Analyzing Functional Entropy of Software Intent Protection Schemes

J. Todd McDonald, Eric D. Trias, and Alan C. Lin
Air Force Institute of Technology, Wright Patterson AFB, OH, USA
jmcdonal@afit.edu
etrias@afit.edu
alan.lin@losangeles.af.mil

Abstract: Defending a legitimate software program from a malicious host is a most challenging task. In particular, adversaries may subvert forensics tools, find and exploit known application weaknesses, and reverse-engineer code in order to understand and thwart their intended purposes. From a developer's perspective, one standard defence is to dramatically increase the computational resources an adversary expends on analyzing the client code. We explore in this paper ideas related to intent protection, an approach to software security that combines changes in black-box program behaviour with white-box structural changes. We present experimental results that show whether random input/output patterns may be achievable through systematic code transformations or random variation. As a result, we offer observations on the relationship between functional entropy and correlation with generalized software intent protection schemes.

Keywords: Program Protection, Computer Security, Obfuscation, Hacking, Cryptography

1. Introduction

Software protection is of great interest to commercial industry. Millions of dollars and years of research are invested in the development of proprietary algorithms used in software programs. A reverse engineer that successfully reverses another company's proprietary algorithms can develop a competing product to market in less time and with less money. The threat is even greater in military applications where adversarial reversers can use reverse engineering on unprotected military software to compromise capabilities on the field or develop their own capabilities with significantly less resources. Thus, it is vital to protect software, especially the software's sensitive internal algorithms, from adversarial analysis.

Software protection through *obfuscation* is a growing research area, though military commanders have applied obfuscation or *deception* as a battlefield principle for centuries. In this paper, we evaluate solutions to software obfuscation under the intent protection model (Yasinsac and McDonald, 2008), a combination of white-box and black-box protection that reflects how reverse engineers analyze programs using combined static and dynamic analysis attacks. Specifically, we leverage compositional function tables (CFT) and embedded symmetric key cryptography to produce functional entropy on a small scale for the protection of deterministic functions. Functional tables represent the perfect white-box hiding manifestation because only the input/output pairs are made available. Thus, function tables provide black-box information with no correlation to the original algorithm (intermediate computations).

As a contribution, we examine the effectiveness of using symmetric key cryptography and CFTs as a software-only protection technique. Using the intent protection model, we point out the need to hide both the input/output relationships of a program as well as the programmatic logic embodies in syntactic structures. We define, analyze, and measure the functional entropy of functional compositions with respect to traditional data cipher measures. Of primary interest is how well this approach quantifies obfuscation measures and metrics along cryptographic lines. As part of our work, we propose a set of benchmark programs that exercise the effectiveness of current and future obfuscation techniques.

2. Software Protection

Because of the significant amount of resources invested to produce intellectual properties such as proprietary algorithms, software protection remains a vested interest to the commercial industry. In military applications, commanders concern themselves with unprotected software that may compromise military capabilities or inadvertently transfer technology to foreign adversaries. Thus, recent research into computer security also investigates scenarios where benign software may be operating on a malicious host environment [3].

Java, as a programming language, is particularly sensitive to de-compilation attempts because Java programs are compiled into Java bytecode that runs on virtual machines. This feature makes Java an

attractive choice in producing multi-platform code. However, bytecode makes it significantly easier to understand in comparison with code compiled into native processor machine code. Therefore, Java de-compilers (e.g. Mocha, SourceAgain) can recover source code from unprotected, compiler generated, Java bytecode easily [4]. Even without the tools, Java executables contain internal symbolic information, such as class names, giving reversers extremely helpful information in extracting the original source [5].

2.1 Theoretical Model

Software obfuscation is defined in [1] as a process where an alternate obfuscated version of the original code preserves the original's functionality yet does not allow an adversary to recover any information about the original code. Specifically, a polynomial-bounded adversary should not be able to extract any more information when given an obfuscated program than the information that can be extracted by a machine which has black-box oracle access to the original program. Results from theoretical research, most notably several impossibility results in [1], do not completely dismiss obfuscation as a valid approach to software protection. However, it is generally accepted that a general solution does not exist under the definition stated above. Even in research on a defined set of function families where positive results are established [6, 7], actual implementation of the obfuscator is not presented.

2.2 Anti-reversing Techniques

The cataloging of anti-reversing obfuscating transformations in [2] and follow-on work in [3] provide a basis for practical software protection techniques. Transformations are commonly adopted from other computing fields such as compiler design, file compression, and software engineering. Transformational obfuscation operates within the assumption that given enough time, effort, and resources, a competent reverse engineer will be able to break any obfuscating technique. In contrast to the theoretical perspective, using obfuscating transforms is an admission that perfect, higher levels of security may not be available at this time. However, the authors in [2, 3] and commercial developers of obfuscators advocate that some protection is better than none. And while this concession may not satisfy security and cryptographic experts, [2, 3] provide a functional baseline to work and experiment with.

Because it is assumed that obfuscation techniques are delaying tactics, it is important for a developer and customer to know how much delay can be expected for a given technique. [2, 3] established four properties, derived from seven proposed metrics based on software engineering principles, to evaluate the quality of generic software obfuscation techniques. Table 1 provides a short explanation of the four properties [3, 4].

Properties	Explanation
Potency	Difficulty in understanding the obfuscated code
Resilience	Difficulty in automating a tool to de-obfuscate obfuscated code
Cost	Penalty in execution time/memory space incurred by obfuscated code
Stealth	Statistical similarity of obfuscated code compared to pre-obfuscated code
Quality = (Potency, Resilience, Cost, Stealth)	

Table 1. Characteristics of obfuscating transformations

It is important to note that potency and resilience, two out of the four properties, are heavily dependent on human cognitive ability which is inherently difficult to measure. Potency is by definition related to human cognition. Resilience is also tied to an attempt to measuring human cognitive ability because de-obfuscating programs must be first programmed by a person that understands what and how obfuscation techniques work. While [2] uses research in mature software engineering principle to derive the metrics and corresponding properties, they are nevertheless definitively weaker than metrics used by traditional cryptographers. For instance, figure 1 is a pedagogical illustration on the potential limitation to using metrics based on confusion of human cognition; for some people, it takes mere seconds to recover the original text while for others, it may take much longer. The result of the AltaVista-Bablefish translator as a language "de-obfuscator" is provided in brackets furthering the argument that resilience is also related to a programmer's cognitive ability [8].

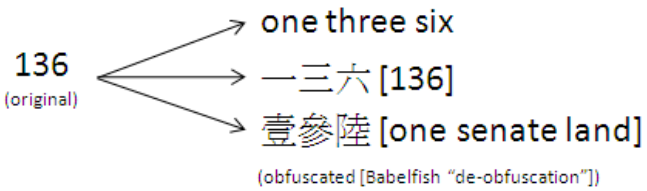

Figure 1. Obfuscations of "136" in increasing order of difficulty for a reader unfamiliar with the traditional form of Chinese characters and their pronunciations.

In contrast, it is currently computationally infeasible for an adversary to break a properly implemented encryption cipher such as the AES which provides 2^{128} bits of security no matter how well the attacker understands cryptography. This limitation suggests that an obfuscation technique with more generically quantifiable metrics independent of human cognitive ability would appeal to both practical obfuscators and theoretical cryptographers.

Three attributes generally characterize software protection techniques: applicability, efficiency, and security. The Holy Grail of software techniques would be one that is general in application, secure in implementation, and efficient in execution. Thus far, however, research in theoretical obfuscation has yielded positive results that are provably secure but applicable for only specific functional families (Lynn and others, 2004:11).

Practical obfuscation approaches use software engineering metrics that are easily applicable to existing software. Security metrics, however, remains a research area because breaking software protection techniques is in part art and in part science. Software engineering metrics were conceived as metrics to gauge the likelihood of coding errors, not as security metrics. Thus, the software engineering derived metrics and corresponding properties for evaluating software obfuscation are understandably weaker than metrics used by traditional cryptographers in evaluating cryptographic algorithms. This suggests obfuscation techniques with more generically quantifiable metrics, independent of cognitive ability, would appeal to both experts in practical and theoretical obfuscation. Table 2 presents software engineering metrics in white (Collberg and others, 1997:8) and cryptographic metrics in grey for comparison ("National Institute," 2001). We note that the software engineering metrics are traditionally used to assess program complexity where an increase in a metric indicates increased overall complexity while the cryptography metrics are used to indicate the randomness of a bit string produced by encryption algorithms or pseudo-random number generators. A bit string with high randomness means that it is difficult to guess the outcome of a bit with greater than 50% accuracy.

Table 1. Measures in Software Engineering and Cryptography

Metric	Short description
Program Length	Number of operators and operands in P
Cyclomatic Complexity	Number of predicates in functions
Nesting Complexity	Number of nesting level of conditionals
Data Flow Complexity	Number of inter-basic block variables
Fan-in/out Complexity	Number of formal parameters and/or global variables
Data Structure Complexity	Number of fields, size, type of static data structures
Object-Orientated Complexity	Number of depth, inheritance, methods, coupling
Frequency	Proportion of 0's and 1's
Frequency Within a Block	Proportion of 0's and 1's within multiple sequences
Longest Runs of 1's in a Block	Length of uninterrupted sequence of 1's
Runs of 0's and 1's	Number of uninterrupted runs of 0's and 1's
Cumulative Sum	Sum of partial sequences after mapping (0,1) to (-1,1)
Random Excursions	Number of cumulative sum cycles with 0 sum
Random Excursions Variant	Number of sums within cumulative sum cycles

This research proposes a software-only approach using compositional function tables (CFT) and embedded symmetric key cryptography to produce functional entropy on a small scale for the protection of deterministic functions. Functional tables are the perfect white-box because only the input/output pairs are made available. Thus, a function table provides just the black-box information.

By replacing a deterministic function with a function table, we strip the structural implementation of the function to prevent white-box analysis by the adversary.

The objective of this research is to examine the effectiveness of symmetric key cryptography and CFTs as a software-only protection technique. Of primary interest is how well this approach quantifies obfuscation strength with measures and metrics consistent with ones used in traditional data cryptography. In addition, this research proposes a set of benchmark programs to demonstrate this approach and may be useful in determining effectiveness of current and future obfuscation techniques. Finally, we evaluate the generality and efficiency of the CFT approach.

We select Java programs and methods to implement our experiments because decompiled Java code of unprotected functions is very similar to the original Java source code providing a greater contrast between decompilations of protected and unprotected code. Compiled Java code is also more understandable because it compiles into a well-documented bytecode format which retains internal symbolic information, such as class names, that help the adversary and Java de-compilers, such as Mocha, reconstruct the original source code and logic. In contrast, C/C++ code compiles into microprocessor instructions that contain less information about the original code and therefore gives less information to an adversary. Popular C/C++ reverse-engineering tools, such as OllyDbg and IdaPro, are dissassmblers which generates the assembly level instructions, rather than the original source code making qualitative comparison against original source code more difficult. Furthermore, Java is a popular choice for web applications that often execute on un-trusted environments. For these reasons, we chose Java as the language to implement this research's experiments (Travis, 2000; Torri and others, 2007).

1.1 3.3 Alternate Obfuscation Model

The VBB model indisputably describes the ideal criteria for software obfuscation. However, theoretical research has shown that this ideal model is impractical. Therefore, an alternative model is necessary to describe a set of obfuscation criteria that does not lead to the same impossibility results produced by Barak and others.

The three criteria established by the VBB model state that an obfuscated version must preserve functionality of the original, perform in equivalent time to the original, and reveal no information about the original that cannot be obtained by having only black-box access to the original. This research examines a model that removes the first criterion: function preservation. Removing this criterion is clearly a weakening of the VBB model, but in turn shelters this new model from the established impossibility results. Of note, this alternate model clearly distinguishes between the structure of program (white-box information) and the function of the program (black-box information) to reflect our observations in Table 1 where we identified differences between the data cryptography model and the general software obfuscation model. McDonald and Yasinsac propose that obfuscation, at best, protects the structure, protects the functional relationship, or protects both naming this the intent protection model (McDonald and Yasinsac, 2007:2-3).

Other research works also support black-box protection of the function output as a means to obfuscate the white-box structure. Sander and Tschudin propose a protocol for computing with encrypted functions (CEF) under the premise that reversing the underlying proprietary functions generally more useful than full reversal of the program (Sander and Tschudin, 1998b:2). Loureiro and others uses a Boolean equation set representation of the function table approach with the McEliece asymmetric cryptographic algorithm which encrypts the output as an obfuscation technique (Loureiro and others, 2002:4). Chow and others also use combinations of function tables to integrate their white-box version of the AES algorithm to protect other functions (Chow and others, 2002:252). These works all emphasize the need to modify the functionality of the original function as part of an obfuscation technique. We noted that the output is unusable until it is converted back to some usable form, which is usually done on a trusted environment. Figure 5 graphically illustrates this intent protection model for comparison with the VBB standard obfuscation model in Figure 3 and the standard cryptography model in Figure 2.

While it appears that this is the client-server model, there is a key distinction. Traditional client-server hides the proprietary algorithm on the server side forcing the server to bear the computational load. In contrast, the objective of the partial client-server model is to safely offload the computational load

onto the client. For example, a MaS agent, such as the ones described in the previous chapter, can perform secure computations within an un-trusted execution environment and then send information back to the issuer.

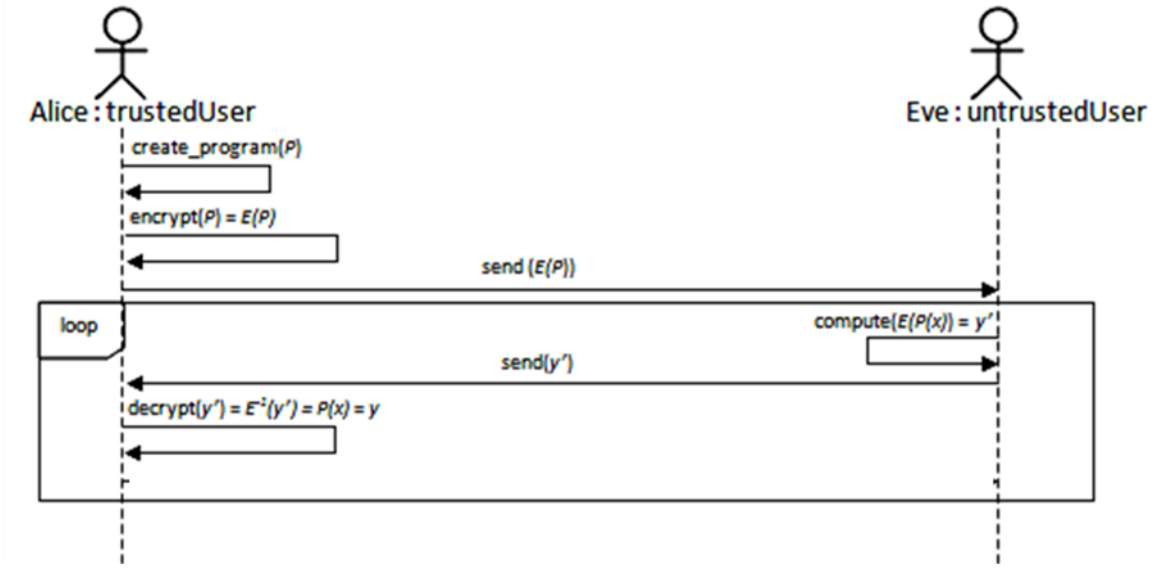

Figure 1. Intent Protection Obfuscation Model

Intent protection weakens the first criterion (functional preservation) of the VBB model. However, by providing functional confidentiality, it may be possible to strengthen protection overall through the third VBB criterion (structural confidentiality). Because VBB requires functional preservation, analysis of black-box information in the original and obfuscated version of the function may allow the adversary to extrapolate the white-box information. This is acceptable, though unintuitive, in the VBB model, especially when we know that adversaries use a combination of black-box and white-box attacks. Conversely, if it is acceptable within an obfuscation model to change the functionality of the obfuscated program, then it is possible to apply techniques that prevent deduction of white-box information through black-box analysis.

We thus revisit data cryptographic techniques since their primary function prevents black-box analysis. We note that any encryption of the output is still in a white-box attackable environment and thus methods for white-box encryption require examination. Figure 6 illustrates the data cryptographic model on the left with the intent protection model on the right for comparison.

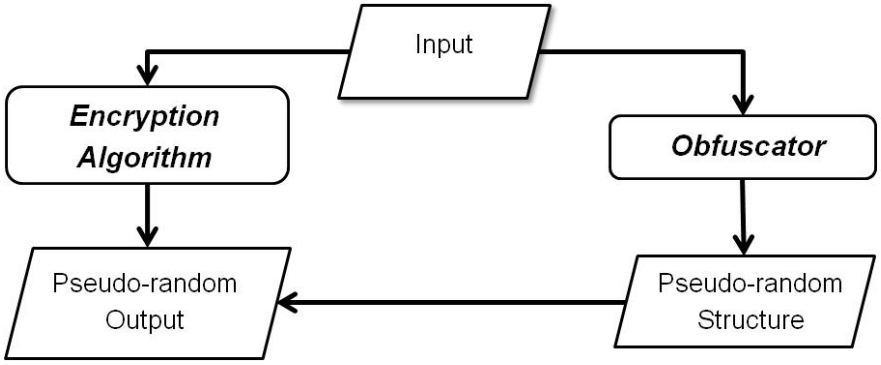

Figure 2. Data cryptography (left) and Intent Protection (right)

As stated previously, we divide a function into its functional behavior and its structure. While an encryption only makes the output appear as random output, we postulate that an obfuscator must also protect the white-box information. We could achieve this by either removing structural information or by emulating structural randomness. Thus, this research examines the input/output produced by random programs for comparison with similar sized functions to gain understanding on

the relationship between the randomness in a program structure with the randomness in the output. To the best knowledge of this research, the relationship between random structures and corresponding output characteristics is unknown. If obfuscation is analogous to cryptography, then we can make the same analytical comparisons on the output. For instance, in order to gauge how well an encryption produces a pseudo-random output, it must exhibit characteristics comparable to a truly random sequence. The National Institute of Standards and Technology (NIST) published a list of established metrics that can empirically determine how closely a sequence exhibits randomness ("National Institute," 2001). Methods to accurately assess the level of randomness of function or program structure are, at this point, unknown and a reason why it is difficult to practically evaluate practical obfuscation techniques under VBB model's security test posed by the third criterion.

This research postulates that any program generated by randomly selecting bit manipulations between the input and output is a random program. Specific implementation details on how this research creates random programs are in the experimental section. By creating randomly generated programs, it is possible to examine their output using statistical measures. If random programs generate non-random output, then it is possible that obfuscation through randomization of structure is sufficient because the output does not correlate strongly to the structure. An indicator of this would be a large set of random programs that produce the same output pattern. If random programs tend to generate random output, then any program, original or obfuscated, that does not produce random output may indicate that a strong relationship between black-box patterns and white-box structure exists. Therefore, even if randomness is induced into the structure, it may never be sufficiently enough due to the predictability of the output. Security then requires a mechanism to produce randomness in the output which intent protection model supports (Hofheinz and others, 2007:17; Algesheimer and others, 2003:5).

Figure 7 illustrates the comparisons made in this research under the intent protection model relative to the comparisons made in the VBB model. In summary, the obfuscation community has yet to agree on how to make structure comparisons for white-box security. Without consensus on the structural security measure, it is difficult for practical obfuscation techniques to claim meeting the VBB security test as shown by the leftmost arrow. Thus, we propose the random program model, where *O(P)* is made to functionally and structurally resemble random programs (P_R), as a derivation of the random oracle model in cryptography (Bellare and Rogaway, 1995). Constructing P_R serves as an intermediate step in understanding and evaluating function structure and output patterns. We can use the results to develop techniques so that *O(P)* exhibits both functional and structural characteristics of P_R.

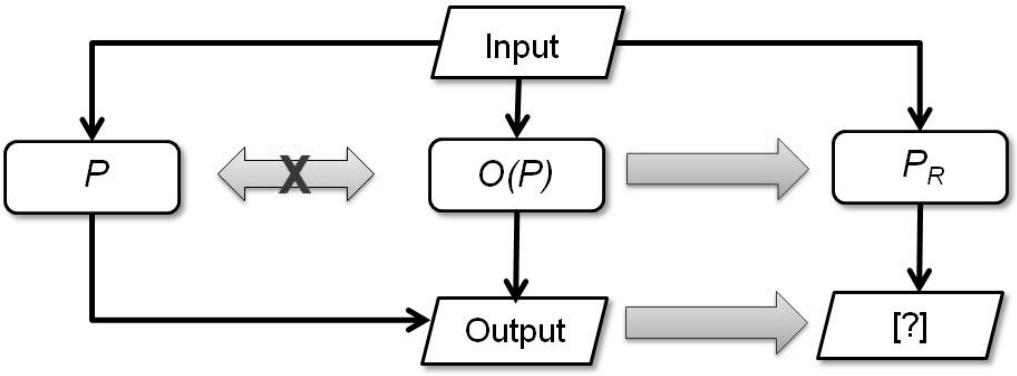

Figure 3. Obfuscation and Random Programs

Canetti and others prove in their work that work that techniques secure in the random based methodology may be insecure in implementation. However, we note that the cryptographic community uses the random oracle because the standard cryptography model based solely on complexity measures is difficult to prove. Therefore, our appeal to randomness is primarily to establish a sanity check on obfuscation approaches, as recommended by Canetti and others in their conclusions, in absence of a stronger security model (Canetti and others, 2006).

Metric

Black-box Metrics

Black-box metrics were adapted from a NIST test suite for pseudo-random number generators. For clarification, we consider each output bit as a generator of a bit string and use statistical analysis to determine if patterns exist for each bit enabling the adversary to guess subsequent bits within the bit string.

We list in Table 7 the statistical tests used in this research and a summarized explanation for each test ("National Institute," 2007). We recognize there are existing test suites such as JDieHard, NESSIE, and the one provided by the NIST; however, these suites were designed for random program generators and some tests required minimum bit string lengths of 10,000 bits or greater. Thus, we had to selectively implement tests that could provide results on much smaller bit string lengths due to our experimental benchmarks that have a relatively smaller input space.

Table 2. Statistical Test to Analyze Function Output

Test	Explanation
Frequency (Sequence)	Ratio of 1's to 0's produced in an output bit
Frequency (Output)	Ratio of 1's to 0's produced by all output bits
Longest runs of 1's	Longest uninterrupted sequence of 1's
Number of 1's runs	Number of runs with uninterrupted sequence of 1's
Maximum excursion	Greatest distance from zero achieved when each output resulting in 0 or 1 is mapped to -1 and 1 respectively and the output's bit string is summed.
Excursion states	Size of the set of distances from zero achieved when each output resulting in 0 or 1 is mapped to -1 and 1 respectively and the output's bit string is summed.
Zero excursion cycles	Number of zero excursion cycles. A cycle the summation of the outputs to an *m*-th bit and back to the origin when each output bit resulting in 0 or 1 is mapped to -1 and 1 respectively; *m* is increased incrementally until it reaches the end of the bit string.
Approximate entropy	Percentage of output bits flipped when a single input bit is flipped; used when gray-code input is used.

White-ox Metrics.

Metrics to evaluate the randomness of a structure is the subject on concurrent research within the PEG research group. Because the proposed approach removes the program structure by converting a function into a two dimensional representation, this research can subjectively examine the structural obfuscation of using CFT using various Java decompilers. We derive quantitative security measures from the steps that an adversary needs to perform to break apart the CFT along with the computational complexity associated with each step, as stated in section 3.5; the theoretical maximum security according to computational complexity is directly correlated to input size.

3.11.3 Side-channel Metrics.

Performance and memory costs are important because they determine the practicality of the obfuscation. We measure performance as execution time in seconds and measure cost in terms of memory size in bits. These metrics are common, non-subjective, and understandable within the computer science. Because cost of the CFT implementation is very different from the time it takes to *generate* a CFT implementation, a developer must decide whether generation costs should factor into the cost of obfuscation. For consistency, we only consider the memory cost of the deployed obfuscation and the performance running the obfuscated function when evaluating an obfuscated program. We note that multiple obfuscations of different functions will cost less to generate because the paired encryption table only needs to be generated once. Thus, future obfuscations of functions with the same bounded input-size incur incrementally less generation costs because we can pair it with any pre-enumerated encryption table.

We compare the above metrics against the four properties of an obfuscation proposed by Collberg and others' work summarized in Table 8 (Collberg and others, 2004:738).

3. Functional Composition and Entropy

In [9], McDonald and Yasinsac suggest three broad application categories of small or bounded input-size programs that motivate our obfuscation approach: sensor nets, positioning devices and financial programs. All of the above applications may perform mathematically intensive calculations, operate on small inputs (e.g. 16-bits) or deploy in hostile operating environments.
Our approach attempts to provide the core framework in developing secure code by using the concept of function tables. In [10], it is postulated that every deterministic algorithm produces one function table thus represents an atomic function. Furthermore, functional composition *(f o g)* or *f(g(x))* is also an atomic operation which is the main security implication of using functional tables. Because the composition of two atomic functions is also an atomic function, it is not possible to find a seam between the two composed functions or any implementation details within either of the composed functions. In addition, table lookups for all atomic functions are identical protecting information leakage from dynamic analysis of control flow. These facts are important because [2] specifically proposes that a reverse engineer typically analyzes a program's data structure and control flow in order to gain understanding of any application.

3.1 Function Tables of Composite Functions

A generic function, is a function $f: \{0,1\}^x \rightarrow \{0,1\}^y$, which takes a binary input, *x*, and produces a binary output, *y*. A generic encryption function, *E*, is also a function that takes an input and generates an output. Only a few characteristics distinguish an encryption function from a generic function. First, encryption functions have the property where the input and output generated are the same size. Second, the relationship of input and output for a particular encryption is identified by an key, $\{0,1\}^k$. The relationship between *f, E, x, y* and *k* are shown in figure 2. Figure 2 also shows that a functional table can be produced by both the function and encryption function

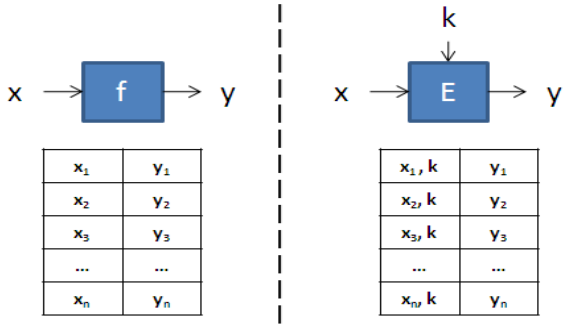

Figure 2. Generic function (left) and generic encryption function (right) diagram with corresponding function tables

We reiterate here that every deterministic function produces one function table. It is also noted here that an infinite amount of functions can produce the same function table. Therefore, a function table is a black-box representation of the function because the white-box details, exactly how the function is implemented, cannot be ascertained from the function table alone.
Figure 3 is an illustration of a composite function of *f* and *g* and their respective function tables. Composition on two functions is possible if the output of the first function is a subset of the input of the second function. It can be seen that the function table of the atomic composite function masks function *f* and *g*'s individual input/output relationship. Additionally, the two functions are inseparable from the composite table because there is no obvious seam joining the two functions. Furthermore, the composite table uses the same lookup table operation as a single function table making composite tables indistinguishable from a single function table from an operational perspective.

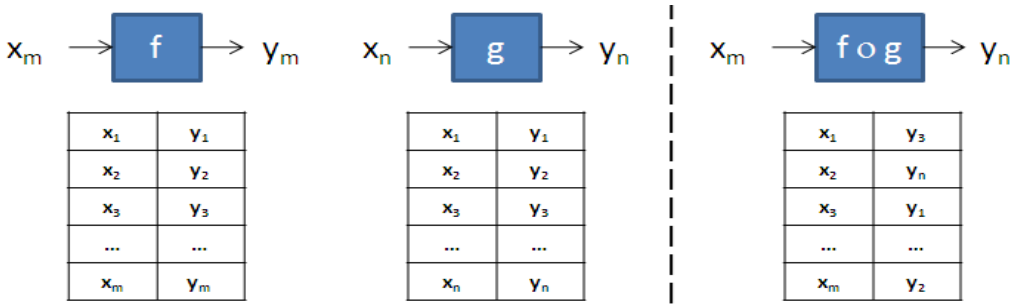

Figure 3. Function composition of *f* and *g* where y_m is a subset of x_n

3.2 Encryption Function

The atomic properties of the composite function table are the fundamental basis for our approach. Because the behaviors of the functions in the composed function are hidden, it is possible to embed a key into a symmetric key encryption without fear that the shared key used will be leaked.

Encryption is essentially a recoverable semantic translation of some input. As a base case example, [10] uses an one-bit input, one-bit output function to illustrate the functional table approach. There are exactly four semantic transformations, or behaviors, available to a function that operates on one bit; these are listed in table 2. The third and fourth semantic transformations, however, unsuitable for use as an encryption function because they produce irrecoverable output.

Semantic Transformations	Sample Input	Sample Output
1. Preserve the input	0, 1, 1, 0	0, 1, 1, 0
2. Flip the input	0, 1, 1, 0	1, 0, 0, 1
3. Flip 1's, preserve 0's	0, 1, 1, 0	0, 0, 0, 0
4. Flip 0's, preserve 1's	0, 1, 1, 0	1, 1, 1, 1

Table 2. List of semantic transformations and sample input and output

The first and second semantic transformation is the best obfuscation possible for this trivial one-bit case because the adversary has a random chance of guessing whether the first or second transformation was used. It is possible, as in [10], to see how this can extrapolated to multiple and stronger bits of encryption. Popular encryptions, such as DES and AES, are recoverable semantic transformations whose behavior and recoverability is identified through the key and mode of operation. Electronic code book (ECB) is used due to the necessity in enumerating input/output pairs for the encryption function table. Security implications of this design decision are discussed in section 4.

A function table that is a composition of a generic function, *f*, with an encryption function, *E*, hides:

1. Input/output relationship of *f*
2. Input/output relationship of *E*
3. Key embedded in *E*
4. Seam between *f* and *E*

3.3 Decryption Function

Until the encrypted composite function output is decrypted, it is protected but un-useable because the desired output is strictly that of the function composed with the encryption. The remapping of input/output is inconsistent with the theoretical obfuscation model property of function preservation. However, this also means the impossibility results in [1] does not apply to the composite functional table approach. In addition, the developer is provided some flexibility in deployment of a protected function through placement of the decryption function.

3.3.1 Partial Client-Server Deployment

If bandwidth is not a concern, then we may possibly release only the secured function and require the remote application to send the computed and encrypted information back to a trusted source for decryption. The information can then be sent back to the remote application if necessary.

Figure 4 illustrates how this method is distinct from the classical client-server model. In the classical client-server model, the protection is achieved by removing all sensitive calculations from the remote application and running them only within a trusted environment placing the burden of computation expense completely on the server side.

In this configuration, a standard decryption that corresponds to the embedded key encryption can be used because the key does not need to be embedded into the decryption function. Any padding to the input or output can also be handled on the trusted client-side. Thus, the remote application takes

input and passes an output from the composed function table back to the server. The decrypted output can either be used directly by the server or sent back to the remote application as needed

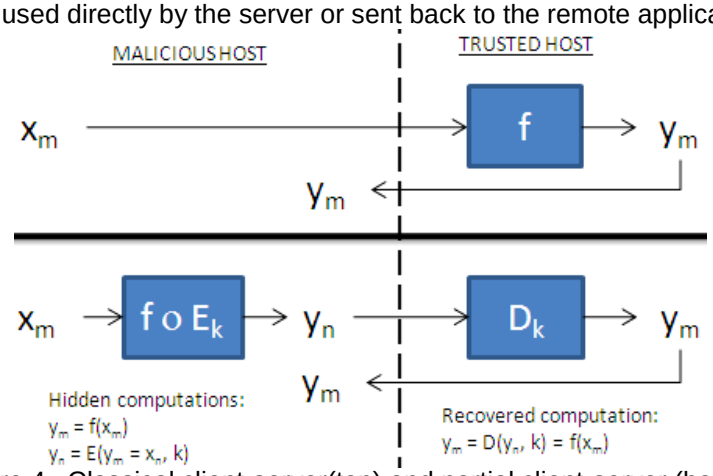

Figure 4. Classical client-server(top) and partial client-server (bottom)

The partial client-server model shares a weakness with the classical model; both have bandwidth requirements that may become a bottleneck. However, this deployment does illustrate how it is potentially possible to securely offload intensive process to the remote applications.

Use normal capitalisation within the text and do not use **bold face** for emphasis. Italics are acceptable. All headings should use initial capitals only, excepting for use of Acronyms. Please avoid the use of footnotes. Endnotes are not permitted and papers containing them will be returned.

3.1 Title and authors

In the paper title only, initial letters of all words of 4 or more letters should be capitalised.

Multiple authors from the same institution should appear as detailed at the start of this document. Multiple authors from different institutions should appear as :

Andrew Nonymous[1], Second Author[2] and Third Author[1]
[1]The department, faculty and name of institute, Town, Country
[2]The department, faculty and name of institute, Town, Country

Give first and last name, in that order. Do not use all caps. Email addresses should be given beneath, one per line and in the same order as the authors are listed.

All author details will be removed by us before the review process.

3.1.1 Sub-sections (Style Heading 3)

You may use up to three levels of heading, as illustrated in this document.

Do not use any further levels of heading.

4. Experimental Results

You are invited to use figures and tables in your paper wherever they will help to illustrate your text. The proceedings are delivered to conference participants in electronic format and therefore support colour figures, however, the book version is printed in black and white and therefore you are advised to refrain from using colours to deliver important information in your figures.

Quantitative Analysis of Black-box Data.

We analyze the sets of 1,000 randomly generated BENCH circuits with the statistical tests listed in Table 7. We use the input/output sizes of the benchmark programs listed in Table 6 as parameters for the RPG. Because the impact of the internal structure is currently unknown, we set the parameter

for the number of intermediate nodes to 100, 300 and 500. For the circuit c17, we generated an additional random program set with six intermediate nodes to match the original circuit description.

First, we conduct an analysis on the collective random function output. Each function produces an output signature which is the output sequence of the function based on an input sequence. The total possible number of unique signatures is $(2^{out})^{(2^{in})}$ where *out* is the number of output bits and *in* is the number of input bits. We check the output signatures of the random function sets for uniqueness using a CRC32 checksum.

Numbers within sets of identical output signature are an indicator of functional equivalency and structural diversity. For a set of randomly generated programs to produce large sets of non-unique output signatures, it may be a signature that exhibits weak correlation between structural pattern and output signature. If we intend to obfuscation white-box information by emulating randomly constructed circuits, then signatures with a large number of candidate structures are good candidates for obfuscation. In practical terms, it means that we can swap the structure of one member within the set with another member in the same set. This obfuscates the original structure because we produced the alternate structural logic randomly without any knowledge of the original structure and therefore the replacement structure cannot leak information about the original structure.

Random function sets of 5-2-6 and 5-2-100 yielded 125 and 71 functions that produced non-unique output signatures respectively presented in Table 9. The other random function sets did not produce any duplicate output signatures.

Table 3. Non-unique Output Signature Characteristics of 1000 Random Functions

5-2-6		5-2-100	
Set size of identical output signatures	**Number of Sets**	**Set size of identical output signatures**	**Number of Sets**
2	**32**	**2**	**7**
3	**2**	**11**	**1**
4	**8**	**12**	**1**
5	**1**	**16**	**1**
9	**2**	**18**	**1**

We expected fully unique signatures for even the small input and output parameters because 4^{32} unique signatures are possible. For a set of 1,000 random functions to exhibit signature collisions may indicate that structural diversity is great for smaller input/output parameters. We observe that the intermediate node is a factor in producing signature collisions. Increasing intermediate node size causes a drop in collision frequency but an increase in collision concentration where the chance of collision is less likely, but in the case of collision, the collision set tends to be greater in size. We graph our observations regarding intermediate node size and signature collisions in Therefore, obfuscation of a complete white-box structure may be more effective with partial obfuscations of smaller input/output size with a large intermediate node size so there are several candidates for replacement.

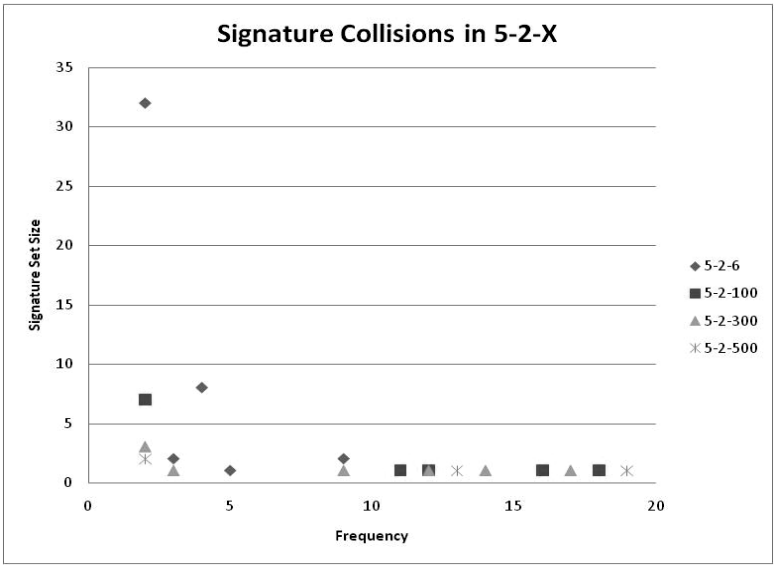

Figure 4. Signature Collisions in 5-2-X

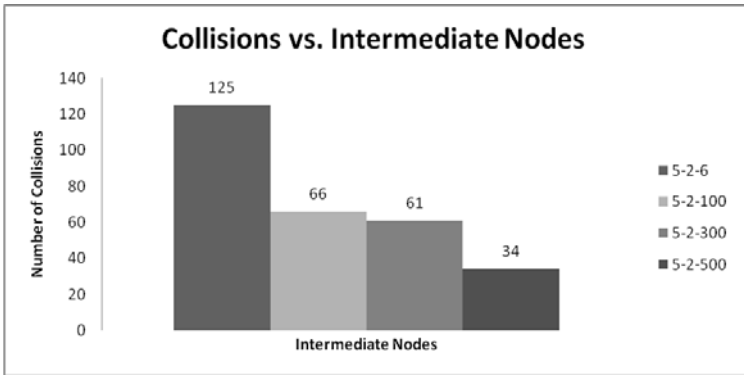

Figure 5. Signature Collision to Intermediate Node Size

We then perform analysis on each individual benchmark program and corresponding function set. For clarity, we display only the gray code input sequence in the following tables. It is important to note that metrics on run lengths and excursion states are dependent on input sequence. In addition, using gray code input provides the avalanche metric for comparison between the benchmark output and random function output. We recognize that there are many there are many possible sequences that exist where we flip only one input bit. We use the gray code as an exploratory technique to observe the avalanche affect of input bits; the avalanche effect on output bits for cryptographic ciphers should be observable using gray code input. We verify by using a black-box analysis of the output from 1,000 AES encryption output tables using a gray code input sequence. Tables illustrating the results of the statistical analysis comparing benchmark and respective input/output size random functions are found in the Appendix; the result of each test by output bit is provided so that the distinction between benchmark and random functions can be visualized. We use averaging across the 1,000 random functions on each output bit to provide a result. The experiments provide a picture of the expected values of the seven statistical tests for a randomly generated program of a certain input/output size. From the results of this experiment, it appears that random functions generate consistent results for each output bit across all tests which can be contrasted against the output bit behavior in the benchmark functions.

In Figure 17, Figure 18, and Figure 19, we graph the standard deviation for all bits in the output by test for some of the benchmark functions and their respective random program set. For these graphs, we included the binary counter sequence. We observe disparity in results; random program sets produce significantly less diversity in their output bits than the benchmarks as shown by the flat lines

generated by the random program sets in the three figures. We note that our two input sequences produced similar results.

Within this limited set of benchmarks, it appears that the number of excursion states is the biggest indicator of an unprotected benchmark function versus the set of random functions while the number of zero cycles tends to be a poor indicator. In addition, this black-box analysis on deviation from expected randomness values lets us know which statistical test best isolates non-random behaving bits in the output. We can then target the control flow of the bits that do not exhibit random behavior with structural randomness. This information is useful in cases where we cannot use black-box protection and the security must rely only on white-box structural entropy.

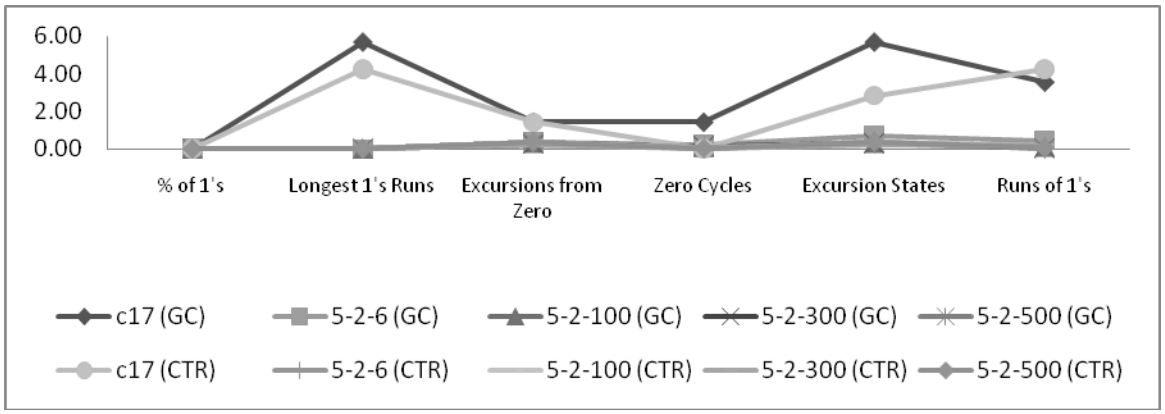

Figure 6. Standard Deviations of All C17 Output Bits by Metric

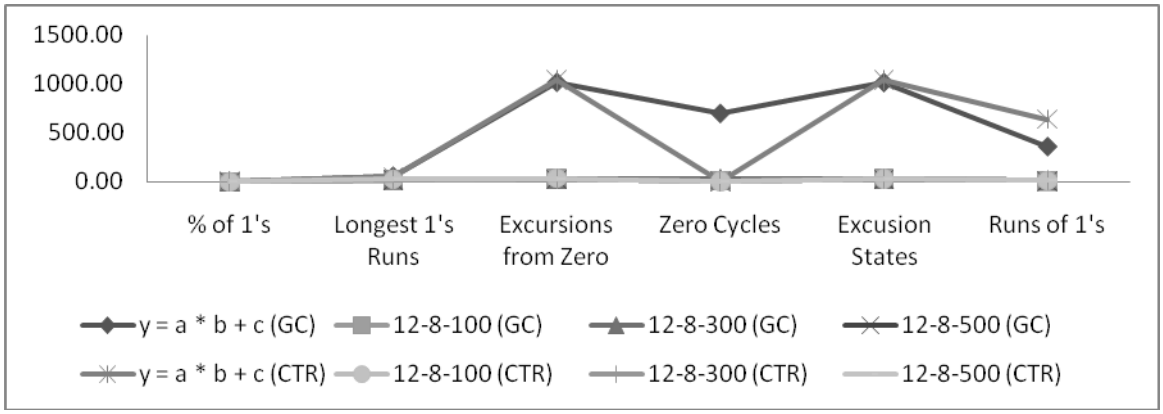

Figure 7. Standard Deviations of All *y = a * b + c* Output Bits by Metric

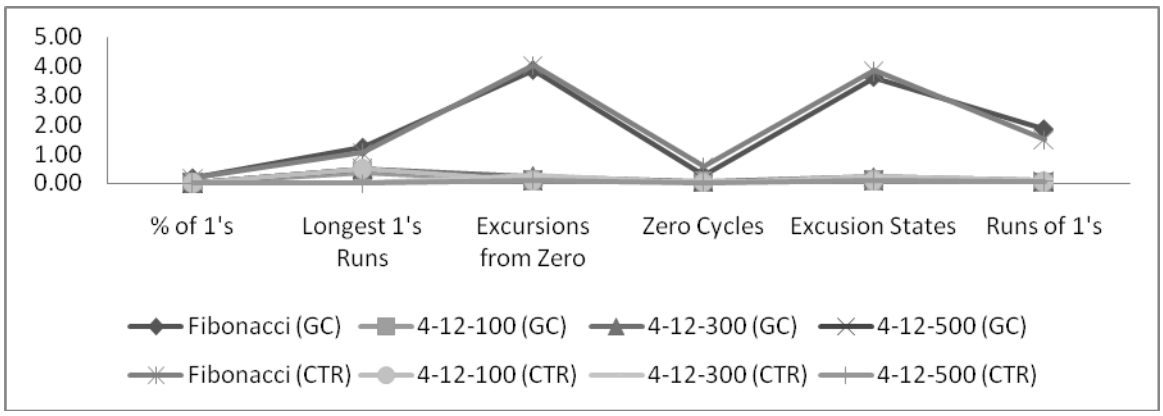

Figure 8. Standard Deviations of All Fibonacci Output Bits by Metric

We conducted a statistical analysis of AES encryption with 1,000 keys and equal input size of five bits to examine the feasibility of protecting a c17 circuit from the ISCAS-85 circuit library with AES. The standard deviations between AES and the random program set for each metric, shown in Figure 20. Standard Deviations of All AES Output Bits by Metric was significantly closer to zero than any other experimental function.

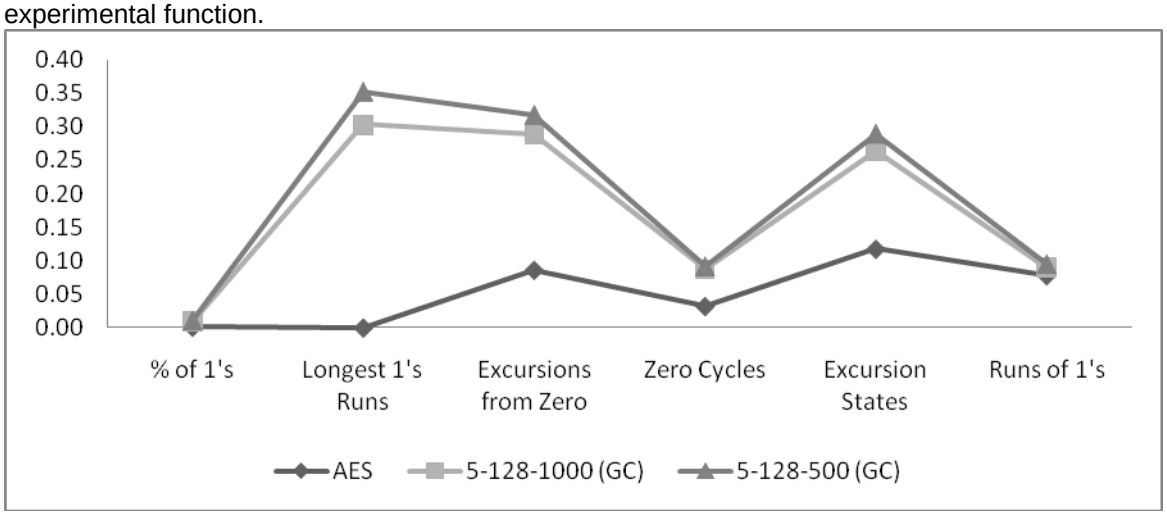

Figure 9. Standard Deviations of All AES Output Bits by Metric

The averages and standard deviations can also be found in Table 10; per bit graphs are not included because it is difficult to clearly represent all 128-bits graphically. We note that we adjusted the random program set parameter from 100 and 300 to 500 and 1000 in order to accommodate the significantly larger output size in AES. Different results between the AES and random program set produce approximately the same results. The metrics provided by these random sets are valuable because the results for these metrics are unknown for random program structure. Thus, these metrics provide a comparison point for functions that may have the parameters such as input size, output size, or intermediary node size.

Table 4. Statistical Results of AES and a Random Program Set

Function	% of 1's	Longest 1's Runs	Excur. from Zero	Zero Cycles	Excur. States	Runs of 1's
AES avg	0.50	4.00	6.62	0.69	8.96	8.25
AES std dev	0.00	0.00	0.09	0.03	0.12	0.08
5-128-500 avg	0.50	8.05	18.34	1.23	19.51	4.55
5-128-500 std dev	0.01	0.35	0.32	0.09	0.29	0.09
5-128-1000 avg	0.50	7.95	18.35	1.12	19.51	4.60
5-128-1000 std dev	0.01	0.30	0.29	0.09	0.26	0.09

We note that the metrics did not change significantly between the 500 and 1000 internal node set or random functions indicating that intermediate node size may not be a significant factor on the randomness of individual output bits. This was also true for the benchmark programs even though we did produce a small percentage of signature collisions in the 5-2-X set of experiments. Standard deviations also remained small though we note that the standard deviations of the two random set in our 5-2-X with AES experiments mirrored each other which could indicate that our RPG construction is a factor. No functions within the two 5-128-X sets shared the same output signature.

In addition, the test verified that the 1,000 AES keys produced 50% approximate entropy on the output as expected when we use gray code input. We note that the unprotected benchmark functions on average produce only 26% approximate entropy. This means that a change in a single input bit has significantly less impact, or more specifically, less of an avalanche effect on the output bits of randomly generated circuits. Therefore, our results indicate that structural entropy alone does not, on average, produce the same black-box entropy as cryptographic functions. We are interested in the

approximate entropy results specifically because the greater entropy tends to hinder black-box analysis. We graph our results in Figure 21. The first column is our verification of approximate entropy on AES, followed by the approximate entropy observed in our randomly generated sets. We obtained the fourth column results by using an AES encryption table to protect the output of the c17 circuit. This did not increase approximate entropy because ECB does not hide output patterns. We achieved approximate entropy results similar to AES when we applied two different padding schemes to diffuse the output space prior to applying the AES encryption, as shown in the last two columns.

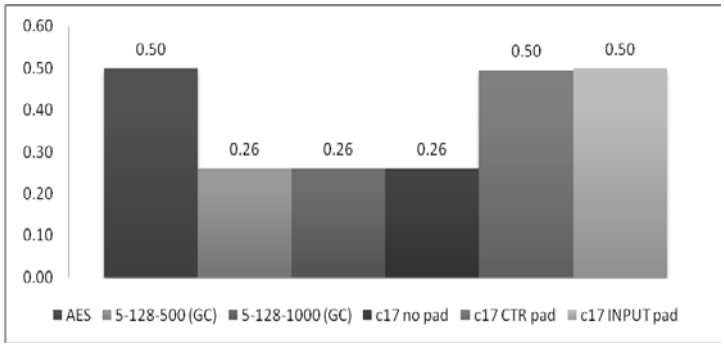

Figure 10. Approximate Entropy of AES and 5-128-X

Analysis of Side-Channel Data.

CFT is not fixed to an implementation because the security concept is to prevent adversary analysis by flattening of functional structure to two-dimensions. Because this research implemented the CFT using text files, the protected programs took longer to run due to frequent file accesses; the disk accesses incurred cost penalty in performance time because disk access operations are slower than the operations in the benchmark programs which did not require significant processing power.

In complexity terms, a lookup operation in the encryption table is constant time, O(1) making CFT very scalable. Constant time is achievable because every entry is the same size and we can provide the entries, input order sorted, so that an index search is possible. We can use the original cryptographic primitive to decrypt and recover the output and we know that the cryptographic primitive runs in polynomial time. If we use the function table for decryption, we could first apply sorting to the output table and then use a binary search to achieve O($n$log$n$) performance. We cannot use the same indexing method as the encryption table because the ciphertexts sparsely populate too large a range.

We found the file sizes consistent to our estimates of $2^n * m$ bits where n is the number of input bits and m is the number of output bits. We note that a side effect in our implementation under the NTFS file system test environment is that Windows file explorer reports a difference between the actual file size and the size the file takes on the disk.
For BES representations of CFTs, we found early in our experiments that storing the BES as a file take much more memory space than the CFT in our implementation. For a BES, we cannot estimate the length or the number of prime implicate for each output bit. However, we are attempting to achieve random output so we expect each output bit to produce significantly long Boolean equations making textual representation very inefficient. We do not propose BES implementation as a text file; we generate it as a blueprint for a minimized sum-of-products two-dimensional gate structure that can be then implemented as code. We implement BES textually mainly to examine this structure generation for future experimentation. In terms of performance, BES runs with complexity O(n) where n is the number of output bits because each output bit has its own Boolean equation that runs in constant time.

The research shows that random programs can be a comparison tool for intent protected obfuscation techniques such as CFT. While there is yet to be a set of agreed upon metrics to compare program structure, there are metrics in use that analyzes function output. The results shown in this chapter show that programs with randomly generated structure produce randomness across the output bits. The randomness closely equals that of AES, a strong encryption algorithm. In the same way that functional randomness produces output that is hard to discern a pattern, structural randomness may

produce program structure that is difficult to analyze. Thus, if it becomes possible to accurately assess structural randomness, it will be possible for an obfuscation to be intent protected by creating an obfuscated version of a function that is both structurally random and functionally random. In the absence of such metrics, this research uses CFT with symmetric encryption to remove the structural details of a program while creating measurably random output as an obfuscation technique.

5. In conclusion

Our work demonstrates the generality and efficiency of the CFT approach using simple Java programs and deterministic functions implemented as combinational logic circuits. We also use such benchmark programs to consider the nature of structural randomization induced by obfuscating algorithms based on iterative selection and replacement strategies. As a contribution, we consider the correlation between structural (programmatic/syntactic) randomness versus functional randomness. We report positive results on the efficacy of our approach to induce statistical properties by analyzing whether structural randomization produces entropy in the output bits of protected programs. The approach shows promise to add yet another layer of protection against potential adversarial reverse engineers.

References

McDonald, J. and Yasinsac A. (2007) "Applications for Provably Secure Intent Protection with Bounded Input-Size Programs", *The Second International Conference on Availability, Reliability and Security, 2007. ARES 2007,* pp.286-293, 10-13 April 2007.

McDonald, J., Kim, Y, and Yasinsac, A. (2008) “Software Issues in Digital Forensics”, *ACM SIGOPS OS Review*, Vol. 42, No. 3, April 2008.

Yasinsac, A. and McDonald, J. (2008) “Tamper Resistant Software through Intent Protection”, *International Journal of Network Security*, Vol. 7, No. 3, pp. 370–382.

Linn, A. (2008) Software Obfuscation with Symmetric Cryptography.

Collberg, Christian S., Clark Thomborson and Douglas Low. “A Taxonomy of Obfuscating Transforms,” Technical Report 148, Department of Computer Science, University of Auckland, 148: 1-36 (July 1997).

Collberg, Christian S. and Clark Thomborson. “Watermarking, Tamper-Proofing, and Obfuscation—Tools for Software Protection,” IEEE Transactions on Software Engineering, 28.8: 735-746 (August 2002).

Goldwasser, Shafi, and Guy Rothblum. “On Best-Possible Obfuscation,” To appear in The Fourth Theory of Cryptography Conference (TCC 2007), Trippenhuis, Amsterdam, The Netherlands, 21-24 February 2007.

Eilam, Eldad. *Reversing: Secrets of Reverse Engineering.* Indianapolis: Wiley Publishing, 2005.

Yasinsac, Alec and J. Todd McDonald, “Towards Working With Small Atomic Functions”, the Fifteenth International Workshop on Security Protocols, Brno, Czech Republic, 18-20 April 2007, LNCS (to appear).