Software Issues in Digital Forensics

J. Todd McDonald, Yong C. Kim[*] [†]
Air Force Institute of Technology
Dept. of Electrical and Computer Engineering
WPAFB, OH 45433, USA
{jmcdonal,ykim}@afit.edu

Alec Yasinsac[‡]
Florida State University
Dept. of Computer Science
Tallahassee, FL 32306, USA
yasinsac@cs.fsu.edu

ABSTRACT

Whether we accept it or not, computer systems and the operating systems that direct them are at the heart of major forms of malicious activity. Criminals can use computers as the actual target of their malicious activity (stealing funds electronically from a bank) or use them to support the conduct of criminal activity in general (using a spreadsheet to track drug shipments). In either case, law enforcement needs the ability (when required) to collect evidence from such platforms in a reliable manner that preserves the fingerprints of criminal activity. Though such discussion touches on privacy issues and rules of legal veracity, we focus purely on technological support in this paper. Specifically, we examine and set forth principles of operating system (OS) design that may significantly increase the success of (future) forensic collection efforts. We lay out several OS design attributes that synergistically enhance forensics activities. Specifically, we pose the use of circuit encryption techniques to provide an additional layer of protection above hardware-enforced approaches. We conclude by providing an overarching framework to incorporate these enhancements within the context of OS design.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Privacy Protection; K.6.5 [**Computing Milieux**]: Management of Computing and Information Systems—*invasive software, unauthorized access, physical security*

General Terms

Operating systems, digital forensics

Keywords

Forensic software, operating system extensions, security, evidence collection, circuit encryption, obfuscation

[*]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

[†]This material is based upon work supported in part by the U.S. Air Force Office of Scientific Research under grant number F1ATA07337J001

[‡]This material is based upon work supported in part by the U.S. Army Research Laboratory and the U.S. Army Research Office under grant number DAAD19-02-1-0235

1. INTRODUCTION

Because of the modern rise in computer-related and computer-supported criminal activity, a shift in thinking has occurred in the realm of computer systems design. Currently, we may rightly view any given computer system as a *possible* collection platform for forensic evidence. The future of modern operating systems design can support this trend by adding extensible, integrated capabilities for identifying, collecting, preserving, examining, and analyzing digital evidence.

With the presence of super-privileged malware such as rootkits that have capacity to deceive even the underlying OS, establishing a hardware-based low-level root of trust may be the only possible solution that provably guarantees software protection. Such trusted components are not part of standard processor design currently, but several security-based extensions are finding their way into modern systems. Hardware in the form of secure co-processors, tamper-proof evidence collection storage devices, or built-in high-speed intrusion detection will also require future operating system support.

To the degree that we can design cohesive protective software components, end-to-end support for digital forensics will also require a revolution in how we design software components for operating systems. In this particular work, we consider this future software direction and address areas of common interest for both practitioners and researchers. We show how software protection mechanisms based on circuit encryption techniques offer an added layer of security for protecting forensic evidence.

2. FORENSIC FRIENDLY OS

We may consider the technical aspects of digital forensics as mundane in terms of the actual requirements: the job boils down to data capture, storage, and analysis. However, the adversarial use of counter-forensic software and purposed tampering [9, 10] give impetus for more robust integration between forensic software and operating system. Future operating systems need the ability to support integrated tools with capability for secure static (and live) system/hardware collection. Whether we use independent forensic tools, a security-based middleware, distributed cooperating components, or some combination of these, we envision software designed with a conscience purpose to enhance ”possible” future investigations.

Tamper-resistant and obfuscation techniques [14, 13, 3, 18] will profoundly impact future forensic software design and should guide the development of resilient protective software architectures such as intrusion detection systems (IDS). The ability for the operating system to report on its own activities presumes that, at some point, the operating system itself may be overridden or compromised. Much like a governmental balance of checks and powers between branches, architectural support for forensics must assume a similar system of checks (internally and externally), between the operating system, user applications, and built-in forensics software/hardware.

Obfuscation's goal is to prevent an adversary from using program code to better understand an original program's intent. Context reveals a large amount of information regarding code function and intent, however obfuscation cannot prevent such contextual understanding. Even though this somewhat lessens the usefulness of obfuscation for forensic component protection, we can still reason about normal attack vectors for defeating program obfuscation: black-box and white-box analysis. These attacks are independent in that one need not exercise one in order to leverage the other and complementary in the sense that they can be used together to identify program properties and intent. We characterize intent-protection schemes based upon black-box and white-box definitions.

The systematic application of obfuscation primitives via circuit encryption offers one possibility for end-to-end integration with various OS components and forensic support structures. In order to consider principles for forensic software protection using this approach, we consider first a measurement framework for understanding programmatic intent.

2.1 Black-box and White-box Intent

A traditional problem of security by obfuscation is the inability to characterize both what kind of information and how much information leaks from a program after it has been obfuscated. In order to reason about adversarial capability, we define three different levels of program protection: black-box, white-box, and any combined. At the heart of *black-box understanding*, an adversary would like to be able to predict (based on their understanding of a program) what particular input is needed to achieve a known or desired output. In terms of forensic component protection, an adversary may like to cover their tracks by altering the output of various logging features or override security checks. We capture this intuition in Definition 1 as the ability of an adversary to predict the input to a program given an arbitrary element of the range. The adversary has some polynomial size history of input/output pairs to base their prediction on.

Definition 1. An adversary understands the black-box intent of a program $P \rightarrow \{X, Y\}$ if and only if, given an arbitrarily large set of pairs $IO = \{x_i, y_i \mid y_i = P(x_i)\}$ and given y_j an arbitrary element of Y such that $((\cdot, y_j) \notin IO)$, the adversary can efficiently (in polynomial time) compute x_j such that $y_j = P(x_j)$ on the length of P with with greater than negligible probability. Otherwise, we say P is black-box intent protected.

The notion of white-box protection stems from the desire to measure the static analysis capabilities of an adversary: what can be learned about a program given access only to some form or representation of its source code? Many real-world static analysis tools exist including compilers, code formatters, disassemblers, and decompilers. White-box intent captures an adversary's understanding of the input/output relationships of a program that are only derived from analyzing source code (whether at the high-level, assembly, or machine level). Following our Definition 1, we would describe this capability as the ability to compute a program input corresponding to some arbitrarily chosen output, based solely on non-simulating analysis of the source code. This would roughly correspond, for example, to an adversary who looks for a code within a program in order to override an activation question or possibly a credentials check.

In the real world, adversarial analysis of source code rarely takes place without corresponding input/output analysis or without incremental simulation of code statements. Likewise, real world program analysis may only need to focus on understanding internal portions of a program rather than the whole. Suffice to say, we need a better definition for characterizing white-box intent as it makes no sense to consider white-box analysis alone. However, it may be useful from a theoretic viewpoint to measure or understand how much information leaks from source code if only static (structural) analysis is allowed. We consider these ideas under the notion of full intent protection using the random program model.

2.2 Intent Protection

The notion of intentioned manipulation precisely captures an important intrusion category and limits blind disruption to sophisticated intruders. It also captures a large number of attacks against forensic-based OS components. In [18, 13], we outline the *random program model* as a basis for characterizing combined black-box and white-box intent. This model uses unbiased selection as a measure for randomness and appeals to both notions of intent. An adversary should not be able to learn anything about program intent by analyzing the static code structure or by observing program execution.

A *securely* obfuscated version of a program should make the code and all possible execution paths that it produces display random program properties. Unless the obfuscated version hides the I/O properties of the original program, black-box intent may be divulged by a reasonably equipped adversary. In Definition 2 we give a formal measure for full-intent program protection based on the existence of a random program oracle. The notion of a random program oracle is an oracle that reliably produces random programs given some program size and I/O bound. The oracle also is also used to obfuscate any program that it is given according to some underlying algorithm.

To summarize the model, we provide a basis for comparison between an original program (P) and an alternate version of that program (P'). Obfuscators ($O(\cdot)$) produce alternate program versions ($P' = O(P)$) in order to prevent reverse engineering and effectively disrupt dynamic and static analysis. The alternate version P' is (hopefully) a more confused version of P that prevents such adversarial actions but that still provides the same functionality of P. The analytic intuition of such alternate versions in the random program model is straightforward: if the black-box I/O behavior and the white-box structure of P' cannot be distinguished from a randomly chosen program P_R of the same size category, then there is no correlation between P' and P (or at least no more correlation than between P and a completely random program).

Definition 2. Given access to a random program oracle which transforms any program P using obfuscating algorithm $O(P)$ into an alternate version P', and given full access to any obfuscated program P'_x: After knowing any n pairs of original and encrypted programs $\{(P_1, P'_1), (P_2, P'_2), ..., (P_{n-1}, P'_{n-1}), (P_n, P'_n)\}$, an adversary that supplies a subsequent program P_{n+1} will receive P'_{n+1} from the oracle which is either: a random program (P_R) or the obfuscated version of the program $P'_{n+1} = O(P_{n+1})$. The program $O(P)$ provides (full) intent protection if and only if the probability that an adversary is able to distinguish the obfuscated version (P'_{n+1}) from a random program (P_R) is $\frac{1}{2} + \epsilon$ where ϵ is negligible.

Given this notion for measuring intent protection, we discuss now two properties of forensic components that may increase their resilience to alteration and malicious corruption. Both of these concepts have a role within the context of circuit encryption, which we present in the next section.

2.3 Black-box Components

In forensic evidence collection, investigators seek to know what system compromise has occurred and where the instigator accomplished the activity. Strategically, the operating system can leverage tamper resistant techniques to accomplish several goals: decrease likelihood of intentioned alterations, localize where future violations may occur, and help articulate what adversaries may learn or accomplish. Such coordination will require designers to explore the bounds of specific tamper-resistant methods.

Software, by virtue of its use in generalized Turing-machine based processors, is inherently non-opaque. A highly prized goal of security research centers on the ability to securely hide or protect the intent of programs that run in plain view of underlying (untrusted) computational environments. This goal becomes even higher when we consider the actions of malicious parties manipulating operating system software components in order to hide nefarious activity. The more opaque we can create such components, the less likely that an adversary can understand or manipulate those components for their purposes.

Apart from specific functional classes like point functions, we can only achieve *general* obfuscation in the "perfect" sense by using the truth table look up values for a given function $f(x)$. Truth table values and the programs/circuits we create directly from them offer no notion of white-box intent at all (there are no intermediate calculations or data flow: only I/O). In [14], we explore possibilities for protecting software components with small input size using constructions based on two-level truth table reduction and semantically

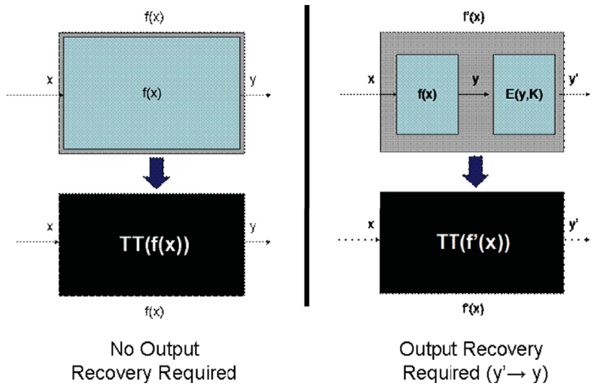


Figure 1: Perfect Obfuscation for Small Input-Size Modules

secure data encryption algorithms (illustrated in Figure 1). Even though the approach is not practical for the general case (program size grows exponentially with input size), it is the only known *general* method to achieve a semantically secure black-box effect of underlying algorithms and their functionalities. By reducing certain segments and modules of operating system code (with appropriate input size) to their black-box, lookup-table functional equivalents, we can effectively shield either the white-box or black-box aspects of underlying components.

As Figure 1 depicts, we can replace a function $f(x)$ with its truth table (TT) lookup equivalent. When we compose the original function $f(x)$ with a semantically strong encryption algorithm ($E(y, K)$), we can provide a truth table version of the resulting function $f'(x)$ that has (recoverable) secure output. With the recovery option, we may hide the full functional intent completely where with no recovery we may hide the code (white-box) intent of the function $f(x)$. Table lookups (in either case) give no notion of intent and may offer great promise in strategically arranging OS components for future forensic support.

2.4 Variation, Mutation, and Randomness

The use of variation has been recognized as a key tenet for defensive programming schemes for quite some time. Cohen [8], for example, offers one of the earliest frameworks to consider Shannon's concepts of diffusion and confusion in the context of code variation. The goal of such security through obscurity focuses on making the difficulty of adversarial actions too costly to succeed. Cohen details several program evolution techniques such as using equivalent instruction sequences, instruction reordering, variable substitution, jump addition/removal, call addition/removal, garbage insertion, program encoding, redundancy, program interleaving, and anti-debugger mutations. These techniques have all found their way into more modern obfuscation tools and research.

As Cohen points out, the right mixture and interleaving of these (program mutating) operations in all aspects of operating system development, deployment, and execution may drive the cost of automated attack up to acceptable levels. We envision the use of variation and randomness in generating operating system components to harden forensic components in a like manner.

As a further enhancement, other researchers are exploring the benefit of self-modifying code to enhance software-based protection mechanisms [12]. We can design code that is not only probabilistically structured (via randomized white-box mutations), we can also design code to probabilistically change itself in a predetermined manner. Such variability can provide the engine for incremental changes in structural descriptions of forensic-based components. These changes add yet another degree of probabilistic analysis an attacker must face in the process of discovering underlying forensic intent.

We contend that an obfuscator that retains semantic equivalence to the original program cannot obfuscate a program that reveals its intent through black box analysis. That is, no matter how scrambled the code, any reasonable adversary can reveal the program's intent.

3. UTILIZING CIRCUIT ENCRYPTION

In [13, 18], we investigate how engineering randomness in program design may offer hope for more general, efficient intent protection for modules with larger input size. In the case of components that are not efficiently distilled via look-up table versions, we appeal to systematic use of code-based obfuscation primitives (code/circuit confusion and code/circuit diffusion). We can compare this technique roughly to symmetric data cipher algorithms that strategically combine diffusion/confusion primitives to offer strong protection.

The application of both randomization and canonical truth table reductions in system design may help pinpoint the activity of future intruders: namely, investigators will know more precisely "where" to look for the sources of compromise and evidence collection. If we can know that certain components offer no opportunity for malicious activity, operating systems of the future may help support forensic activity just by virtue of their design.

We can properly view the process of *circuit encryption* [13] as a set selection mechanism operation. Specifically, if we need to provide an alternate (albeit more confused) version of a program or circuit, then we can view an obfuscating transformation as a mechanism that selects equivalent versions of programs/circuits from a set of functionally equivalent possibilities. We believe circuit/program obfuscators that choose replacements in a uniformly random manner offer the best possibility for tamper resistance and intent protection. In this view, we more properly relate the measure of security to the degree of randomness in the white box structure of a program or circuit. Randomizing obfuscators (if they are possible to create) therefore produce probabilistic (functionally-equivalent) versions of programs with discernible properties of structural randomness.

As Figure 2 depicts, we let P be a program or circuit that we want to intent protect. δ is a set of programs or circuits with a common input/output size $[X, Y]$ and a bounded size $[S]$ (gates, lines of code, etc.). δ_P is a subset of programs or circuits with the same signature (functional behavior / truth table) as P. An obfuscating transformation, $O(\cdot)$, at best can only select an equivalent program/circuit from the

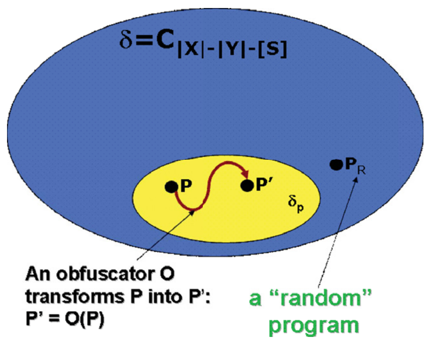


Figure 2: Obfuscation as Set Selection

same set ($C_{|X|-|Y|-[S]}$) or a set with a larger (polynomially-bounded) size ($C_{|X|-|Y|-[f(S)]}$). Figure 2 also depicts that a randomly selected program (P_R) is chosen from the same I/O class set as P; such a selection becomes a valid basis for comparing candidate versions of replacement for P.

As we discuss previously, changing only the internal (white-box) structure of a program/circuit does nothing to hide its black-box intent. We also see this sentiment established in classical theoretical obfuscation results: general, secure, and efficient protection in the Virtual Black Box (VBB) [4] sense is not possible if the obfuscating transformation preserves the same I/O behavior, though no impossibility results are known if the possibility of changing black-box behavior is granted. Under a VBB-based comparison, it can be easily shown that certain programs or circuits have no alternative representation (i.e., regardless of the amount of confusion) that will not leak information when compared to a black-box simulator of the original program. What this means for certain functions is that any version (or variant) of certain programs will leak more information relative to the information obtained from having only simulator access to the program itself.

Since we limit our focus to purposeful manipulation, we avoid certain results related to general obfuscation and leave open the possibility that full intent protection may be achievable. *Circuit encryption* uses a two-pronged approach: first, we transform the input and output behavior of a candidate program/circuit and second, we introduce entropy into the transformed version by iterative, randomizing white-box sub-circuit replacements. As we allude to in Figure 1, transformations that provide recoverability offer hope for hiding intent of a candidate program/circuit, while retaining usable functionality. We envision two distinct methods currently that provide a recoverable means of changing input/output behavior: black-box refinement and semantic transformation. By combining these two methods in efficient probabilistic algorithms and incorporating white-box randomization, we believe that obfuscators may exist that can meet the full intent protection criteria of the random program model.

Figure 3 depicts that I/O based transformations map a candidate program P to another set of functional possibilities. This mapping is considered secret-knowledge, much like a secret key in normal data ciphers. In this example, P is a simple $3-$*input*, $1-$*output*, $8-$*gate* circuit with a truth table

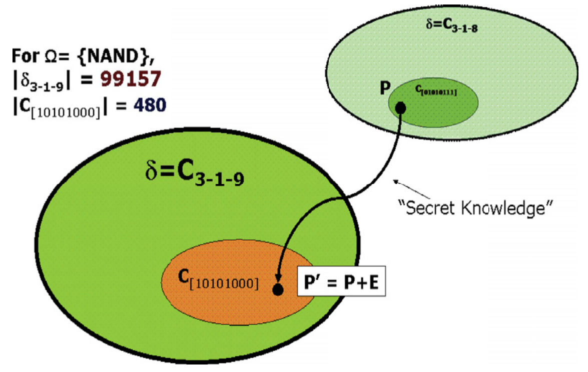


Figure 3: Black-box Transformation

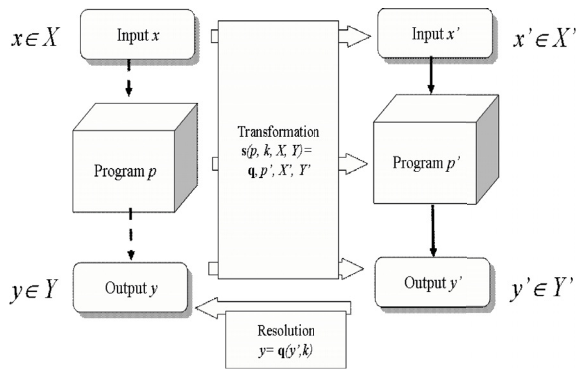


Figure 4: Black-box Refinement

relationship seen as the vector [01010111], representing the 2^3 possible output values based on canonical input ordering ($000 \rightarrow 0, 001 \rightarrow 1, 010 \rightarrow 0, 011 \rightarrow 1$, *etc.*). The I/O transformation in this case does not change either the number of input bits or output bits and is based on a one-way transformation (for 1-bit functions, there are only 2 possibilities: invert all bits or keep all bits the same). The transformation also illustrates the we can recover the output of P when we executed P'. Figure 3 represents a semantic transformation that places $O(P) = P'$ into a functionally different I/O set ($C_{[10101000]}$) and introduces one of the two means we can use to achieve recoverable I/O transformation. We discuss each method and their differences next and finish up with a discussion on white-box randomization.

3.1 Black-box Refinement

We have two goals with I/O transformations that are part of the circuit encryption process. First, P' must have different functional output than P. If we let x and y represent the input size and output size accordingly, then we describe a program/circuit P as $P : 0, 1^{|x|} \rightarrow 0, 1^{|y|}$. If we let x' and y' represent the input size and output size $P' = O(P)$, which is the obfuscated version of P, then $P' : 0, 1^{|x|} \rightarrow 0, 1^{|y|}$. In this regard, we can also state that $\forall(x) : P(x) \neq P'(x)$. For the second goal, we must have some capability to recover the original functional output of P: $\forall(x) : P(x) = S(P'(x))$.

One way of hiding or masking input/output relationships is to do so in plain sight. For a circuit, we can visualize this rather easily and also point to current work where the technique is applied. In [7], for example, Christiansen et al. present a method for creating decoy information pathways through circuits by adding both fake inputs and fake outputs. Interconnections between the input and output gates are generated accordingly. We refer to such techniques as a *black-box refinement* of the original program P and illustrate its algorithmic description in Figure 4. From the viewpoint of a circuit and its corresponding truth table, we can visualize at least five distinct operations that may be part of a black-box refinement and we envision that all five would be applied in a probabilistic manner based on configurable properties found in a secret key.

First, if we let X represent the domain of the original P and confine it to a fixed number of bits, a black-box refinement may add input bits so that a new domain with a larger possible bit string X' is created. Second, we may introduce a random permutation on the input bits themselves which results in a virtual reordering of the bits. Third, we may introduce intermediate gates that would result in new truth table columns for P'. These intermediate gates will need to randomly take input signals from at least every new input gate and some random number of the original input signals of P. Intermediate gates may also be part of connecting new inputs to new outputs which provide a different functional flow within the circuit. Fourth, we can introduce some random number of output gates. Output gates are distinguished intermediate gates and we can therefore use some random selection of newly created intermediate gates or specifically create new gates and backward connect them to the existing circuit or parts of the newly created circuit. Finally, the fifth possible black-box refinement technique involves a random permutation of the output bits themselves, once we introduce any new output gates. As with the domain, we introduce with such operations a possibly new domain Y' which represents a larger bit string that comes as the output of P'.

As Figure 4 depicts, a black-box refinement transformation $s(p, k, X, Y) = q, p', X', Y'$ produces a 4-tuple consisting of a resolution algorithm q, the obfuscated program p', a new domain X', and a new range description Y'. X and Y are the domain and range description of p while k represents secret information that drives the probabilistic behavior of s (a seed to a pseudo-random number generator for example). Program p' now takes input x' from domain X', uses any new intermediate gate logic, and produces output y' which is part of the range Y'. Given the obfuscated program output $p'(x') = y'$, we can then use the resolution algorithm q which takes as input y' and any key information (k) relative to the probabilistic choices made by s.

This algorithm meets both of our requirements for transformation as laid out previously ($\forall(x) : P(x) \neq P'(x)$ and $\forall(x) : P(x) = S(P'(x))$), but there are no absolute security statements that may be assumed between x/x' and y/y'. The fact that the original input and output bits of P still exist in the form of x' and y' which are used and produced by P' does not semantically produce black-box intent protection as we require from Definition 1. However, it does add a

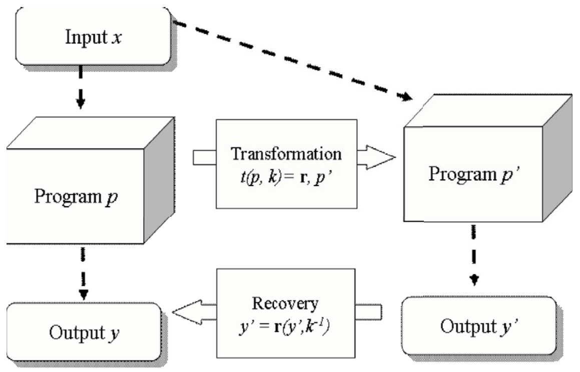


Figure 5: Semantic Transformation

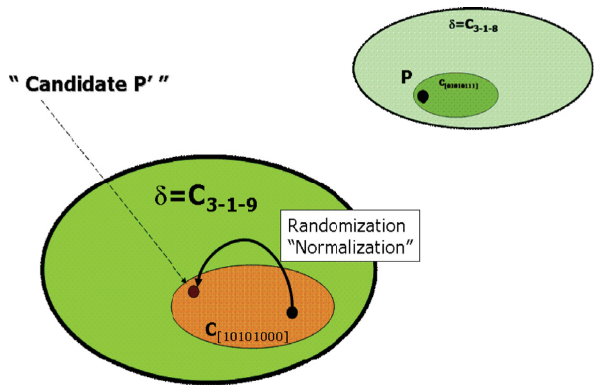


Figure 6: Randomizing White-box Transformation

layer of confusion which provides impetus to further white-box randomization on the structure of P' and also provides a more general method to mask input/output relationships of specialized programs such as point functions. In order to provide a one-way relationship in the black-box properties of a candidate P, we apply semantic transformation techniques after performing black-box refinement.

3.2 Semantic Transformation

In [15, 18, 14] we outline an approach for transforming input/output relationships that we allude to in Figure 1 and now depict more accurately in 5. We let algorithm $t(p, k) = (p', r)$ be a process that creates program p' so that it has a strongly one-way input/output with an original program p. Although other transformations may be possible, we explore transformations that compose the output of program p to the input of strong data encryption algorithms (e). Because semantically secure algorithms are black-box intent protected under Definition 1, it follows that program compositions with a semantically secure algorithm are black-box intent protected as well.

As Figure 5 illustrates, the composition process $t(p, k)$ takes as input both the original program p and also a key k that embodies secret knowledge and points to a one-way identity. In our approach, we embed the key so that $\forall x \in X, p'(x) = e(p(x), k)$. The output of the obfuscation process $t(p, k)$ generates a new program (p') and a recovery program (r) with the property $p(x) = r(p'(x), k^{-1})$, where r is efficiently computable and the output of $p'(x) = y'$ is simple to invert given knowledge of special information (k^{-1}). The obfuscation process uses a key that provides security control and allows correlation with data encryption paradigms. To be cryptographically strong, the obfuscation method must be public and its strength dependent only on knowledge of the key. Though we do not show it, semantic transformation may also increase the output bit size of the resulting p' as it will correspond to the output size of the encryption algorithm e. We also do not discuss the difference between the output size of p in bits ($|p_y|$) and the possible input size of e in bits ($|e_x|$) as there are three possibilities: $|p_y| < |e_x|$, $|p_y| = |e_x|$, and $|p_y| < |e_x|$. Where the sizes are not equal, we must either pad the output of p or provide a means to handle larger input size with multiple encryption algorithms.

Referring back to our example $[3-1-8]$ circuit in Figure 3, the semantic transformation, though trivial (a 1-bit XOR), illustrates the composition and resulting change in I/O behavior. Figure 3 shows that our composed program ($P' = P + E$) resides in its own (different but recoverable) functionality class $\delta_{P'}$. We note also that the new super set of circuits under consideration (C_{3-1-9}) is composed of circuits with larger total gate size than the original P (which was size 8). As indicated in the figure, the new family of $|3| - |1| - [9]$ circuits has 99,157 members (assuming a $NAND$-only circuit basis Ω and assuming we create the set by fully enumerating all possible gate combinations up to the given total size bound). As program and input size grow, the number of possible program replacements also grows super-exponentially.

Once we know that an adversary cannot discern intent from the I/O behavior itself, we can then concern ourselves with the traditional goal of both practical and theoretic obfuscation: providing a replacement program that is functionally equivalent. By combining both black-box refinement and semantic transformation, we meet not only the requirements for black-box intent protection in Definition 1 but also provide high probability that an adversary can not recover the original I/O relationships of a candidate program P based on the I/O relationships of P'. These preliminary transformations form the (required) foundation for considering white-box structural randomization, discussed next.

3.3 White-box Randomization

Once we accomplish I/O transformation of the original program P, we consider the goal of selecting an alternate (but equivalent) version of this intermediate form from a functionally equivalent family. This process, as we mention previously, reduces to an intelligently directed set selection operation. Figure 6 illustrates based on our example, how the specific subset that ($P' = P + E$) belongs to has 480 elements, based on identical truth table signature. We may consider candidate P' replacements from this set and evaluate obfuscators based on how random and uniform the selection process is (or is not). In order to achieve a target level of randomization, the overhead will indeed require circuits with larger total gate size as a result or programs with larger lines of code. Candidate P' circuits with measurable properties of structural randomness will more than likely come only from (considerably) larger circuit family sets of the form $C_{3-1-S'}$, where $S' > S$. We are currently investigating obfuscators that efficiently choose versions of programs and circuits based on these semantic black-box and randomizing white-box transformations.

Figure 7 shows the traditional meaning of obfuscation as understood in both theoretical and practical study: a transformation $w(p, k) = p'$ takes as input a program p with some (possibly) probabilistic information embodied in a key k. The output of w is a program p' that remains functionally equivalent to the original program p and represents a different version (albeit a more confused variant) of the original. Current obfuscation research centers around the transformation algorithm w, in whatever form it takes. We have created a foundational architecture with which to study our approach to white-box change in the context of combinational logic circuits. In our current approach, we have developed algorithms that take a circuit and iteratively introduce random circuit structures (intermediate gates) by replacing (very) small sub-circuits with functionally equivalent sub-circuits. Our current algorithms are exploring the replacement of a single gate with multiple gate variations, selected uniformly and randomly from (efficiently) small circuit family libraries.

In order to reach the goal of full intent protection (as outlined in Definition 2, our final circuit must be indistinguishable from a completely random circuit of the same (relative) size and I/O category. There are many open questions to this problem which will have major ramifications for traditional obfuscation approaches unrelated to circuit encryption. Three questions we currently consider involve the nature of the algorithm itself that induces structural entropy:

1. Are there white-box structural metrics that reveal or indicate randomness or entropy?
2. Are there algorithms which diffuse control flow of a circuit more efficiently than others?
3. Will a polynomially-sized version of a circuit be capable of demonstrating structural randomness in comparison to a completely random one?

Figure 8 illustrates the end-to-end nature of our circuit encryption algorithm. It incorporates both black-box refinement, semantic transformation, and white-box randomization. This algorithm at a minimum meets the black-box intent protection specified by Definition 1 and we continue to experiment with algorithms that focus on white-box only transformations. We believe this technique holds great promise for forensic protection possibilities within OS design, which we discuss next.

4. APPLIED FORENSIC PROTECTION

Our circuit encryption approach offers yet another way to view program variation via systematic and random application of structural function-preserving mutations. It also provides a unique viewpoint for generating polymorphic programs used for security-based OS purposes. The ability to efficiently create probabilistically different versions of an original software component provides defensive protection qualities which creators of *malicious* programs already use extensively. In this light, we can help improve resilience of forensic-based OS components by leveraging applications of circuit encryption techniques.

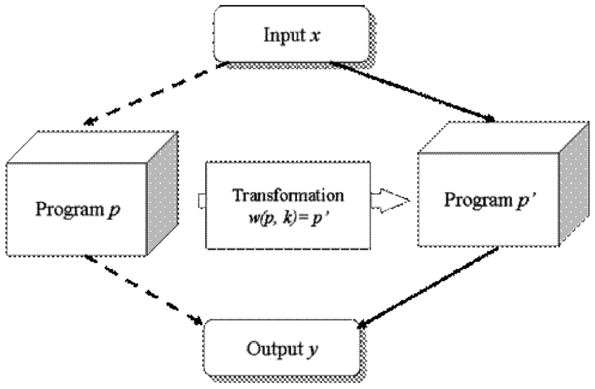


Figure 7: White-box Randomization

4.1 Protected Logging Functions

In order to provide sufficient incident response, investigators need the ability to preserve the digital crime scene and guarantee integrity of data. Log files provide a basic mechanism (currently) to dust for electronic fingerprints of adversarial activity. If we protect all (or even selected) log functions via circuit encryption techniques, an intruder cannot manipulate log creation. Because semantic transformation and its resulting I/O manipulation changes the true output of the logging process, an adversary will not be able to determine what specific information is being logged. Though other methods exist to defeat logs beyond deletion and replacement, preventing primitive manipulation closes certain avenues of attack and reduces the OS attack surface.

Moreover, if an attacker cannot efficiently determine the specific information logs that are recorded, the idea of log replacement loses its appeal. Even in the presence of strong adversarial activity (i.e., via root-kit hypervisor/administrator alteration), logs processed via circuit encryption techniques give kernel functions greater resilience. The highest value evidence for forensic collection, which depends on kernel integrity and unbiased operation, can therefore be isolated from probably adversarial mitigation. As a result of this approach, even the executing system cannot determine the program intent of log file.

4.2 Recoverable Functionality

Besides logging functions themselves, the general techniques related to circuit encryption may hold promise for protecting or enhancing protection of a wide range of OS-related software components. In order to realize such possibilities, we must consider how outputs of protected components (that create encrypted output) might be either used locally or used in conjunction with (other) protected operations.

This particular problem draws many parallels with how traditional mobile agent paradigms have grappled with secu-

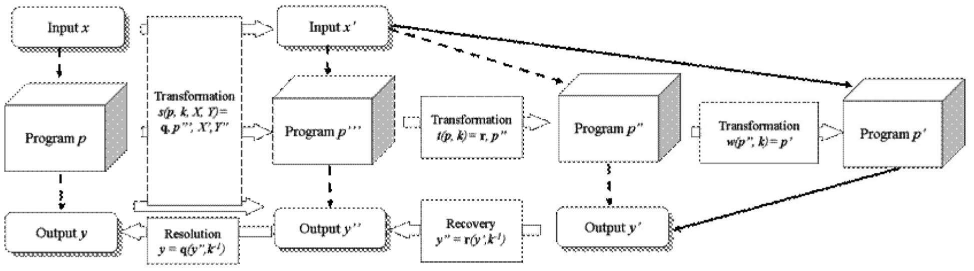


Figure 8: Circuit Encryption Process

rity. Specifically, mobile agents have several security issues when executing on malicious host platforms. As in these environments, we need the ability to use protected (or encrypted) output in the presence of possibly malicious execution environments (hardware or other corrupted OS services/components). Researchers in previous work have given approaches for computing secure functions in adversarial environments via encrypted circuits. Abadi and Feigenbaum [1], for example, describe a multi-round technique while Cachin et al. [5] describe a single-round secure multi-party computation algorithm. Algesheimer and colleagues [2] consider as a follow-on question how the result of the encrypted computation might be used by the mobile agent on the possibly malicious server via use of a semi-trusted third party (a trusted computation service).

As an initial approach to describe how components protected via circuit encryption may use encrypted output, we appeal to the use of composition with appropriate recovery algorithms. Consider for example two software components $f_1()$ and $f_2()$ in which f_2 uses the output of f_1: $f_2(f_1(\cdot))$. Since we cannot present a protected function's output in the clear during the composition process (it becomes subject to I/O analysis), we incorporate a decryption step for candidate functions that require use of local services.

Using semantic transformation techniques, we can semantically protect f_1 by replacing it with f_1' as follows:

$$\forall x : y' = f_1'(x) = e_K(f_1(x)) \tag{1}$$

If we decrypt y' and use it as input to $f_1(\cdot)$ via recovery function $d_{K^{-1}}(\cdot)$, we essentially setup a pipeline composition: (1) decrypt the input (y'); (2) conduct function $f_2(\cdot)$; and (3) encrypt the output (seen as y"). We compose the decryption, function (f_1), and encryption and then use a randomized white-box replacement (f_2') to complete the circuit encryption process.

$$\forall x : y\text{"} = f_2'(f_1'(x)) = e_K(f_2(d_{K^{-1}}(f_1'(x))))) \tag{2}$$

Functions designed to accept only encrypted input will help to strengthen existing operating system components from adversarial alteration. By injecting entropy (engineering randomness) into both input and output of components themselves, we provide cryptographically software-based strength into the forensic shielding process.

4.3 Configurable Key Management

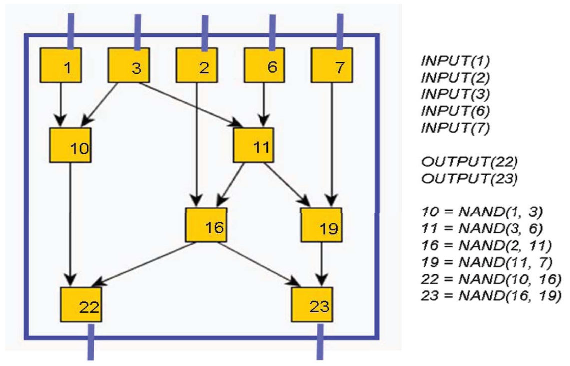
If we introduce key-embedded algorithms to provide I/O protection of important forensic-relevant operating system components, we also induce traditional key management issues at the same time. For composed $P + E$ constructions, the encryption algorithm (E) may use either asymmetric or symmetric keys: in either case, we use E to protect the black-box intent of an underlying functional software component (P). The encryption component itself can provide either a parameterized key or an embedded key option, but we envision predominantly embedded key versions.

As with all cryptographic schemes, key management remains the most problematic issue. We believe that future support of circuit encrypted components will require routine and secure update services, much like automated OS component updates occur currently (Microsoft Windows update feature, for example). Such updates may come more naturally in the form of native circuit definitions and our approach fits best using native circuit description formats.

4.4 Hardware/Software Protection

Because of current trends in the architectural world, the line between software and hardware has become less clear. As Vahid [16] points out, a modern computation platform often requires support by supplementary coprocessors or accelerators. With a new sophisticated video and audio encryption and decryption algorithms, a reconfigurable component like field programmable gate arrays (FPGAs) can configure a particular circuit merely by downloading a particular sequence of bits. Hence Vahid claims a circuit implemented on an FPGA is literally software. A reconfigurable computing platform like FPGAs have hundred of thousands of small lookup tables. These configurable lookup tables can be combined together through use of multiplexors and can be feed into flip-flops to virtually any arbitrary logics. A programmer can implement a desired circuit on an FPGA by writing the proper bits to each lookup tables, routing multiplexors, and flip-flops. Use of such FPGA system can significantly speed up operations hundreds or even thousands of times faster than a generalized microprocessors.

To this point we have left the definition of software and hardware security somewhat up in the air. All computer systems contain a mix of hardware and software and only a limited amount is accomplished with purely hardware. To create a security system purely in hardware would significantly ham-

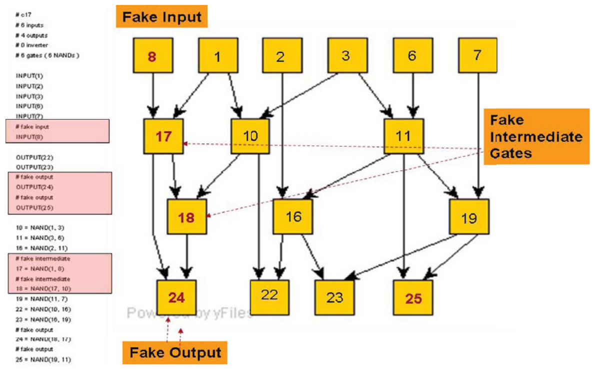


```
INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
INPUT(7)

OUTPUT(22)
OUTPUT(23)

10 = NAND(1, 3)
11 = NAND(3, 6)
16 = NAND(2, 11)
19 = NAND(11, 7)
22 = NAND(10, 16)
23 = NAND(16, 19)
```

Figure 9: Basic Combinational Logic Circuit



```
# c17
# 6 inputs
# 4 outputs
# 0 inverter
# 6 gates ( 6 NANDs )

INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
INPUT(7)
# fake input
INPUT(8)

OUTPUT(22)
OUTPUT(23)
# fake output
OUTPUT(24)
# fake output
OUTPUT(25)

10 = NAND(1, 3)
11 = NAND(3, 6)
16 = NAND(2, 11)
# fake intermediate
17 = NAND(1, 8)
# fake intermediate
18 = NAND(17, 10)
19 = NAND(11, 7)
22 = NAND(10, 16)
23 = NAND(16, 19)
# fake output
24 = NAND(18, 17)
# fake output
25 = NAND(19, 11)
```

Figure 10: Example Black-Box Refinement

per the flexibility and modifiability of such a system reducing the number of future attacks to which a system could potentially respond. Solutions such as an FPGA can be used to extend software flexibility into hardware, though it does require performance tradeoffs to add protection is not pivotal to this aspect of our discussion.

However, a pure hardware solution is not our goal when we talk about hardware-based security. The key component of hardware-based security is the communication between the production system and the security system. Whether a specific monitor is pure hardware, a FPGA, or software running on some combination of hardware that remains separate from the production system hardware, what qualifies a security component as hardware-based is that connection back to the production system.

As forensic-based components need both black-box and white-box protection, circuit encryption techniques offer an ideal solution while keeping the native representation in a circuit-based domain. The integration of FPGAs to embody such functionality also provides an opportunity for consistent and regular updates of key-based components. At a minimum, such configurability provides a way to recover from key-disclosure or key-discovery vulnerabilities without physical replacement of hardware components.

5. QUALITATIVE ASSESSMENT

In order to illustrate the overhead or measurement of circuit encryption techniques, we consider a simple example of small circuit functionality with an appropriate measurement technique. We follow with specific results from a sample test case. Simple combinational logic can easily represent programmatic decisions such as checking or validating a value or computing arithmetic values based on input.

5.1 Rudimentary Example

Consider a circuit chosen from the $C_{5-2-11-NAND}$ family. In particular, this circuit (c17) is a well known part of the ISCAS-85 benchmark suite of circuits. Figure 9 illustrates the gate structure and textual netlist for this circuit. In many cases, the actual part of a program or circuit that we want to protect (from adversarial intent manipulation) is only a small part of some overall larger library or circuit. We can let this circuit (and its associated function) represent some piece of logic that is important for forensic data gathering or protection.

Based on our notional construction for circuit encryption depicted in Figure 8, we first change the original I/O class of the circuit via a black-box refinement process, seen in Figure 10. This process adds one fake input bit, 2 fake output bits, and 2 fake intermediate gates to the original circuit, placing it now in the $C_{6-4-16-NAND}$ family. We introduce these gates via combined pseudo-random/directed algorithm which normalizes the overall gate structure. By knowing which input and output bits are not applicable, we can recover the intended output (easily). However, we provide a greater analysis task for an adversary to understand the input/output relationships of the original circuit. For larger circuits, the black-box refinement process would create random numbers of inputs, outputs, and intermediate gates (even for the same circuit).

After black-box refinement, we can apply a white-box randomization algorithm to the circuit. This process effectively diffuses random structural changes throughout the entire circuit. We are developing several algorithms and corresponding evaluation metrics currently, but all of which involve two basic steps: selection and replacement. The white-box selection process may be random or directed and the replacement process may be random or directed as well, giving us four possibilities for algorithm experimentation. We illustrate the effect of a two-gate random selection and random replacement algorithm on the black-box refined version of the circuit in Figure 11.

Based on this version of the original circuit, we now apply a semantic transformation technique which will create a one-way permutation on the output bits of the circuit. Since our $C_{6-4-16-NAND}$ circuit has 4 outputs, we must (at minimum) provide a 4-bit to 4-bit permutation circuit with strong semantically secure properties. For actual embedded circuits, we would choose to transform the circuit output into a much larger possible output space (128-bit, 64-bit, etc.) and use semantically secure encryption algorithms. We would also consider a probabilistic encryption scheme where the input/output pairs of the cipher are not predictable based on the same input. For this example, we simply create a 4-bit permutation circuit and apply another

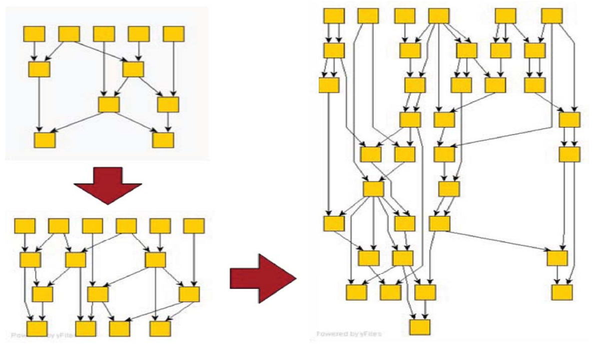


Figure 11: Example White-Box Randomization

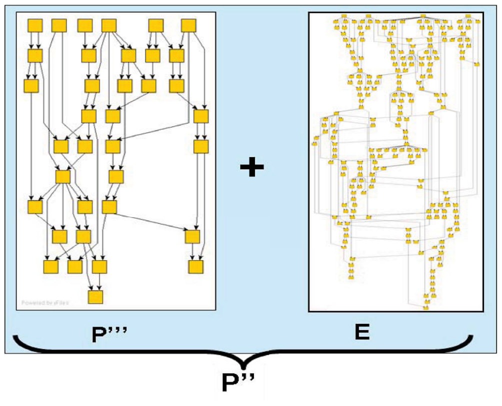


Figure 12: Example Semantic Transformation

round of white-box randomization to create an encryption padding circuit, E.

We now finish the semantic transformation by composing the current version of our c17 circuit with the encryption algorithm, illustrated in Figure 12. This version of the circuit is now candidate for a much more extensive series of white-box randomization algorithms. For simplicity, we illustrate only circuits create from our 2-gate random selection and replacement algorithm. Figure 15 shows the final version of the circuit after 200 rounds of iterative selection and replacement with a summary of changes to size, input, and output.

5.2 Case Study Analysis

For a more appropriate case study example, we consider a generalized comparator to illustrate how a software program can be mapped on hardware, such as an FPGA. We show the results of experiments using circuit encryption techniques in regards to specific metrics that reflect obfuscation properties. Figure 15 shows a generalized comparator for checking a 4-bit key, $A_3A_2A_1A_0$, against a code generated by some program or user supplied data, $B_3B_2B_1B_0$. A higher number of bit key is often used such as a 256-bit key, but the 4-bit example suffices to show how this circuit may be vulnerable to both black-box and white-box attacks without any additional protection. Normally, to compare two numbers we subtract their values and compare to zero. The circuit

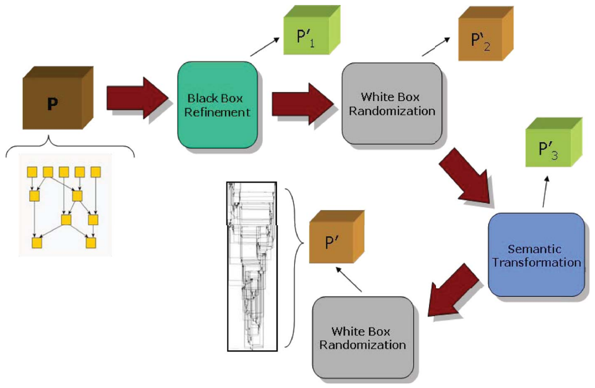


Figure 13: Full Intent Protection

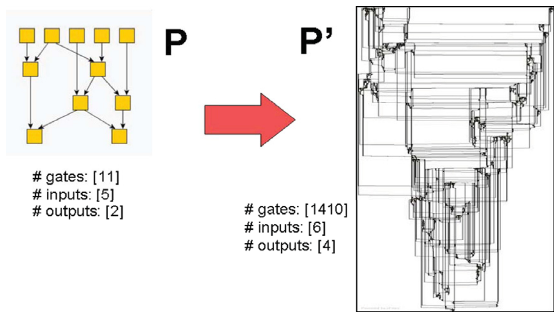
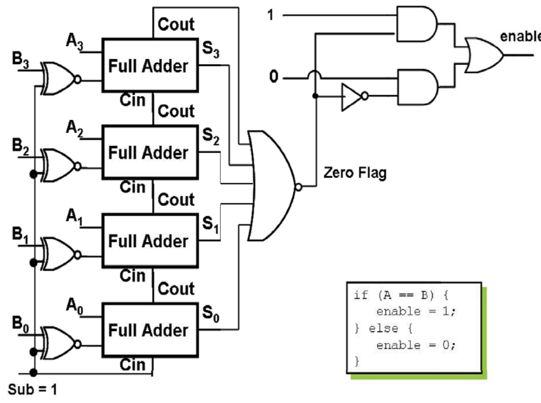


Figure 14: Final Candidate Version of P'

in Figure 15 thus represents a subtractor-based comparator here the value of $A - B$ is first computed. The result is then checked against zero, which corresponds to a wide range of forensic-based authentication and validation tests within the operating system. The output of the zero flag controls a 2-to-1 multiplexor, which assigns the value 1 or 0 to the enable signal. This 1-bit output represents the authentication result.

The notions of controllability and observability of signals in a circuit originate in automatic control theory. Controllability for a digital circuit is defined as the difficulty of setting a particular logic signal to a 0 or a 1. Observability for a signal circuit is defined as the difficulty of observing the state of a logic signal. We consider these metrics important for measuring the strength of circuit encryption techniques because while there are methods of observing the internal signals of a circuit, they are prohibitively expensive. Electron beam testing, for example, can actually scan the VLSI chip-under-test and produce a picture of the chip layout [6]. The signal at logic 0 will appear one color in the image, and those charged to logic 1 appear as another color. However, this testing method is used only for a localized area of circuit and is not practical for larger area due to larger number of transistors. Therefore, an adversary must instead set internal signals by setting signals at primary inputs to primary outputs.

The controllability and observability measures are useful because they approximately quantify how hard it is to set and



```
if (A == B) {
   enable = 1;
} else {
   enable = 0;
}
```

Figure 15: Case Study Comparator Schematic

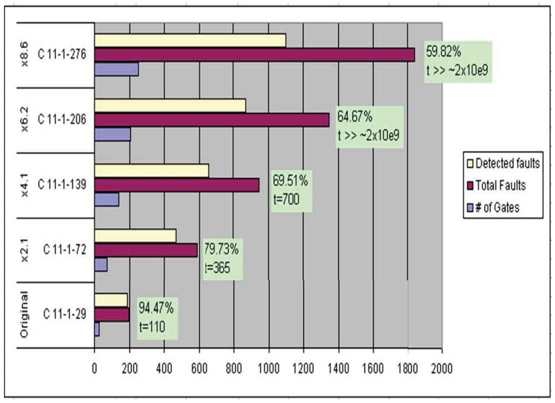


Figure 16: Case Study Findings

observe internal signals of a circuit relative to manipulating white-box intent. Testability measures, which combine both controllability and observability, have two significant attributes. First, they involve topological circuit analysis, without requiring test generation. Testability is also a static type of analysis that can provide an useful measure for the white-box protection of the circuit. Second, it has linear complexity, because otherwise testability analysis is pointless and one might as well use automatic test pattern generation (ATPG) or fault simulation. A linear estimator is useful and preferred since ATPG is a fairly expensive process that involves significant commitments for both time and cost due to the NP-complete nature of test generation algorithms.

Goldstein developed the SCOAP testability measures [11] and contributed a linear complexity algorithm to compute them. Although SCOAP is not absolutely accurate due to reconvergent fan-outs, it is fairly effective in predicting relative coverage levels of the entire circuit fault sets. A number of faults in a circuit under investigation is an effective indicator of how many points of interest exist for a given circuit for probing. More faults often means more places to probe and higher number of detected faults means more about the circuits can be understood since potential monitoring locations (fault sites) can be set to desired value of 1 or 0. A fault coverage shows the percent of faults that can detected.

Figure 16 shows the summary of statistics for testability measures, number of faults, number of detected faults, and fault coverage (seen as a percentage) in the original comparator circuit (depicted in Figure 15) plus four obfuscated versions of the circuit with resulting sizes 2.1x, 4.1x, 6.2x, and 8.6x, respectively. We include the testability measure (t) for each circuit as well. For this study, we note that the two-gate random selection and random replacement algorithm (mentioned in our rudimentary example) causes an exponential increase in testability after only a 6x increase in gate size. As we increase the level of obfuscation, we observe exponential growth in testability measures and a corresponding drop in fault coverage. These measures give us a more qualitative insight into effectiveness of our circuit encryption approach: higher testability measures reflect the difficulty of monitoring and controlling a circuit.

6. SECURITY SPECIFIC SOFTWARE

In forensic evidence collection, investigators seek to know what system compromise has occurred and where the instigator accomplished the activity. Strategically, the operating system can leverage tamper resistant techniques to accomplish several goals:

6.1 Forensic Software Components

Many tools currently exist to assist computer forensic processes. There exist efforts currently (OCFA [1] for example) to develop environments where forensic tools and libraries may be easily inserted-we envision these approaches implemented at the operating system level. Future operating systems will need the ability to support tools that provide static (a posteriori) and live analysis. When things go bad, we need the operating system to readily ”interface” with such tools, but yet shut such investigatory channels off from normal use. Operations such as file carving, write blocking, disk imaging, file analysis, and memory imaging are strong candidates for O/S integration. Forensic components may reduce threats by identifying malicious behavior or proactively assisting investigators. We envision operating systems that have cryptographically protected parts of the operating system that may only be ”opened” (with proper legal authorization) by investigators. These parts may include run time library extensions and the ability to dynamically insert forensic software tools with kernel-level permissions. Protection of the secrecy and integrity of these hooks of course require further analysis and specification.

6.2 Protective Software Components

Tamper resistance attempts to prevent modifications to software for malicious purposes. While these approaches may defend against outside threats from malicious software, insider threats remain the more serious and damaging threat in terms of forensic data discovery. Protective software components such as intrusion detection systems may provide the best solution for detecting or preventing anomalous behavior-regardless of the source from inside or outside. As such, future operating systems may need the ability to offer open-

[1]http://ocfa.sourceforge.net: Open Computer Forensics Architecture (OCFA) is a modular computer forensics framework being used by the Dutch National Police Agency.

source or modular choices for built-in intrusion detection libraries.

6.3 User Application Design

Most instances of software exploitation are really software failure. In [19, 17], we present several arguments regarding software assurance and protection from the perspective of educational paradigm shifts. Even though we cannot eliminate vulnerability from modern information systems, we can reduce exploitable code long term with sound, robust development practices. By starting at the "root cause" for system vulnerability, we may be able to also influence how and where adversarial action may take place. Future application design should not only be *security* aware, but also be focused towards identifying adversarial action. In the same way that knowing where to *point* a camera may specifically reveal a point of intrusion, we would like to know where attacks will most likely come from by virtue of our software design. Education, in large ways, may be the best way to influence future application design in this regard.

6.4 Defending Against Malware and Counter-Forensic Tools

An adversary (one that uses a computer for criminal activity) may use the same tamper-resistant methods and obfuscation approaches that are used to protect "good" software in order to hide their trail or disguise their actions. An awareness of these techniques may also help identify (and pinpoint) the location of activity. Depending on the resilience of the methods used, we may not be able to understand adversarial action by examining malicious code. However, the very use of such techniques with counter-forensic tools may be the primary "evidence" of wrong-doing or an (early) indication of wrong-doing.

7. CONCLUSIONS

By making use of strategic techniques in different areas of software design, we can achieve a level of support for digital forensics. We foresee a more synergistic approach for software components in the future that not only directly support investigatory practices (when required), but also give an awareness of how operating systems are defended and where possible vulnerabilities exist and are most likely to be exploited. We present here an end-to-end approach to forensic software protection at the operating system level based on circuit encryption techniques.

8. REFERENCES

[1] M. Abadi and J. Feigenbaum. Secure circuit evaluation: A protocol based on hiding information from an oracle. *Journal of Cryptology*, 2(1):1–12, 1990.

[2] J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth. Cryptographic security for mobile code. In *Proc of the 2001 IEEE Symposium on Security and Privacy*, pages 2–11, May 2001.

[3] D. Aucsmith. Tamper-resistant software: an implementation. *Lecture Notes in Computer Science*, 1174:317–333, 1996.

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Proc. of the 21st Annual Intl Cryptology Conf on Advances in Cryptology (CRYPTO '01)*, pages 1–18, 19-23 Aug 2001.

[5] C. Cachin, J. Camenisch, J. Kilian, and J. Muller. One-round secure computation and secure autonomous mobile agents. In *Automata, Languages and Programming*, pages 512–523, 2000.

[6] B. D. Christiansen, Y. C. Kim, R. W. Bennington, and C. J. Ristich. Decoy circuits for fpga design protection. In *IEEE Intl Conf on Field Programmable Technology (FPT 2006)*, volume 9, pages 373–376, December 2006.

[7] B. D. Christiansen, Y. C. Kim, R. W. Bennington, and C. J. Ristich. Decoy circuits for fpga design protection. *IEEE Conf on Field Programmable Technology (FPT 2006)*, pages 373–376, Dec. 2006.

[8] F. B. Cohen. Operating system protection through program evolution. *Comput. Secur.*, 12(6):565–584, 1993.

[9] J. P. Craiger, J. Swauger, and C. Marberry. Digital evidence obfuscation: recovery techniques. In *Sensors and C3I Technologies for Homeland Security/Defense IV*, volume 5778, pages 587–594. Intl Society for Optical Engineering, May 2005.

[10] M. Geiger. Evaluating commercial counter-forensic tools. In *Proc of the 2005 Digital Forensic Research Workshop (DFRWS)*, 2005.

[11] L. H. Goldstein. Controllability/observability analysis of digital circuits. *IEEE Trans on Circuits and Systems*, CAS-26(9):685–693, September 1979.

[12] Y. Kanzaki, A. Monden, M. Nakamura, and K. ichi Matsumoto. Exploiting self-modification mechanism for program protection. In *COMPSAC*, pages 170–181, 2003.

[13] J. T. McDonald and A. Yasinsac. Program intent protection using circuit encryption. In *Proc of the 8th Intl Symposium on System and Information Security.* IEEE Computer Society, 8-10 Nov 2006.

[14] J. T. McDonald and A. Yasinsac. Applications for provably secure intent protection with bounded input-size programs. In *Proc of the Intl Conf on Availability, Reliability and Security (ARES 2007).* IEEE Computer Society, 10-13 April 2007.

[15] W. Thompson, A. Yasinsac, and J. T. McDonald. Symmetric encryption transformation scheme. In *Intl Workshop on Security in Parallel and Distr. Systems (PDCS 2004)*, 14-17 September 2004 2004.

[16] F. Vahid. It's time to stop calling circuits ḧardware̤. *Computer*, 40(9):106–108, September 2007.

[17] A. Yasinsac, R. F. Erbacher, D. G. Marks, M. M. Pollitt, and P. M. Sommer. Computer forensics education. *IEEE Security and Privacy*, pages 15–23, July/August 2003.

[18] A. Yasinsac and J. T. McDonald. Of unicorns and random programs. In *Proc of the 3rd IASTED Intl Conf on Communications and Computer Networks (IASTED/CCN)*, 8-10 Nov 2005.

[19] A. Yasinsac and J. T. McDonald. Foundations for security aware software development education. In *Proc of the 39th Hawaii Intl Conference on System Sciences (HICSS '06)*, volume 9, page 219c, 2006.