005.

[35] G. Wroblewski, *General Method Of Program Code Obfuscation*, Technical Report, Institute of Engineering Cybernetics, Wroclaw University of Technology, 2002.

[36] A. Yasinsac, and J. T. McDonald, "Of unicorns and random programs," *Proceedings of the 3rd IASTED International Conference on Communications and Computer Networks (IASTED/CCN)*, pp. 24–30, Marina del Rey, CA, 2005.

Alec Yasinsac is an Associate Professor in the Department of Computer Science, Florida State University. He joined the faculty in August 1999 after receiving his doctoral degree in Computer Science from the University of Virginia. He has published over forty refereed conference, symposium, and journal papers in the past seven years. He is presently funded by the National Science Foundation, Department of Defense, and the Army Research Office. He has taught over fifty college courses in mathematics, computer science, and information security. His major research interests are network security, cryptography, and security protocols. Dr. Yasinsac is a retired Marine, a senior member of IEEE, and a member of ACM and the IEEE Computer Society.

J. Todd McDonald is an Assistant Professor in the Department of Electrical and Computer Engineering at the A.F. Institute of Technology. He received his doctoral degree in Computer Science from Florida State University in 2006, his M.Sc. degree in Computer Engineering from the Air Force Institute of Technology in 2000, and his B.Sc. degree in Computer Science from the U.S. Air Force Academy in 1990. His research interests include mobile agent security, software protection, obfuscation, and software engineering. Dr. McDonald is an active duty Air Force office and member of ACM and IEEE Computer Society.