Obfuscation Framework Based on Functionally Equivalent Combinatorial Logic Families

THESIS

Moses C. James, Captain, USAF

AFIT/GCS/ENG/08-12

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expresses in this thesis are those of the author and do not reflect the official policy of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GCS/ENG/08-12

Obfuscation Framework Based on Functionally Equivalent Combinatorial Logic Families

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
In Partial Fulfillment of the Requirements for the
Degree of Master of Science (Computer Science)

Moses C. James, B.S.
Captain, USAF

March 2008

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/GCS/ENG/08-12

Obfuscation Framework Based on Functionally Equivalent Combinatorial Logic Families

Moses C. James, B.S.

Captain, USAF

Approved:

/signed/	22 Feb 2008
Lt Col J.T. McDonald, PhD (Chairman)	Date
/signed/	22 Feb 2008
Lt Col S.H. Kurkorski, PhD (Member)	Date
/signed/	22 Feb 2008
Dr. H.B. Potoczny (Member)	Date

AFIT/GCS/ENG/08-12

Abstract

This thesis aims to be a few building blocks in the bridge between theoretical and practical software obfuscation that researchers will one day construct. We provide a method for random uniform selection of circuits based on a functional signature and specific construction specifiers.

Additionally, this thesis includes the first formal definition of an algorithm that performs only static analysis on a program; that is analysis that does not rely on the input and output behavior of the analyzed program. This is analogous to some techniques used in real-world software reverse engineering.

Finally, this thesis uses the equivalent circuit library to empirically produce some statistical data about enumerated circuit families and explains how this data may be useful to future researchers.

Table of Contents

Page

Abstract . . . iv

List of Figures . . . vii

I. Introduction . . . 1
1.1 Scope . . . 2
1.2 Organization . . . 2
1.3 Definition of Terms . . . 3

II. Background . . . 4
2.1 Virtual Black Box Obfuscation . . . 4
2.1.1 Virtual Black Box Obfuscation Applications . . . 5
2.2 Indistinguishability Obfuscation . . . 6
2.3 Best-Possible Obfuscation . . . 7
2.4 Intent Protection . . . 7
2.5 Program Encryption Obfuscation . . . 9
2.5.1 Program Encryption Applications . . . 9
2.6 Program Encryption Obfuscator Development . . . 11
2.7 Black Box Protection . . . 12
2.8 Characterizing White Box Protection . . . 12

III. Methodology for Circuit Enumeration and Random Selection . . . 14
3.0.1 Bounding the obfuscation question . . . 14
3.0.2 Circuit set analysis . . . 14
3.0.3 Implementation . . . 15
3.1 Sets of circuits . . . 15
3.1.1 Graphs . . . 15
3.1.2 A more complex notion . . . 18
3.2 An algorithm framework . . . 19
3.3 Refining the algorithm framework . . . 21
3.3.1 `SymmetricGates` . . . 21
3.3.2 `RedundantGates` . . . 21
3.3.3 `AllowConstants` . . . 21
3.3.4 `DoubleInputs` . . . 22
3.3.5 `ExactCount` . . . 22
3.3.6 `SimpleOutputs` . . . 22
3.3.7 A refined algorithm . . . 22

Page

IV. Analysis of Functionally Equivalent Logic Families 24
4.1 Library size 24
4.1.1 Fitting the data to a curve 25
4.2 Output entropy 25
4.3 Structural metrics 29
4.4 jCXL performance analysis 30

V. Advancing Some Theoretical Models of Obfuscation 33
5.1 Motivation 33
5.2 Intent Protection Weakness 33
5.3 Program encryption and random programs 34
5.4 Set selection obfuscation 37
5.4.1 Comparison with previous definitions 37
5.4.2 Impossibility result 39
5.5 Structural indistinguishability 39
5.5.1 Flawed notions of structural indistinguishability 40
5.5.2 A better notion of structural indistinguishability 41
5.6 Conclusion 41

VI. Conclusions 43
6.1 Future work 43

Bibliography 46

Appendix A. Software Development 48
A.1 Analysis of Legacy Software 48
A.2 Initial design 49
A.2.1 Circuits 49
A.2.2 Circuit enumerator 52
A.3 User interface 54
A.4 Development of persistence layer 54
A.5 Development of machine interface 55
A.6 Refinement of algorithm runtime 56
A.7 Software Testing 57

List of Figures

Figure		Page
2.1	Program encryption visualization. Note that C' is intent protected.	10
2.2	Full intent protection	13
3.1	A high-level visualization of a sub-circuit selection/replacement obfuscation algorithm	15
3.2	An example of a sub-circuit selection and replacement	16
3.3	Obfuscating transformations as set selection operations	17
3.4	Implementation domain model	17
3.5	An example of a gate where both inputs have the same origin	19
3.6	An example of a circuit containing two gates with the same external signature	19
3.7	A circuit with two open outputs	20
3.8	An example of a one-output circuit that is not enumerable unless the algorithm allows intermediate circuits with an illegal number of open outputs	21
4.1	Count of 3-1-X-Nand circuits with various properties	26
4.2	Count of 4-1-X-Nand circuits with various properties	26
4.3	Count of 3-1-X-And+Or+Nand+Nor+Xor+Xnor circuits with various properties	27
4.4	Black-box entropy distribution for some of the 3-1-X-And+Or+-Nand+Nor+Nxor+Xor-F-F-F-T-T classes of circuits	28
4.5	An example truth table of a function with maximal black-box entropy but other undesirable properties	29
4.6	Average fanout distribution for some of the 3-1-4-Nand classes of circuits	30
4.7	Relationship between library size × number of gates and disk usage with linear regression	31

Figure | Page

4.8 Relationship between library size × number of gates and time to persist with linear regression 32

5.1 A visualization of a uniform random set selection 37

5.2 A structural distinguisher 42

A.1 Software use case 49

A.2 Domain model 50

A.3 Legacy CXL Class Diagram 51

A.4 Portion of legacy source code 52

A.5 The initial circuit class diagram 53

A.6 Circuit enumerator class diagram 53

A.7 Part of the output of PrintCircuits running in unfiltered mode for a 3 input, 2 output, maximum of 4 gate circuit 54

A.8 Part of the output of `PrintCircuits` running in filtered mode for a 3 input, 2 output, maximum of 4 gate circuit 55

A.9 The persistent circuit chooser and library manager 55

A.10 Developer-provided benchmarks for H2 56

A.11 The more efficient circuit class diagram 57

Obfuscation Framework Based on Functionally Equivalent Combinatorial Logic Families

I. Introduction

As technological progress marches on, the protection of software in critical military applications becomes more and more important. And yet at the same time, United States policy strongly encourages the sale and transfer of some military equipment to foreign governments and makes it easier for potential adversaries to get access to this critical software [10]. How can we stop a potential adversary with access to our software from effectively using it against us?

This line of reasoning always leads to the critical question, "How can you protect a piece of software from the computer running it?" The answer to this question has applications in many fields including cryptography, mobile agent security, and software protection. Conventional wisdom says that this type of protection (or *obfuscation*) is impossible and anecdotal evidence from professional software reverse engineers seems to support this statement. In fact, Barak, et al. [1] formalize the notion of a "Virtual Black Box" and then prove that it is impossible to construct a general, efficient obfuscator using this model.

The results of Barak, et al. are not as catastrophic as they might appear. We derive this impossibility result by constructing a family of unobfuscatable functions that produce useful output to a potential adversary through the introduction of self-reference. One might attempt to approach the problem by formalizing a method of excluding self-referential functions from consideration. However, we will not travel this path as it is fraught with peril of Gödelian proportions. Perhaps a more sane way of attempting to avoid the consequences of this result is to focus on the program's usable output.

The crux of the *program encryption* obfuscation paradigm [21] is that we perform a two-phase obfuscation. In the first phase we encrypt the output of program **P** in such a way that the output of the program appears to be that of a random oracle. We refer to this construction as **P+E**. In the second, we protect the structure of the program from intentioned manipulation by an adversary.

This thesis explores the creation of a circuit obfuscator for use in this second phase as described in [15] that researchers may also use to construct useful circuit library metrics. A secondary objective of this thesis is the detailed formalization of some new models of obfuscation. Both of these objectives serve to further the study of theoretical obfuscation.

1.1 Scope

The scope takes a two-fold approach to the obfuscation question, one from the practical direction and one from the theoretical direction. The intent of this thesis is not to fully bridge the gap between the two but one can think of it as a few pieces of building material that may be used in the construction of this bridge. First, this thesis details and justifies an algorithm designed to enumerate all possible circuits in a given set. Then, it provides and justifies several new ways of looking at obfuscation models developed by others. Definitively answering the question posed at the beginning of this chapter is the stuff of legends and is far beyond the scope of this thesis.

1.2 Organization

We organize this thesis as follows. Chapter II provides a detailed background on obfuscation formalization. Chapter III describes the creation of an algorithm designed to fully enumerate a specific family of functions (combinational circuits). Chapter V describes several novel formal descriptions of obfuscation and some results based on those models. Finally, Chapter VI and summarizes the scientific contributions of this thesis.

1.3 Definition of Terms

We use the notation of [1] and [6].

(PPT) Probabilistic polynomial-time Turing machine

(Circuit) A Boolean circuit composed of binary logic gates

(Program) Either a circuit or a TM

($A^O(x)$) The execution of algorithm A on input x with oracle access to oracle O.

($\langle M \rangle$) The function computed by TM M given as $M(t, x)$. $\perp$ if $M(x)$ does not terminate in t steps.

(Negligible function) A function $\mu : \mathbb{N} \rightarrow \mathbb{R}$ such that for every positive polynomial $p(\cdot)$ there exists an N such that for all $n > N$,

$$\mu(n) < \frac{1}{p(n)}$$

II. Background

Until recently, the study of software obfuscation has remained in the the realm of so-called "fuzzy security" [2], that is security with poorly defined objectives. While there are a number of commercially available products that make claims about obfuscation capabilities, none have attempted to quantify the strength of their products. However, theoretical researchers have developed a few models to better quantify exactly what it means to obfuscate a piece of software.

2.1 Virtual Black Box Obfuscation

Virtual black box is a formalization of perhaps the most intuitive idea of software obfuscation, that of indistinguishability between an obfuscated program and a "black box" that performs the same function as the program. [1] describes virtual black box obfuscation for TMs and circuits as follows:

Definition II.1. *A probabilistic algorithm O is a* [virtual black box] TM obfuscator *if the following three conditions hold:*

- *(functionality) For every TM M, the string O(M) describes a TM that computes the same function as M.*
- *(polynomial slowdown) The description length and running time of O(M) are at most polynomial larger than that of M. That is, there is a polynomial p such that for every TM M,* $|O(M)| \leq p(|M|)$*, and if M halts in t steps on some input x, then O(M) halts within p(t) steps on x.*
- *("virtual black box" property) For any PPT A, there is a PPT S and a negligible function* α *such that for all TMs M*

$$\left|\Pr\left[A(O(M)) = 1\right] - \Pr\left[S^{\langle M \rangle}(1^{|M|}) = 1\right]\right| \leq \alpha(|M|)$$

We say that O is efficient *if it runs in polynomial time.*

Definition II.2. *A probabilistic algorithm O is a* [virtual black box] circuit obfuscator *if the following three conditions hold:*

- *(functionality) For every circuit C, the string O(C) describes a circuit that computes the same function as C.*
- *(polynomial slowdown) There is a polynomial p such that for every circuit C,* $|O(C)| \leq p(|C|)$
- *("virtual black box" property) For any PPT A, there is a PPT S and a negligible function α such that for all circuits C*

$$\left|\Pr\left[A(O(C)) = 1\right] - \Pr\left[S^C(1^{|C|}) = 1\right]\right| \leq \alpha(|C|)$$

We say that O is efficient *if it runs in polynomial time.*

These definitions say that an algorithm O is an obfuscator if for every PPT adversary A with access to the text of an obfuscated circuit/TM there exists another PPT S with only oracle access to an obfuscated circuit/TM such that for any obfuscated circuit/TM the probability of S returning a different result than A is negligible. Informally, that is to say that anything A can do, S can also do.

2.1.1 Virtual Black Box Obfuscation Applications. Barak, *et. al* [1] note several possible applications for a general program obfuscator. Two of the most obvious ones are:

- (Software Protection) A virtual black box would allow an untrusted host to use a proprietary algorithm without the possibility of reverse engineering. This would also allow software developers to securely protect any number of software-based copy-protection schemes that are currently easily bypassed.
- (Public-key Cryptography) If we could apply a virtual black box to a secret-key cryptography system we would essentially be converting it to a public key

cryptography system because no adversary would be able to access the hidden secret key.

Unfortunately, [1] proves that it is impossible to construct a general obfuscator of this type by introducing a family of TMs and a family of circuits that are inherently unobfuscatable. While this does not mean that these applications are impossible to achieve, it does prove that there is no general way of doing so. We can compare this to the halting problem where although there is no general way of determining if a program will halt, it is quite simple to determine whether many specific programs will. As an alternative to virtual black box obfuscation (for circuits), [1] propose the idea of indistinguishability obfuscation.

2.2 Indistinguishability Obfuscation

Definition II.3 (indistinguishability obfuscator). *We define an* indistinguishability obfuscator *in the same way as a circuit obfuscator, except that we replace the "virtual black box" property with the following:*

- *(indistinguishability) For any PPT A, there is a negligible function α such that for any two circuits C_1, C_2 which compute the same function and are of the same size k,*

$$|\Pr[A(O(C_1))] - \Pr[A(O(C_2))]| \leq \alpha(k)$$

That is to say, the obfuscations of any circuits that compute the same function are not distinguishable in polynomial time. [1] note that inefficient indistinguishability obfuscators exist ("Let $O(C)$ be the lexicographically first circuit of size $|C|$ that computes the same function as C") and postulate that an indistinguishability obfuscator is "as good" as any other obfuscator. [8] clarify the notion that an obfuscator is "as good" as any other obfuscator with the notion of *best-possible obfuscation* wherein a program may leak some non-black-box information but no more than any other program that performs the same function.

2.3 Best-Possible Obfuscation

Definition II.4 (best-possible obfuscation). *An algorithm O, which takes as input a circuit in C and outputs a new circuit, is said to be a* (computationally/statistically/perfectly) best-possible obfuscator for the family C*, if it has the preserving functionality and polynomial slowdown properties and also has the following property (*instead *of the virtual black-box property).*

- *Computational/Statistical/Perfect Best-Possible Obfuscation. For all large enough input lengths, for any polynomial size circuit adversary A, there exists a polynomial size simulator circuit S such that for any circuit $C_1 \in C_n$ and for any circuit* $C_2 \in C_n$ that computes the same function as C_1 *and such that* $|C_1| = |C_2|$, *the two distributions $A(O(C_1))$ and $S(C_2)$ are (respectively) computationally/statistically/perfectly indistinguishable.*

[8] prove that all best-possible obfuscators are also indistinguishability obfuscators and *efficient* best-possible obfuscator is equivalent to an efficient indistinguishability obfuscator. Unfortunately, there are no obvious applications for the best-possible or indistinguishability obfuscator paradigms.

2.4 Intent Protection

McDonald and Yasinsac introduce intent protection in [15] and further elaborate in [13, 21]. We do not refer to this as an obfuscation paradigm because it does not necessarily imply the existence of an obfuscator. Instead, it poses two decision problems:

- Does a specified adversary *understand* a specified program?
- Is a specified program *intent protected*?

We define *understanding* and *intent protection* as follows:

Definition II.5. *A TM A* black box understands *TM P if the following condition holds when A has oracle access to P:*

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

Definition II.6. *A TM A* black box understands *circuit C if the following condition holds when A has oracle access to C:*

$$\Pr\left[A(y) = x, x \in X | C(x) = y\right] > |X|^{-1} + \epsilon$$

Definition II.7. *A TM A* white box understands *TM P if for A, given access to the string describing P:*

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

Definition II.8. *A TM A* white box understands *circuit C if for A, given access to the string describing C:*

$$\Pr\left[A(y) = x, x \in X | C(x) = y\right] > |X|^{-1} + \epsilon$$

In each definition, ϵ is a small constant. What this means, informally, is that for an adversary to understand a program, the adversary must be able to, given some level of access to the program, be able to determine, with a probability better than guessing, an input that will produce a specified output. We explore some weaknesses in this model and propose an alternative in Chapter V.

Definition II.9. *A circuit/TM as* black box intent protected/while box intent protected *if there does not exist a TM that* white box understands/black box understands *that circuit/TM.*

Under white box intent protection an adversary will be able to perform both static and dynamic analysis of the program. Under both white box and black box

intent protection, an adversary will be able to run the program any number of times (subject to a polynomial bound.) We categorize these types of attacks respectively as **white-box** and **black-box** attacks in order to show that we need to think of intent protection from both a black-box and a white-box perspective. McDonald and Yasinsac also propose a type of attack, based entirely upon the structure of a program, that is analogous to static attacks used when reverse engineering real software. We explore a way of formalizing this type of attack in Chapter V.

2.5 Program Encryption Obfuscation

Because the user of a program typically needs to have some level of understanding of the function of the program, intent protection is not a desirable property for many programs. However, one possible application would be a program that produces encrypted output. We conjecture the existence of an obfuscator defined as follows[1]:

Definition II.10. *A probabilistic algorithm T is a* program encryption circuit obfuscator *if the following three conditions hold:*

- *(functionality recoverability) There exists a probabilistic algorithm, R such that for every circuit C, every string k, and every x in the domain of C,* $R(T(C,k)(x),k) = C(x)$.
- *(polynomial size) There is a polynomial p such that for every circuit C and every string k,* $|T(C,k)| \leq p(|C|)$
- *("intent protection" property) For any circuit C and any k, $T(C,k)$ is intent protected.*

We say that T is efficient *if it runs in polynomial time.*

2.5.1 Program Encryption Applications. Clearly it would not make sense to program encrypt most pieces of software because it produces output that is, by design,

[1]Program encryption is not necessarily restricted only to circuits but since this thesis will not explore TM program encryption, we only use the circuit definition.

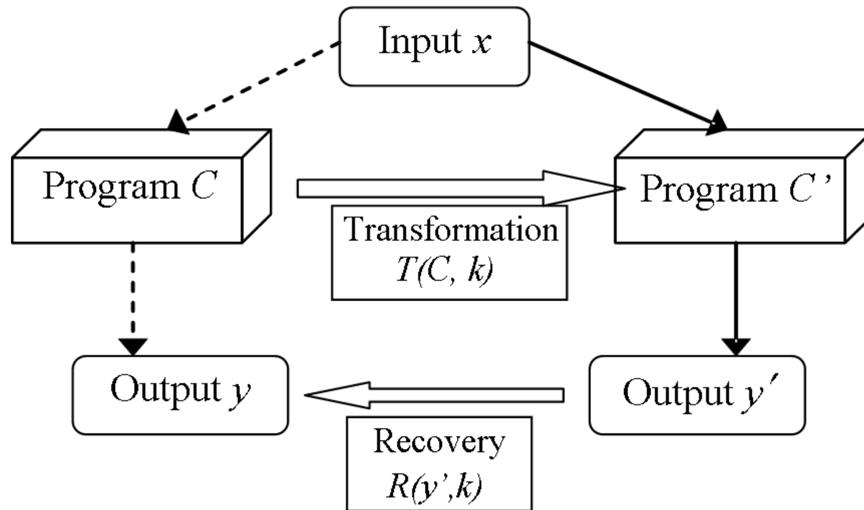

Figure 2.1: Program encryption visualization. Note that C' is intent protected.

not immediately useful to the program's user. However, there are a surprisingly large number of potential uses for this paradigm.

- (Public key cryptography) In the trivial case, an encrypted program functions as a public key cryptosystem. Simply create a program that outputs its input and use a program encryption obfuscator on it. By the definitions of intent protection and program encryption, an adversary would be unable to decrypt any output generated by this program.

- (Software/algorithm protection) In [18], Sander and Tschudin introduce function hiding, a concept similar to program encryption and imagine the following scenario:

  Suppose Alice devises an algorithm and wishes to allow Bob to use the algorithm without allowing Bob to understand how it works and also wants to restrict how many times Bob can use the algorithm. We also suppose that the computational intensity of the algorithm prevents Alice from merely running it for Bob.

  If Alice gives Bob a program encrypted version of the program, Bob can run the program as many times as he wants, but will only be able to use the outputs that Alice decrypts for him.

- (Secure utilization of distributed computing resources) Suppose that we have access to a large network of insecure computers and wish to use them to extract a private key from a public key via a brute-force attack. Program encryption would prevent an adversary with access to some or all of these machines (but not the machine decrypting the results) from discovering the key when it is cracked.
- (Spyware) Consider a piece of software designed to collect information about the machine on which it is running and transmit it over the internet. A spyware author could use program encryption to prevent a host machine from determining which information the malicious program is collecting[2].
- (Military applications) Suppose a system must perform data analysis in an unsecured location while relaying some information back to a central location.(A possible application here would be a program that performs image analysis from an attached video camera.) If an adversary physically compromises the system, program encryption obfuscation would prevent this adversary from determining anything about the program that cannot be inferred from the context of the system.

2.6 Program Encryption Obfuscator Development

If a general efficient program encryption obfuscator exists, it will at least need to satisfy the following properties: functionality recoverability, polynomial size, and intent protection.[3] To accomplish this, we divide the task into two stages. In the first stage, **black box protection**, we establish functionality recoverability and hard-to-invert. In the second stage, **white box protection**, we establish virtual black box.

[2] [3] has suggested a method for doing this using Private Information Retrieval. However, program encryption could reduce communication complexity for a single query from $O(\log(n))$ to $O(1)$.

[3] If we drop the efficiency and polynomial size requirements, McDonald and Yasinsac show that we can construct this type of obfuscator by applying the output to a cryptographically strong encryption algorithm [15] and enumerating all inputs [20].

2.7 Black Box Protection

[21] show that the existence of one-way functions implies that theoretically, it is easy to accomplish black box protection. We can hide the output of our circuit without the aid of external random numbers (which may compromise the hard-to-invert property) as follows[4]:

$$C'(x) = (f_{k'}(x), f_k(f_{k'}(x)) \oplus C(x))$$

where f is a key-based one-way function, k is the recoverability key and k' is another key randomly generated during the program encryption process. This additional randomness key ensures that if, for example, $C(1) = C(5)$, not only is it the case that $C'(1) \neq C'(5)$, but it is also the case that there is no way for an adversary with black-box access to C' to determine that $C(1) = C(5)$.

2.8 Characterizing White Box Protection

The real stumbling block to successfully developing a program encryption engine is, as one might expect, protecting the white-box implementation. This will be the primary focus of this thesis. McDonald and Yasinsac postulate (under the random program oracle model [14]) that if an adversary, given full access to P is unable to distinguish $O(P)$ from a randomly selected program then $O(P)$ is intent protected.

Full intent protection implies that neither white box structure or black box behavior reveals programmatic intent. As Figure 2.2 depicts, full intent protection involves transformation of both input/output relationships ($s(p, k, X, Y)$ and $t(p, k)$) as well as semantic preserving changes to the internal representation of a program ($w(p'', k)$).

Though other work focuses on black box transformations [1, 21], in this thesis we focus on white box changes that preserve semantics. In particular, we may con-

[4]suggested by [1]

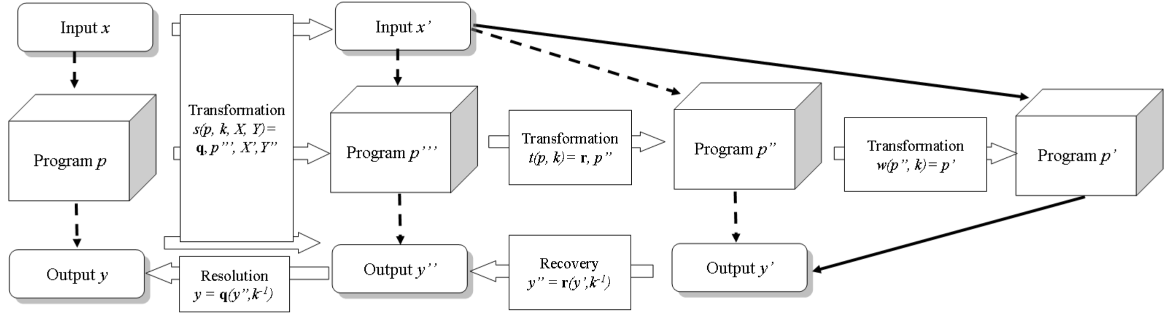

Figure 2.2: Full intent protection

sider white box transformation as an iterative set selection algorithm. A white box algorithm that uniformly selects a random equivalent program from a bounded set of all programs with equivalent behavioral semantics provides one way to characterize a random program or "set" selection.

Since we can accomplish black box protection in polynomial time, the question remains whether an efficient (polynomial-time) algorithm exists that can perform a fully uniform, random selection when given some initial program as a starting point. For small, bounded program sizes, we can enumerate all possible program representations and in polynomial time make a uniform, random selection from such a set. For the purposes of this research, we limit programmatic scope to functionality embodied in combinatorial logic circuits. In Chapter III, we describe the design and implementation of software that performs full set enumeration and random selection of combinational circuits. In Chapter IV, we characterize the efficiency of our algorithm and analyze the polynomial limitations for efficient set generation based on specific circuit characteristics. In Chapter V, we provide a more formalized definition for set selection operations.

III. Methodology for Circuit Enumeration and Random Selection

We developed the circuit enumeration algorithm as a method of enumerating entire sets of legal circuits for two experimental purposes: random circuit selection and complete circuit set analysis. The initial motivation for developing this algorithm was the ability to select a random circuit with a specific truth table from the size-bounded set of all possible circuits. Research indicates that an algorithm that performs random sub-circuit selection and replacement (as visualized in Figure 3.1) may provide a level of randomness to a program that can frustrate adversaries. The algorithm developed in this chapter is designed to fulfill the requirements of the "Random Equivalent Selection" component of this meta-algorithm. We postulate that performing these operations repeatedly can results in a diffusion of control flow within the circuit, analogous to the confusion/diffusion properties of operations in cryptographic theory [19]. For a detailed look at the "Sub-circuit Selection" component, see [17]. Figure 3.2 gives an example selection/replacement operation.

3.0.1 Bounding the obfuscation question. Essentially, all obfuscators perform set selection operations. That is to say an obfuscator takes as input a program from the set of all programs and return another program from the set of all programs. We can further qualify this by saying that for any program, P we can define $O(P)$ as a selection from a size-bounded set of programs, that is, from a finite set. Additionally, we can distinguish between a semantics preserving transformation, one that selects from the set of equivalent programs and a recoverable semantics changing transform, one that selects from the set of programs See Figure 3.3 for a visual representation of the two types of transformations.

3.0.2 Circuit set analysis. An additional motivation, one that we explore in Chapter V, is the ability to perform empirical analysis of entire sets of circuits. We believe that if it is possible to somehow characterize an entire set of circuits, there may

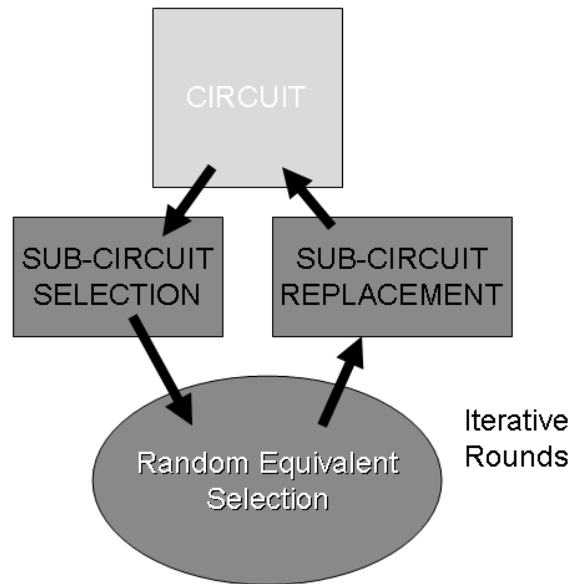

Figure 3.1: A high-level visualization of a sub-circuit selection/replacement obfuscation algorithm

be a way to reveal certain uniform properties of randomness that may lead the way forward toward an obfuscation algorithm that can seek to optimize these properties.

3.0.3 Implementation. As a proof of concept, we developed a Java implementation of this algorithm to support various types of interfaces and methods of persistence. See Figure 3.4 for a domain model. For more information on this implementation, see Appendix A.

3.1 Sets of circuits

Both purposes bring up a question: what exactly constitutes a set of legal circuits? Here we explore several different approaches.

3.1.1 Graphs. One way to approach the question is to treat each circuit as an undirected graph where each node represents a gate or input and each edge represents a connection between two gates. A naive approach would be to assume that our set of circuits is the size-bounded set of all graphs. This method would be obviously advantageous as the algorithm would be quite simple and the size of

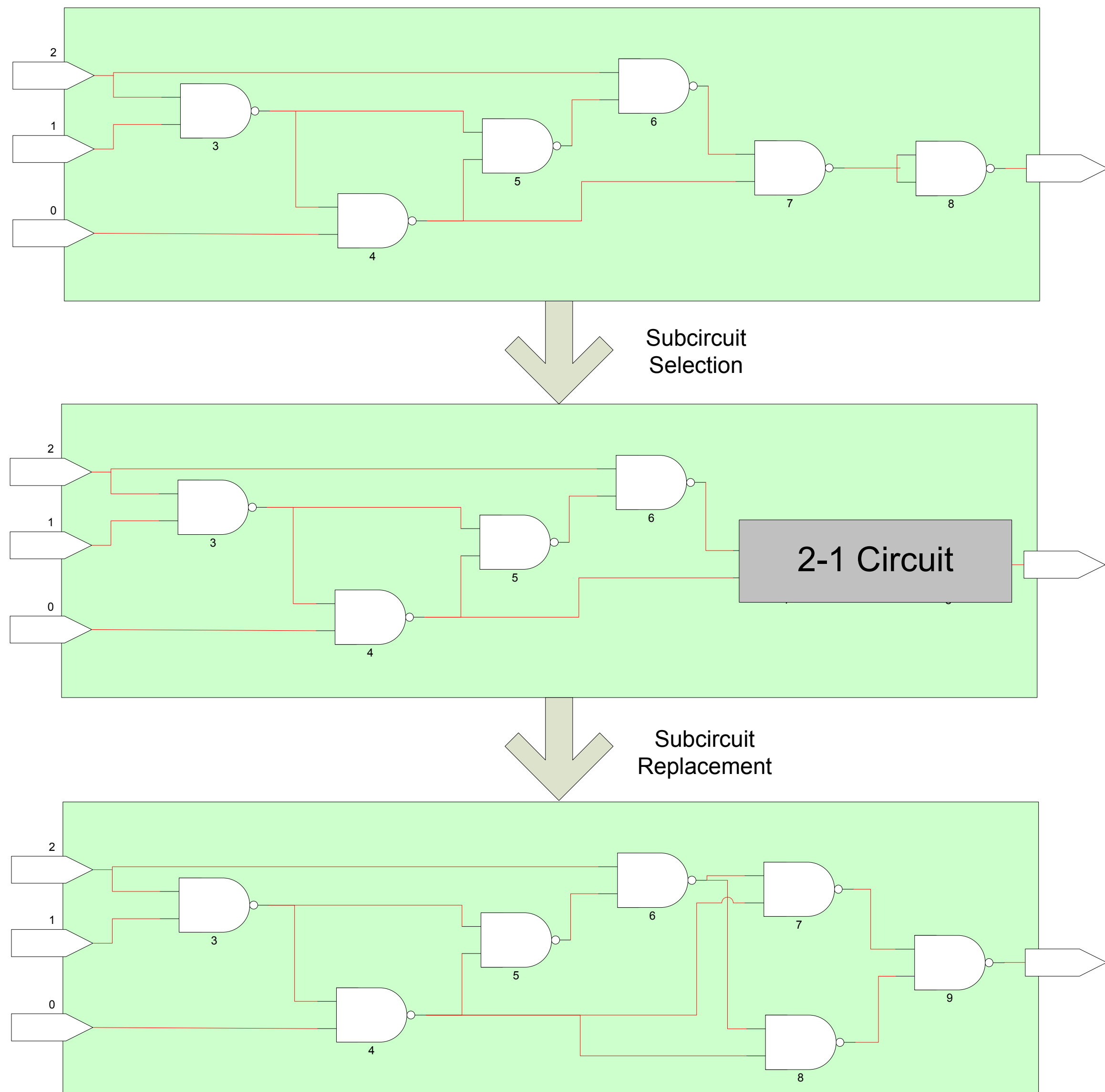

Figure 3.2: An example of a sub-circuit selection and replacement

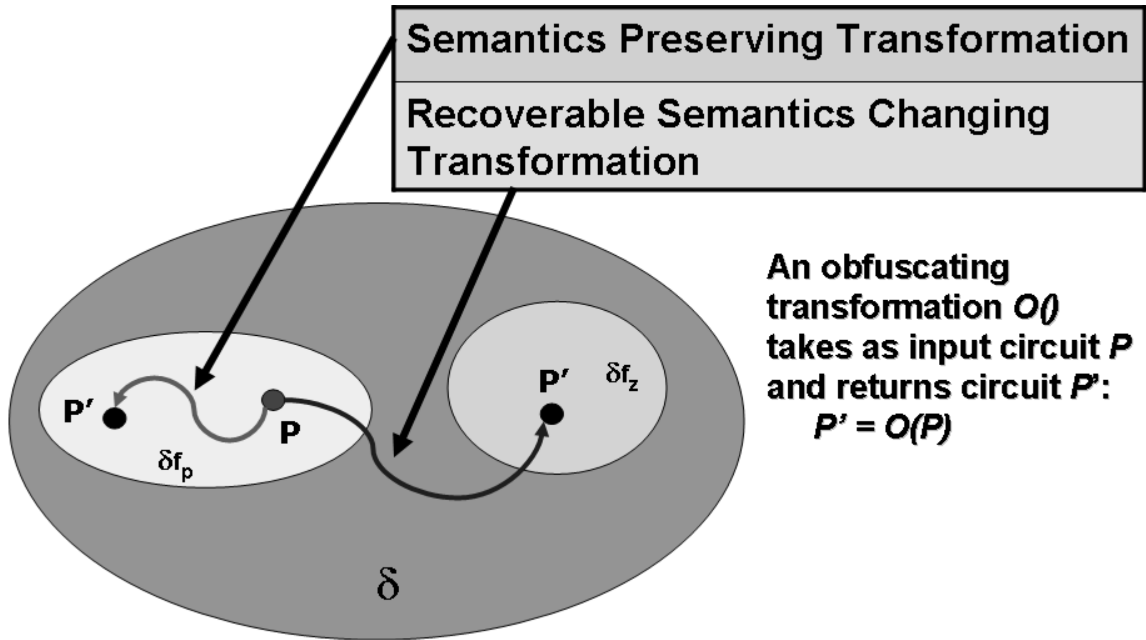

Figure 3.3: Obfuscating transformations as set selection operations. δ represents the finite set of programs to select from. δf_p represents the set of programs in δ that are semantically equivalent P. δf_z represents the set of programs in δ with recoverable functionality of δf_p.

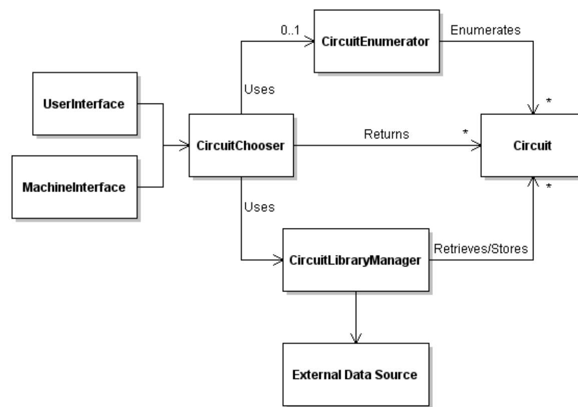

Figure 3.4: Implementation domain model

a set would be easily calculable. Indeed, some researchers have used just such an approach [16].

However, we can easily see that this method would tend to create illegal circuits. Specifically this would generate, circuits with loops, gates with no outputs, and gates with no inputs, connections between inputs, and inputs to inputs. A solution using undirected graphs (where an edge represents a forward connection only) or directed acyclic graphs would also tend to have all of the previous problems except for inputs to inputs and circuits with loops. This does not mean that a graph data structure is useless for representing a circuit, but it clearly does not present a method for easily enumerating legal circuits.

3.1.2 A more complex notion. In fact, what constitutes a "legal circuit" is not obvious. Even assuming that a circuit consists of inputs and gates with exactly two inputs each, some of which are also outputs, there are still quite a few questions that we need to ask in addition to the obvious questions of number of inputs, number of outputs, and size.

1. What types of gates do we allow in the circuit? A two-input Boolean circuit can exhibit as many as 16 different behaviors.
2. Are gates symmetric? That is to say, should we consider a gate with inputs (X_1, X_2) as equivalent to a gate with inputs (X_2, X_1)? This will depend on which gate types are used.
3. Should we allow gates that are identical to other gates based on the inputs? That is to say, can we have two gates in a circuit such that the truth table for each gate, based on all *circuit* inputs, is the same? (See Figure 3.6.)
4. Should we allow the circuit immediate access to the constants $True$ and $False$? Gates that exhibit these properties may exist in a circuit, but it may change the properties of a set of circuits if these constants are available immediately.

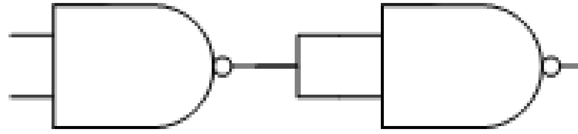

Figure 3.5: An example of a gate where both inputs have the same origin

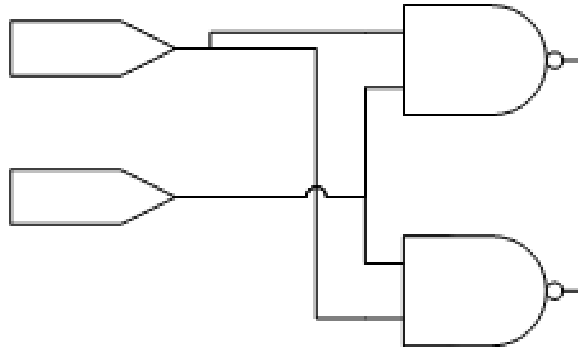

Figure 3.6: An example of a circuit containing two gates with the same external signature

5. Should we allow both inputs to a gate to originate in the same place? (See Figure 3.5.)

6. Does the set contain all circuits within a certain size bound or only all circuits of an *exact* size?

7. Which gates may be outputs? In an ideal circuit, any gate may be an output. However, if we want to index a circuit by output signature, outputs must be restricted to a specific set of gates.

Since users of this algorithm may require various answers to each of these questions, it must allow a significant amount of flexibility. That is to say, the algorithm must allow users to select values for each of these.

3.2 An algorithm framework

A basic recursive enumeration algorithm is as follows:

```
generateAll(gateNum) {
1.   for each gate type:
```

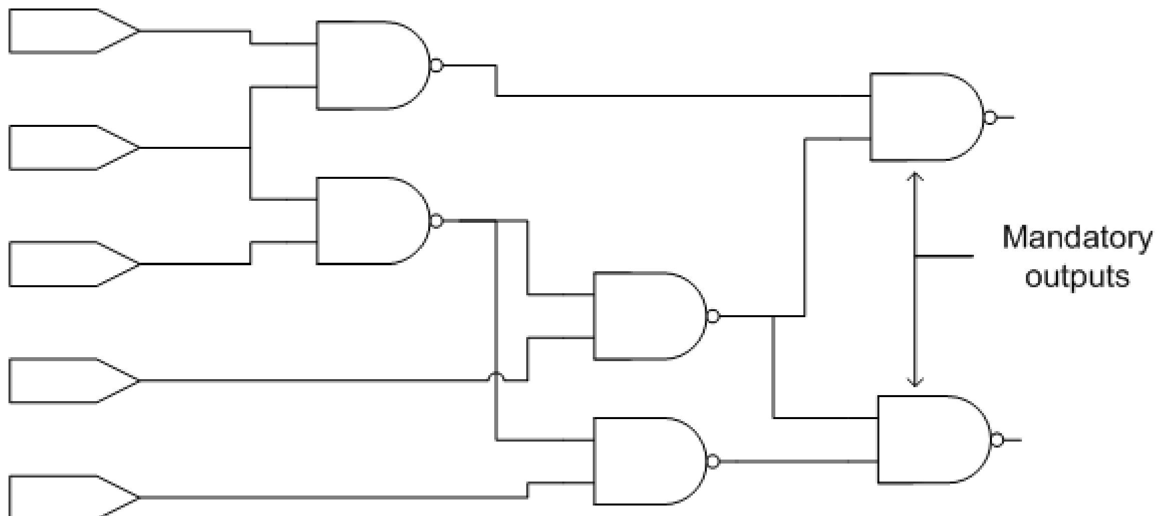

Figure 3.7: A circuit with two open outputs

```
2         for each possible combination of inputs:
3.            add gate type with specified inputs
4.            if (legal circuit)
5.                output circuit
6.            if(gateNum<size bound)
7.                generateAll(gateNum+1)
}
```

The reader may note that there is a subtlety regarding output gates. Since every gate and every input in a circuit is a potential output, it might at first appear that we can ignore the number of output gates altogether when determining which circuits a specified circuit type can categorize. After all, an algorithm can assign each output to any gate or node in the circuit. However, we must also consider the fact that all circuits must have at least one gate that does not output to another gate in the circuit. We must assume that this gate must be an output. What if there are more than one of these gates? This introduces the concept of "*open outputs*" (see Figure 3.7) and causes us to realize that a circuit may have no more open outputs than the total number of outputs. Line 4's legal circuit check accounts for this. However, one must note that just because a circuit has an illegal number of open outputs, it is

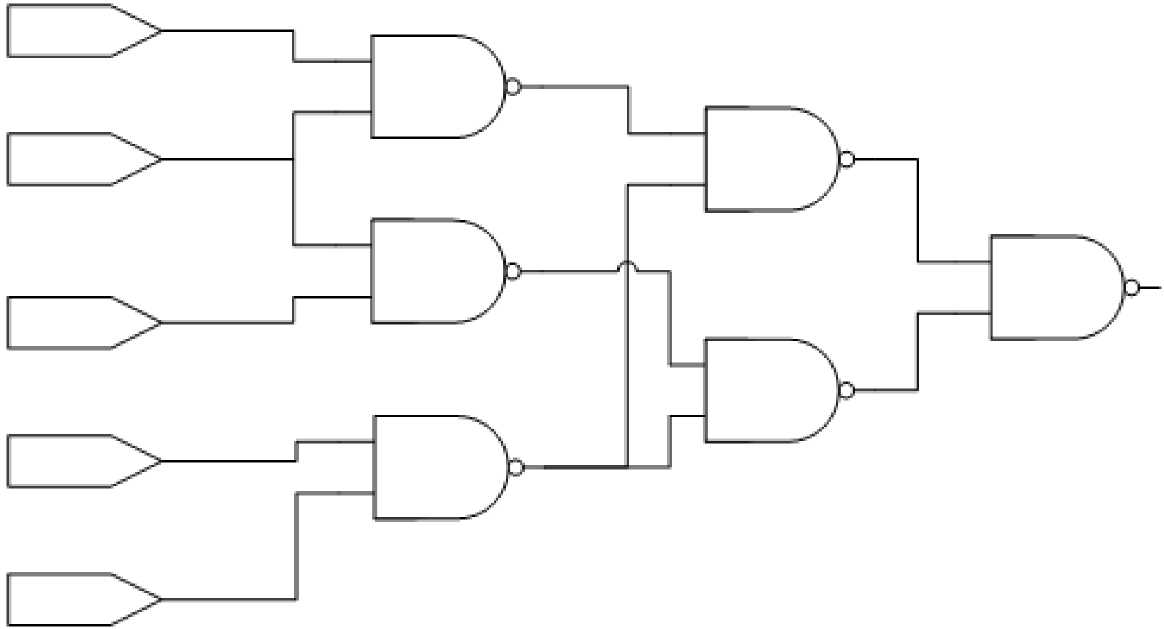

Figure 3.8: An example of a one-output circuit that is not enumerable unless the algorithm allows intermediate circuits with an illegal number of open outputs

not necessarily the case that circuits based on this circuit have an illegal number of outputs. See, for example, Figure 3.8.

3.3 Refining the algorithm framework

We can see that this algorithm satisfies question 1 but does not address any of the other questions. For each other question, we introduce a Boolean parameter to the algorithm and modify the algorithm as follows:

3.3.1 `SymmetricGates`*.* This variable answers question 2. This effects the set of possible combinations of inputs referred to in line 2 of the algorithm.

3.3.2 `RedundantGates`*.* This variable answers question 3. To implement, we need to add an additional filter after line 2 checking for redundant gates.

3.3.3 `AllowConstants`*.* Responding to question 4, we can modify line 2 to include the two constants along with the circuit's standard inputs.

*3.3.4 **DoubleInputs**.* See question 5. To implement, we again modify line 2 to include this property in the legal combination of inputs.

*3.3.5 **ExactCount**.* See question 6. If this is true, we modify the definition of a legal circuit in line 4 to include an exact number of gates.

*3.3.6 **SimpleOutputs**.* See question 7. Once again, we turn to line 4 and adjust the definition of a "legal circuit". This time, we ensure that all open outputs in a circuit must be at the end of the circuit, that is the last n gates where n is the number of outputs in the circuit definition.

3.3.7 A refined algorithm. After taking these new constraints into account, we can create a fully fleshed-out algorithm:

```
generateAll(gateNum)
{
   for each gate type:
      for each enumerateInputCombinations()
          add gate type with specified inputs
          if (RedundantGates and truth table of new gate is
              not equal to another gate's truth table)
              if (legalCircuit())
                  output circuit
              if(gateNum<size bound)
                  generateAll(gateNum+1)
}

enumerateInputCombinations()
{
   if(AllowConstants)
     include the constants True and False with the inputs
```

```
   select all gates and inputs g (1..n)
   if(SymmetricGates)
      deselect all combinations (a,b) such that a<b
   if(not DoubleInputs)
      deselect all combinations (a,b) such that a=b
   return remaining combinations
}

legalCircuit(){
   if (ExactCount and circuit does not contain
      the maximum number of outputs)
      return false
   else if (SimpleOutputs and any but the last numOutputs
      gates contain open outputs)
      return false
   else if (the circuit contains more than numOutputs
      number of open outputs)
      return false
   else
      return true
}
```

IV. Analysis of Functionally Equivalent Logic Families

This chapter demonstrates some of the metrics that researchers may use to characterize both a random selection from a circuit family and the family itself in a meaningful way. This chapter focuses on analyzing circuit families that can be enumerated and on analyzing the efficiency of the enumeration and random selection software developed in Appendix A. In this chapter, we will refer to classes of circuits using the following notation: **I-O-G-T-R-C-D-E-S**

- **I** numbers of inputs
- **O** number of outputs
- **G** number of gates
- **T** The set of gate types
- **R** `RedundantGates` property
- **C** `AllowConstants` property
- **D** `DoubleInputs` property
- **E** `ExactCount` property
- **S** `SimpleOutputs` property

For example, the circuit family **3-1-3-And+Or-T-T-F-T-T** would correspond to the family of circuits with 3 inputs, 1 output, 3 gates, gates of type AND and OR and the remaining Boolean properties set as specified. Note that we do not include the `SymmetricGates` property because we will not be using asymmetric gates and so always set this property to true.

4.1 Library size

It should be apparent from the description of the algorithm that increasing the **I**,**O**, and **G** properties will increase the size of the library as will increasing the magnitude of the **T** property. We can also see that the **R**, **C**, and **D** properties will increase the size of the class and that the **E** and **S** properties will decrease its size.

Table 4.1: Count of 3-1-X-Nand circuits with various properties.

	Smallest	+RedundateGates	+AllowConstants	+DoubleInputs	-ExactCount
1	3	3	10	15	15
2	9	9	50	90	105
3	33	45	400	855	960
4	186	333	4550	11430	12390
5	1350	3393	68800	201195	213585
6	12936	45369	1323950	4468050	4681635
7	152532	769005			
8	2141907	16093413			

For our first set of experiments, we consider just the 3-1-X family with only `Nand` gates. In Table 4.1 and Figure 4.1 we start with the smallest circuit classes in the family (3-1-X-Nand-F-F-F-T-T) and selectively activate and deactivate switches until we have the largest. (Note that for the 3-1 family we do not change the `SimpleOutputs` property because a one output circuit will always have this property.)

One interesting thing to note: the last two lines on the graph appear to overlap. This is because the size increase produced by negating the `ExactCount` property is overshadowed by the exponential blow-up of the circuit class. We elide this property in the other graphs. This can be compared to other families such as 4-1-X-Nand (Figure 4.2) or 3-1-X-And+Or+Nand+Nor+Xor+Xnor (Figure 4.3).

4.1.1 Fitting the data to a curve. Using exponential regression, we can fit data acquired in this manner to an exponential curve of the form $y = ae^{b \times x}$. For example, running exponential regression on the data in Table 4.1 produces the data in Table 4.2[1].

4.2 Output entropy

One metric that may be useful for some applications, including perhaps one way of distinguishing a circuit with output that appears to be truly random is via

[1]This data was produced using the web-based regression application available at http://www.xuru.org/rt/ExpR.asp

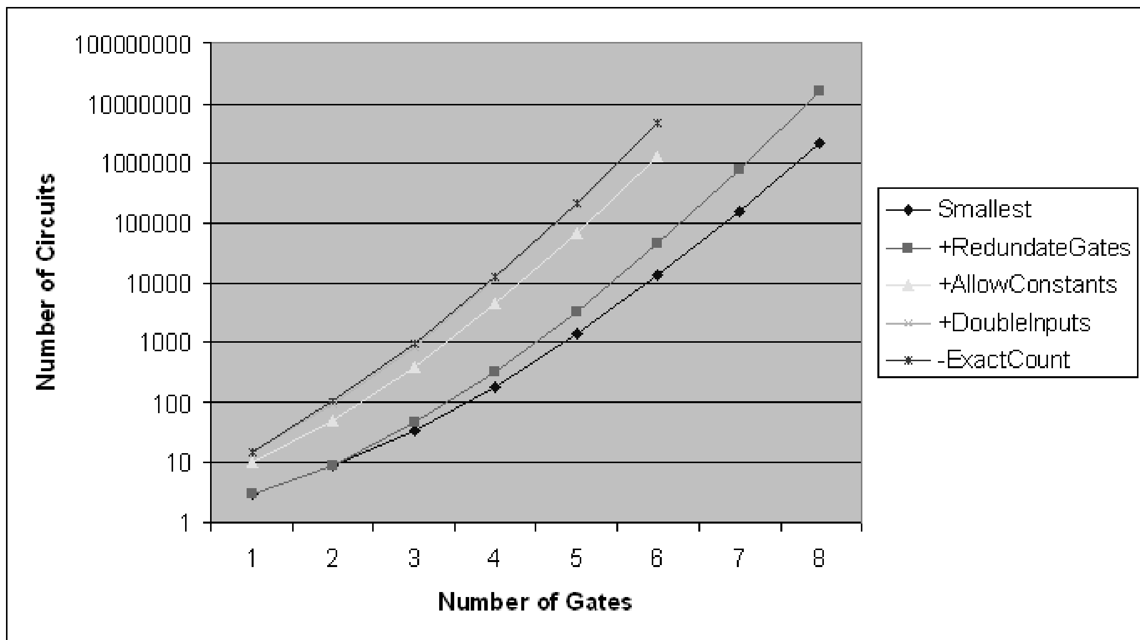

Figure 4.1: Count of 3-1-X-Nand circuits with various properties

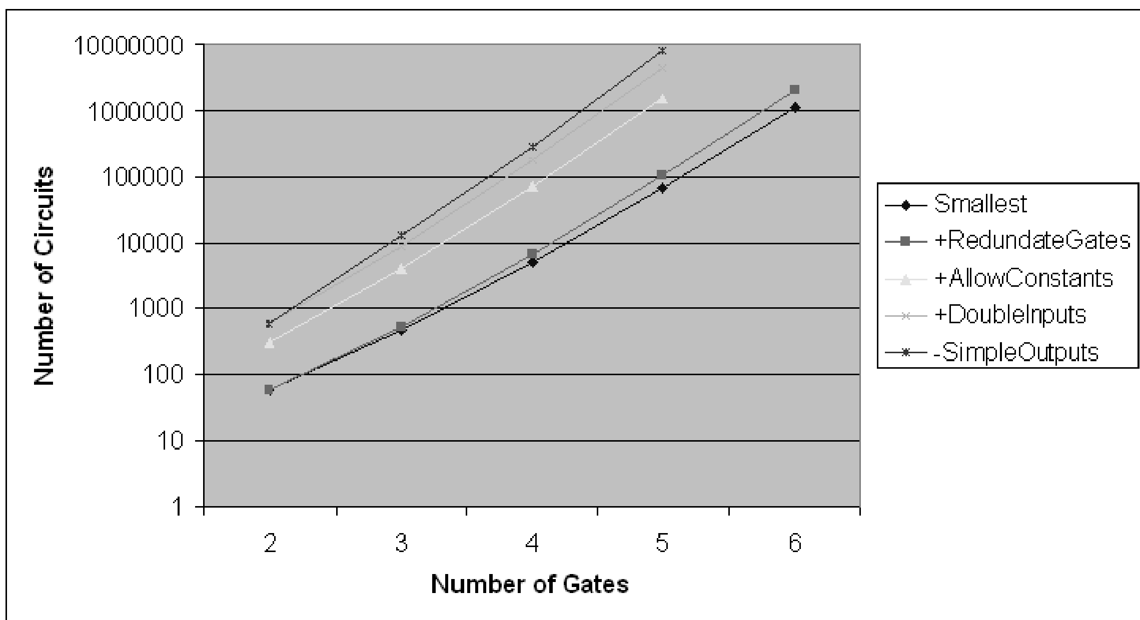

Figure 4.2: Count of 4-1-X-Nand circuits with various properties

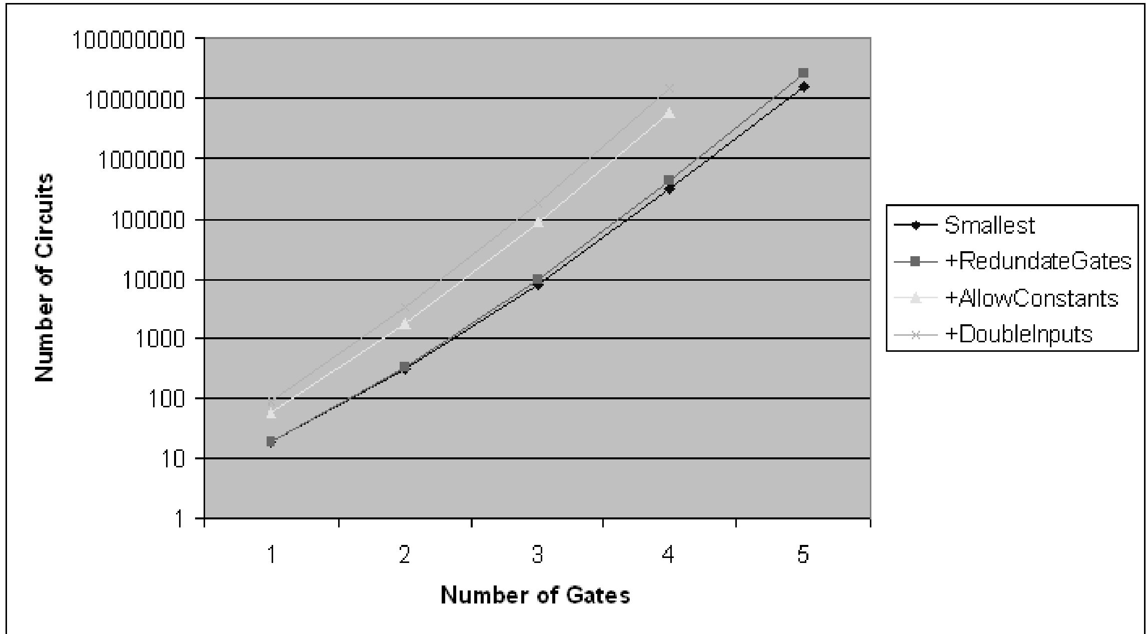

Figure 4.3: Count of 3-1-X-And+Or+Nand+Nor+Xor+Xnor circuits with various properties

Table 4.2: Best fit of some of the data in Table 4.1 to $y = ae^{b \times x}$.

	Smallest	+RedundateGates
a	$1.933749384 \times 10^{-3}$	$5.538010625 \times 10^{-4}$
b	2.602981062	3.011479389
Error for 1	2.97388677	2.988748183
Error for 2	8.647368738	8.771391936
Error for 3	28.23809065	40.35527049
Error for 4	121.6954897	238.6310097
Error for 5	481.6361261	1475.664165
Error for 6	1209.67142	6413.653633
Error for 7	5819.569087	22467.72118
Error for 8	3537.836907	12717.36283

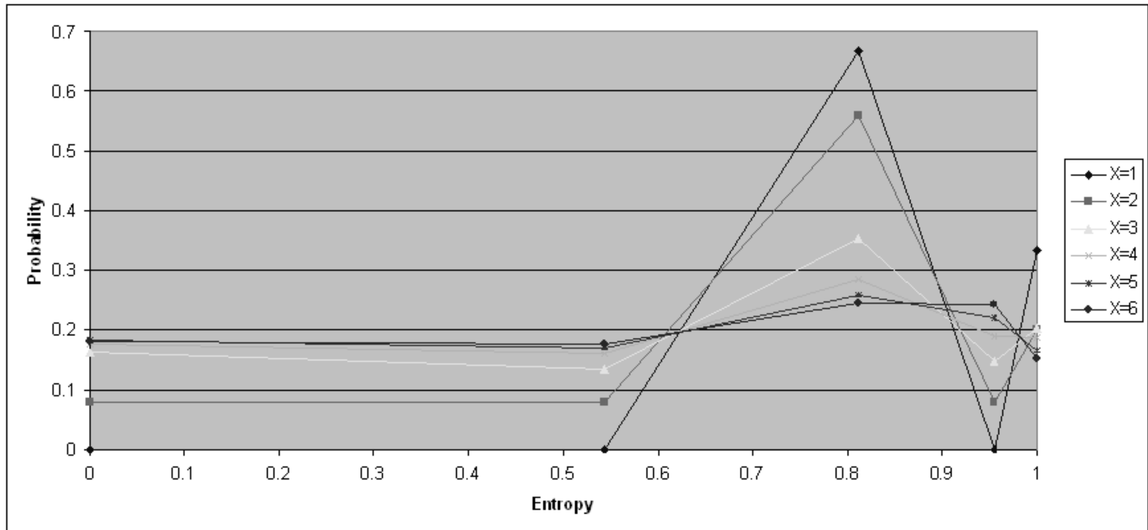

Figure 4.4: Black-box entropy distribution for some of the 3-1-X-And+Or+Nand+-Nor+Nxor+Xor-F-F-F-T-T classes of circuits

some measure of entropy. One measure of entropy for a circuit, as used by Macii and Poncino [12] is that of a circuit's output entropy or what one might call black-box entropy (We use the term "black-box entropy because we are referring to the entropy of the circuit when viewed as a black-box, that is without any information about the internal structure . Whether a similar concept of "white-box entropy" also exists is an interesting open question.) This is one way of calculating the "uncertainty" or "randomness" of a circuit's output.

We compute the black-box entropy H of a circuit X via the following formula.

$$H(X) = -\sum_{i=1}^{2^n} p_i \log_2(p_i)$$

where n is the number of outputs and p_i is the probability over all input combinations that the circuit will output the ith possible output combination.

Figure 4.4 gives an example of the entropy distribution for several sets of circuits. At this time, we can not say much about this particular property. However, it should be noted that there is no guarantee that entropy will indicate the presence of the

x_1	x_2	x_3	y_1	y_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

Figure 4.5: An example truth table of a function with maximal black-box entropy but other undesirable properties

desired "hard to invert" property discussed in Chapter II. For example, any circuit where all possible output combinations are equally represented across all possible input combinations would have maximal black-box entropy. For example, the function described in Figure 4.5 specifies a function that is easy to invert but has maximal black-box entropy.

4.3 Structural metrics

In Chapter V, we mention the possibilities for structural metrics that reveal nothing about a circuit's black-box characteristics. But what is a concrete example of a structural metric? Let's consider the obvious metric of "fanout". Simply, fanout for a gate or an input is the number of gates that are dependent on that gate or input. One might think that *average* fanout would be a possible structural metric for a family, but we need to realize that this metric will always be the same for a given family of circuits with the same number of inputs, outputs, and gates. This is because the total number of inputs to gates in a circuit will always be the same (that is, 2 times the number of gates). But what about a distribution metric? Here's an interesting experiment: let's try to find out what effect the boolean variables have on average fanout of a circuit library.

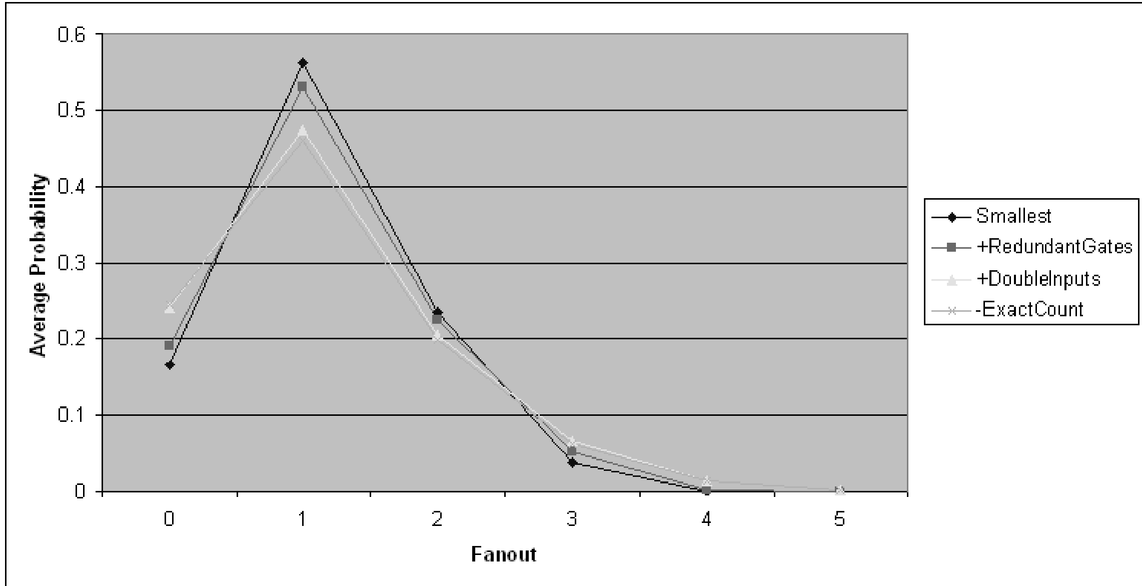

Figure 4.6: Average fanout distribution for some of the 3-1-4-Nand classes of circuits

Figure 4.6 gives an example of average fanout distribution of a few circuit classes. It is interesting to note that properties that increase the set size tend to skew the fanout distribution away from the median.

4.4 jCXL performance analysis

Since we can estimate library size via exponential regression we can also estimate the performance of the jCXL software (in terms of both time and disk space use) in much the same way. First though, we need to set up some baselines and establish a relationship between library size and enumeration time.

Experiments on 104 different libraries[2] revealed a strong linear correlation between library size $\times$ maximum number of gates and disk usage (see Figure 4.7.) This reveals that we can estimate disk usage of a library via the equation $y = 0.307x - 16.239$ where x is product of library size and number of gates and y is the estimated disk space usage in kB. Likewise, these same experiments reveal a weaker

[2]using an experimental environment of a Intel Xeon CPU 5160 @ 3.00FHz, 3.00 GB of RAM running Windows XP

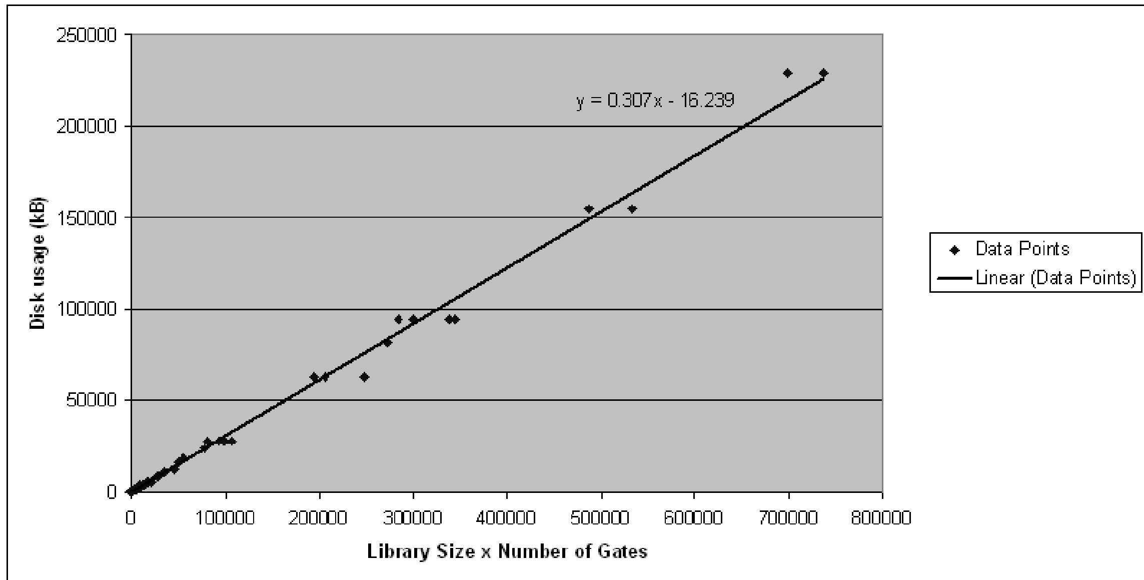

Figure 4.7: Relationship between library size × number of gates and disk usage with linear regression

but still quite significant linear correlation between library size × maximum number of gates and time to persist on our test machine (see Figure 4.8.) This time the relevant equation is $y = 0.0906x - 262.58$.

So for any library class we can estimate the number of circuits in the library via exponential regression and then estimate the time to enumerate and disk space usage using the two above formulas. In the next chapter, we look at the software obfuscation problem from a slightly different direction and explore some of the more theoretical notions of obfuscation discussed in Chapter II.

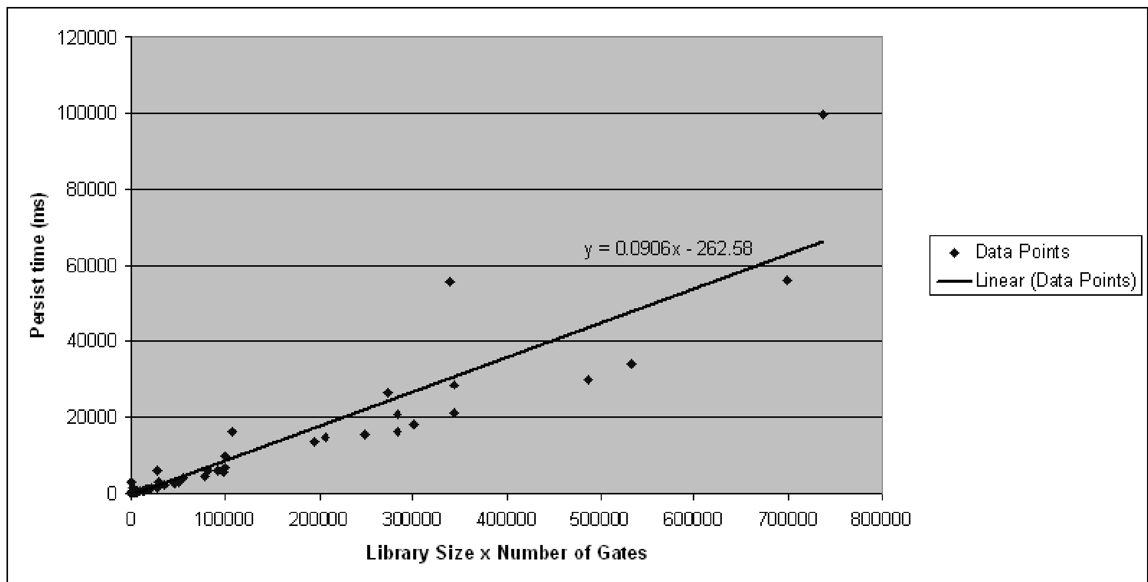

Figure 4.8: Relationship between library size × number of gates and time to persist with linear regression

V. Advancing Some Theoretical Models of Obfuscation

5.1 Motivation

In this chapter, we explore some weaknesses in previous formal models of obfuscation, introduce some new models, and make some statements about these new models.

There are two obvious directions we can take in an attempt to bridge the gap between theoretical and practical software obfuscation: expand the practical to include the theory, or expand the theory to include the practical. The previous chapter expanded the practical; this chapter expands the theoretical.

5.2 Intent Protection Weakness

If trapdoor one-way functions exist, the definition of intent protection introduced in Chapter II may not be strong enough. Consider the following pseudocode ($F(y)$ is a trapdoor one-way function.):

```
P(x){
    sqr=x^2;
    return F(sqr);
}
```

Since F is a one-way function, it is impossible for an adversary to determine *sqr* based on y even with access to this source code. *sqr* is a function of x so the adversary also cannot determine x. However, an adversary with access to this source code will be able to determine that the program computes and returns an encryption of a square function. This seems to defeat the purpose of protecting the intent of a program. Therefore, we propose a new definition of intent protection based on the previously virtual black box property and the standard definition of a one-way function [7]:

Definition V.1. *A circuit C with n inputs and m outputs is* virtual black box intent protected *if the following conditions hold:*

- *("virtual black box" property) For every PPT A, there is a PPT S and a negligible function α such that*

$$\left|\Pr\left[A(O(C))=1\right]-\Pr\left[S^{C}(1^{|C|})=1\right]\right|\leq\alpha(|C|)$$

- *("hard to invert" property) For every PPT A', there is a negligible function α' such that*

$$Pr[C(A'(C,y))=y]\leq\alpha'(|C|)$$

  where $y\leftarrow\{0,1\}^{m}$

5.3 Program encryption and random programs

We introduced the concept of program encryption in Chapter II. In [14], McDonald and Yasinsac introduce the *random program model*, a practical framework for defining the semantics of software that correlates to the traditional notions of a data encryption cypher. A program satisfies the random program model if it is computationally indistinguishable from a comparable randomly generated program. They define an ideal program encrypter as one that would generate a program that satisfies the random program model. Formally, we define program encryption as follows:

Definition V.2. *A probabilistic algorithm T is a* program encrypter *if for all circuits C the following four conditions hold:*

- *Functionality recoverability: There exists a probabilistic algorithm, R such that for every string k, and every x in the domain of C $R(T(C,k)(x),k)=C(x)$.*
- *Polynomial size increase: There is a polynomial p such that for every string k, $|T(C,k)|\leq p(|C|)$*
- *Hard to invert: For every PPT A, every string k, and every x in the domain of C there is a negligible function α' such that*

$$Pr[(T(C,k)(A(T(C,k)(x)))=T(C,k)(x)]\leq\alpha'(|C|)$$

This is to say that if we program encrypt the circuit with some key k as $T(C, k)$, that some adversary A, when given the output of $T(C, k)$ for any x is unable to produce some x' such that $T(C, k)(x) = T(C, k)(x)$ with better than negligible probability. This is the same thing as saying that $T(C, k)$ must evaluate a one-way function.

- *Obfuscation: The VBB, indistinguishability, or BP property.*

Note that because the output of a program encrypted circuit is not useful to an adversary, Barak's impossibility results do not rule out the possibility of creating a VBB program encrypter. For the remainder of this paper, we will refer to VBB program encrypters simply as program encrypters. However, we conjecture that all three types of program encrypters are equivalent. An alternative, utilizing the random program model, which we refer to as random program encryption is as follows:

Definition V.3. *We define a* statistical/computational random program encrypter *T the same as a program encrypter except that we replace the Obfuscation property with the following:*

- *Random program indistinguishability: With k as a random variable, the distribution $T(C, k)$ is respectively statistically/computationally indistinguishable from a uniform distribution of the set of* all *programs of the same size as $T(C, k)$.*

This definition was introduced because intuitively, the idea of producing a program with certain measurable properties of randomness may logically lead toward an algorithm for producing this effect. However, we suspect that it is impossible for a program to possess both the the "random program indistinguishability" property and the "hard to invert" property. Intuition combined with physical examination of some types of randomly generated programs tend to indicate that the signatures of random programs do *not* appear random and therefore do not satisfy the "hard to invert" property. (For example, we would expect random programs to generate significant number of gates with constant output.) Whether it is possible to define "random" programs in such a way that their output is computationally indistinguishable from

a one-way function is an intriguing open question[1]. However, for the purposes of this thesis, we will look for a purely "structural" way of comparing two circuits. In an attempt to find a similar definition that is equivalent to that of a program encrypter, we propose a new definition that we base only on "structural" properties of the circuit. We also propose a new type of obfuscation based on the same properties.

Definition V.4. *We define a* structurally random program encrypter T *the same as a program encrypter except that we replace the Obfuscation property with the following:*

Random program structural indistinguishability: With k as a random variable, the distribution $T(C,k)$ is structurally indistinguishable from a program randomly selected from the set of all programs of the same size as $T(C,k)$.

Definition V.5. *A* structurally random obfuscator *is defined the same way as a VBB obfuscator except that the VBB property is replaced with the following:*

- *Random program structural indistinguishability: The distribution $O(C)$ is structurally indistinguishable from a program randomly selected from the set of all programs of the same size as $O(C)$.*

These definitions introduce a new term: *structurally indistinguishable.* Before we formally define this term, some discussion on what it means is appropriate. Consider our previous definition of a computational random program encrypter. In order for the two distributions to be computationally indistinguishable, it must be the case that no PPT can distinguish between a truly random program and a program encrypted program. But we know that truly random programs exhibit I/O behavior that is not consistent with the "hard to invert" property so any PPT can look for this behavior. Our goal, therefore, is to restrict the set of distinguishing PPTs to just those that do not perform I/O analysis. That is to say, just those that perform *structural analysis.* But how in the world can we do that? This leads us to the concept of *structural indistinguishability.*

[1] Perhaps one approach would be to consider sets of circuits without the `RedundantGates` property mentioned in Chapter III. One way of at least determining that a circuit is not hard to invert might involve the use of Shannon entropy as used in [11].

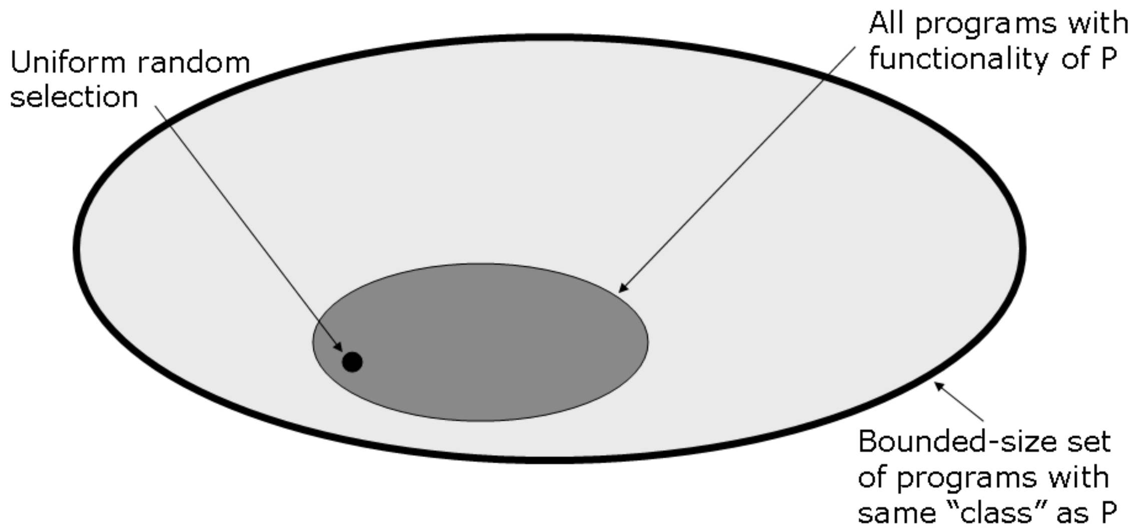

Figure 5.1: A visualization of a uniform random set selection

5.4 Set selection obfuscation

Definition V.6. *A* Computational/statistical/perfect set selection obfuscator *is defined the same way as a VBB obfuscator except that the VBB property is replaced with the following:*

- *Computational/statistical/perfect set selection property: There exists a polynomial α such that $O(C)$ is respectively computationally/statistically/perfectly indistinguishable from a uniform distribution over the set of all all circuits of size $\alpha(|C|)$ that perform the same function as C.*

  See Figure 5.1 for a graphical notion of this concept.

5.4.1 Comparison with previous definitions.

5.4.1.1 Indistinguishability Obfuscation. We can prove that a perfect set selection obfuscator is also a perfect indistinguishability obfuscator. We can also prove that statistical and computational set selection obfuscators are (respectively) statistical and computational indistinguishability obfuscators.

Proof. Suppose that algorithm O is a perfect set selection obfuscator and C_1 and C_2 are two circuits that compute the same function and are the same size k. By the definition of a set selection obfuscator, the distribution $O(C_1)$ is identical to the distribution $O(C_2)$. Therefore, for any algorithm A

$$|\Pr[A(O(C_1))] - \Pr[A(C_2)]| = 0$$

Therefore O is a perfect indistinguishability obfuscator. □

Proof. Suppose that algorithm O is a statistical set selection obfuscator and C_1 and C_2 are two circuits that compute the same function and are the same size k. We'll refer to a perfect set selection obfuscation for C_1 or C_2 as P. By definition

$$|\Pr[A(O(C_1))] - \Pr[P]| < c$$

and

$$|\Pr[A(O(C_2))] - \Pr[P]| < c$$

Therefore, by simple algebra it must be the case that

$$|\Pr[A(O(C_1))] - \Pr[A(O(C_2))]| < c$$

and O is a statistical indistinguishability obfuscator. □

Proof. Suppose that algorithm O is a computational set selection obfuscator.

Let's refer to the uniform distribution over the set of all circuits of size $\alpha(|C|)$ that perform the same function as P. By definition we know that for any $C1$ and $C2$, $O(C1)$ is computationally indistinguishable from P and $O(C2)$ is computationally indistinguishable from P. That is to say that for any PPT A

$$|Pr[A(O(C1)) = 1] - Pr[A(P) = 1] \leq \alpha(n)|$$

and

$$|Pr[A(O(C2)) = 1] - Pr[A(P) = 1] \leq \alpha(n)|$$

From this, we can see that

$$|Pr[A(O(C2)) = 1] - Pr[A(O(C2)) = 1] \leq \alpha(n)|$$

□

5.4.1.2 Best-possible obfuscation. We know that if O is an *efficient* indistinguishability obfuscator for a circuit family C, then O is also an (efficient) best-possible obfuscator for C [8] and we have proven that perfect/statistical/computational set selection obfuscation is perfect/statistical/computational indistinguishability obfuscation. Therefore we can positively say that an efficient perfect/statistical/computational set selection obfuscator is also an efficient perfect/statistical/computational best-possible obfuscator.

5.4.2 Impossibility result. Since the existence of a statistical/perfect best-possible obfuscator implies a collapse in the polynomial hierarchy [8] and an efficient statistical/perfect set selection obfuscator is a statistical/perfect best-possible obfuscator, the existence of an efficient statistical/perfect set selection obfuscator implies a collapse in the polynomial hierarchy. However, we can not make such a statement about a computational best-possible obfuscator.

5.5 Structural indistinguishability

There are several different intuitive ways of attempting to formally define structural indistinguishability. Structural indistinguishability is a key component in defining an adversary that performs white-box analysis of a program without performing black-box analysis.

5.5.1 *Flawed notions of structural indistinguishability.*

Consider an efficient computational set selection obfuscator $O1$ and a structurally random program obfuscator $O2$ (with the concept of *structural indistinguishability* still undefined.) We know that $O1$ is necessarily a BP obfuscator, that is to say, it is better than any other obfuscator. Now, let's attempt to define *structural indistinguishability* in such a way that $O2$ approximates $O1$. Perhaps we want the distribution of a structurally random obfuscator to be the same as a set selection obfuscator?

But no. Our concept of structural indistinguishability requires that there is some property present in truly random circuits that is not present in others. Since a set selection obfuscator performs a random selection over the entire set, it does not imply the existence of any property that is only true in some circuits.

Consider two intuitively ideal models for obfuscation, reduction to the two-level circuit representation of the circuit and uniform selection from the random set of equivalent programs. While both of these models are inherently non-polynomial to realize, perhaps we can assume that there is something about one of these models that can be quantified in some way that may lead to a polynomial-time algorithm for approximating this "ideal-ness" of a circuit.

Specifically with regards to the two-level model, perhaps there is some property that the two-level representation possesses but that only some of the circuits in the polynomial size-bounded family of equivalent circuits possess. Perhaps we can call this property the *truth table property* and conjecture the following:

For every circuit signature, an algorithm can find the truth table property is in the two-level truth table representation of the circuit as well as in a non-negligible number of circuits within the polynomial-size bounded set of circuits with that signature. This set also contains a non-negligible number of circuits that do not have this property. But once again, there is a flaw: it is easy to define a "truth table property" such that this conjecture is true (for example, if we select the property strictly in terms of the size of the circuit.)

5.5.2 A better notion of structural indistinguishability. Let's once again consider the idea of a random program distinguisher that takes as input the structural property. We will define a means of creating metrics for a circuit that reveal nothing about the black-box behavior of the circuit.

5.5.2.1 NCMP.

Definition V.7. *A* non-black-box circuit metric producer *(NCMP) is a PPT N such that for any PPT D (a distinguisher), any circuit C, the truth table representation of C (tt), the uniform distribution of all random truth tables of the same size (TR)* $D(N(C), tt)$ *is computationally indistinguishable from* $D(N(C), tr)$ *(where tr is a random variable distributed according to TR). That is,* $Pr[D(N(C), tt) = 1] - Pr[D(N(C), tr) = 1]$ *is negligible.*

Definition V.8. *A NCMP N is a* best possible NCMP *if for any NCMP M, any PPT A, and any circuit C there exists a PPT S such that* $S(N(C))$ *is computationally indistinguishable from* $A(M(C))$

This definition guarantees that for every adversary who tries to compute some information about the circuit from the metric produced by *any* NCMP, a simulator can produce that information from the metric produced by N. This definition gives us a NCMP that must generate all non-black-box information about a circuit. From here it is simple to define what it means for two circuits to be structurally indistinguishable (see Figure 5.2)

5.6 Conclusion

Clearly, this section has proposed more questions than it has answered. However, one important question answered is: "How can we formally define a machine that performs only structural analysis of a circuit without performing I/O analysis?" This question is extremely significant because it provides a theoretical basis for a concept that until now has only existed in the world of practical obfuscation: static analysis. We accomplish this via use of our concept of a "best possible NCMP." This

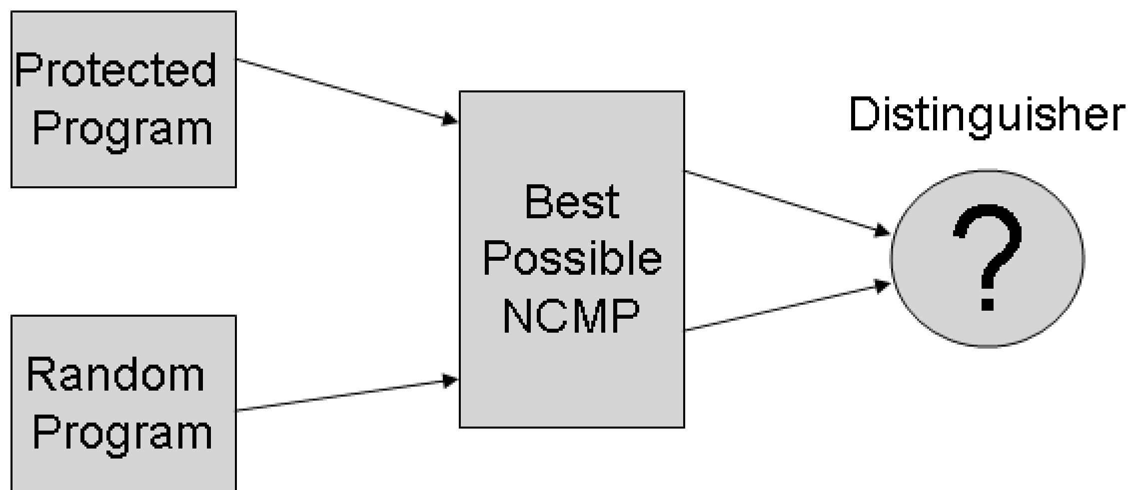

Figure 5.2: A structural distinguisher. If the distinguisher (any PPT) cannot distinguish between the protected program and the randomly selected program using only the best possible NCMP, the programs are structurally indistinguishable.

formalization is one of the key results of this thesis and is certainly worthy of further study.

VI. Conclusions

The research presented in this thesis has significant relevance to the mission of the United States Air Force and United States Department of Defense. Now, more than ever, the protection of critical software is an essential component of national security. However, this thesis aims toward an incremental furthering of the science of program encryption; we did not design it to stand alone. Instead, we expect that its results will one day be pieces in the bigger mission of finally building a bridge between practical and theoretical obfuscation.

Once realized, the applications specified for both VBB obfuscation (impossible in the general case, but certainly open to exploration in other specific cases) and program encryption in Chapter II will have any number of potential military uses. In Chapter III we proposed an algorithm for enumerating entire sets of circuits. This algorithm can be used both for performing random selection of circuits by signature and for perfectly calculating metrics on small classes of circuits. This algorithm also lends itself to a simple modification that allows calculation of statistical metrics on large classes of circuits. Then, in Chapter IV, we demonstrated some of the capabilities of the algorithm introduced in Chapter III and implemented in Appendix A. Finally, in Chapter V, we formally define what it means for an algorithm to perform only static analysis on a circuit (which can be easily translated to a formalization of an algorithm that performs only static analysis on any type of program). This represents a significant step in the science of software obfuscation as, to our knowledge, we are the first researchers to accomplish this feat. If future research is invested in these subjects, the following directions seem the most likely to bear fruit.

6.1 Future work

- Expand/refine algorithm - We developed the algorithm in this thesis in a somewhat ad-hoc manner and in something of a vacuum. That is to say, we did not formalize specifically how it was to be used in combination with a replacement algorithm. Whether a future researcher focuses on the algorithm random

equivalent circuit selection or on metrics generation, a better formalization of the algorithm's purpose should be developed first.

- Optimize software - A researcher could improve the prototype software detailed in Appendix A in at least two obvious ways, both involving efficiency. First, the software is in dire need of speed optimization. The software as it is is actually significantly slower than the legacy software and could be significantly faster if some of the more costly Java constructs were optimized or the program were re-written in C/C++. Additionally, our objectives in constructing the the database-based persistence layer currently implemented were primarily ease of use and not speed. A customize sorted flat-file based system implemented using the `LibraryManager` interface would almost certainly be better than what is currently implemented.
- Analyze usefulness of metrics with regards to properties of random programs - One hypothesis that could be more fully explored is that there is some measurable property of random programs that makes them somehow "more obfuscated" than programs created for a purpose. This thesis touches the surface of this hypothesis but does not espouse any definite opinions on this subject. One possible metric suggested by [16] is the existence of certain "Motifs" or patterns that appear to be more common in constructed circuits than in random circuits.
- Analyze usefulness of metrics with regards to sub-circuit selection - Another way ahead builds on the work of Ken Norman [17]. Developers of algorithms that obfuscate random circuits by selecting sub-circuits and replacing them with random selections must be aware of what the replacement set consists of. This is especially true if the algorithm (as suggested but not implemented by [17]) is driven by a stochastic search algorithm designed to approximate the optimization of some desirable property.
- Adversary characterization - Many techniques in commercial obfuscation rely on informal definitions of the capabilities of an adversary. As a contrast, for-

mal definitions of adversaries (including the ones used in this thesis) rely on comparisons to Turing Machines or other formal constructs in theoretical computer science. One way ahead may be a “best of both worlds” compromise that characterizes a realistic adversary in formal terms.

- Theoretical work - Another possible approach toward future results from this research lies within the realms of theoretical computer science. Specifically, researchers with strong backgrounds in computational complexity theory, information theory, and cryptographic theory may be able to develop and prove additional statements regarding the relationship between obfuscation and theoretical computer science.

Bibliography

1. Barak, B., O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. "On the (im) possibility of obfuscating programs". *Electronic Colloquium on Computational Complexity*, 8, 2001. URL `http://eccc.hpi-web.de/eccc-reports/2001/TR01-057/index.html`.

2. Barak, Boaz. "Can we obfuscate programs?" URL `http://www.cs.princeton.edu/~boaz/Papers/obf_informal.html`.

3. Bethencourt, John, Dawn Song, and Brent Waters. "Analysis-Resistant Malware". *15th Annual Network & Distributed System Security Symposium.* 10 - 13 February 2008.

4. Cockburn, Alistair. "Resources for writing use cases", 2007. URL `http://alistair.cockburn.us/index.php/Resources_for_writing_use_case`.

5. Foundation, The Eclipse. "Eclipse Test & Performance Tools Platform Project", 2008. URL `http://www.eclipse.org/tptp/`.

6. Goldreich, O. *Foundations of Cryptography: Basic Tools.* Cambridge University Press, 2001.

7. Goldreich, O. *Foundations of Cryptography: A Primer.* Now Publishers Inc, 2005.

8. Goldwasser, S. and G. N. Rothblum. *On best-possible obfuscation*, 194–213. Theory of Cryptography. 4th Theory of Cryptography Conference, TCC 2007. Proceedings (Lecture Notes in Computer Science Vol. 4392). Springer-Verlag, Germany; Berlin, 21-24 Feb 2007. ISBN 3 540 70935 5.

9. "H2 Database Engine". URL `http://www.h2database.com`.

10. Huber II, Arthur F. and Jennifer M. Scott. "The Role and Nature of Anti-Tamper Techniques in US Defense Acquisition". *Acquisition Review Quarterly*, Fall 1999:355–367, 1999.

11. Impagliazzo, R., L. Levin, and M. Luby. "Pseudo-random generation from one-way functions". *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, 12–24, 1989.

12. Macii, E. and M. Poncino. "Exact computation of the entropy of a logic circuit". *VLSI, 1996.Proceedings., Sixth Great Lakes Symposium on*, 162–167, 1996.

13. McDonald, J. Todd and A. Yasinsac. "Applications for Provably Secure Intent Protection with Bounded Input-Size Programs". *Proceedings of the International Conference on Availability, Reliability and Security*, 2007.

14. McDonald, J. Todd and Alec Yasinsac. "Of unicorns and random programs". *To appear, 3rd IASTED International Conference on Communications and Computer Networks (IASTED/CCN).* 2005. October 24-26, 2005.

15. McDonald, J. Todd and Alec Yasinsac. “Program Intent Protection Using Circuit Encryption.” *8th International Symposium on System and Information Security*, 8-10 November 2006.

16. Milo, R., S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. “Network Motifs: Simple Building Blocks of Complex Networks”. *Science*, 298(5594):824–827, 2002.

17. Norman, Ken. *Architecture for White-box Obfuscation Using Randomized Subcircuit Selection And Replacement*. Master’s thesis, Air Force Institute of Technology, 2008.

18. Sander, T. and C. F. Tschudin. “On software protection via function hiding”. *Information Hiding*, 111123, 1998.

19. Shannon, Claude E. “Communication theory of secrecy systems”. *Bell Systems Technical Journal*, 28:656–715, 1949.

20. Yasinsac, Alec and J. Todd McDonald. “Towards Working With Small Atomic Functions”. *To Appear in Proceedings of The Fifteenth International Workshop on Security Protocols*, 2007.

21. Yasinsac, Alec and J. Todd McDonald. “Tamper Resistant Software through Intent Protection”. *To appear in International Journal of Network Security*, 2008.

The first step in developing the software was to determine exactly how users of the software should interact with it. We developed a simple primary use case (Figure A.1) using Cockburn's use case template [4].

One key component of this simple use case is the performance target. It becomes quickly obvious that enumeration and lookup of circuits alone will not meet this requirement. Clearly we need some sort of persistence layer. This should lead the reader to something resembling the simple UML model/view/controller-based domain model in Figure A.2.

A.1 Analysis of Legacy Software

Previous researchers performed experiments using a utility designed to generate and print circuits called "CXL." A previous researcher developed CXL using C++. While several different versions existed, all had the following limitations:

- No support for circuits with multiple outputs
- No runtime configuration of options including circuit library size (program needed to be recompiled to generate circuits with different sizes)
- No cross-platform support–only compiles in Unix-like operating systems with gcc.
- Unable to produce output in an easily machine-readable form
- Does not take advantage of computers with multiple processors

The first design decision was whether to modify CXL or to develop a new solution from scratch. An attempt to reverse-engineer CXL revealed that the design of CXL was such as to minimize circuit library generation time at the expense of program flexibility and maintainability. We can also see from the class diagram (Figure A.3) that the original author made some attempt to provide abstraction for the purpose of understandability but that his primary concern was the ability to simplify writing a circuit to a flat file.

```
CHARACTERISTIC INFORMATION
Goal in Context: Allow retrieval of all
circuits with specific characteristics with a specified
‘‘signature’’ (number of inputs, outputs, size of circuit,
input/output behavior.)
Level: Primary Task
Preconditions: None
Success End Condition: Lookup may be performed
Failed End Condition: Lookup may not be performed; error returned
Primary Actor: User or other system
----------------------------------------
MAIN SUCCESS SCENARIO
1. User or other system requests
specification-based circuit lookup
2. System returns circuits
----------------------
RELATED INFORMATION
Performance Target: Minimize time between steps 1 and 2
Frequency: Many times per second
```

Figure A.1: Software use case

An analysis of the source code also reveals a quite tight coupling of the user interface, circuit persistence and circuit generator components specified in the domain model as evidenced in Figure A.4.

A.2 Initial design

Because of the original CXL's limitations and inflexible design, we decided to decided to rewrite it from scratch. We chose to use Java in order to make the program inherently cross-platform and because of Java's built-in multi-threading support. The initial software design focused on the `Circuit` and `CircuitGenerator` components and a rudimentary user interface. This was in order to ensure that the output of this phase was a useful product.

A.2.1 Circuits. First, we needed to determine how to represent a circuit. One key component that our use case implies is a way of representing the specific

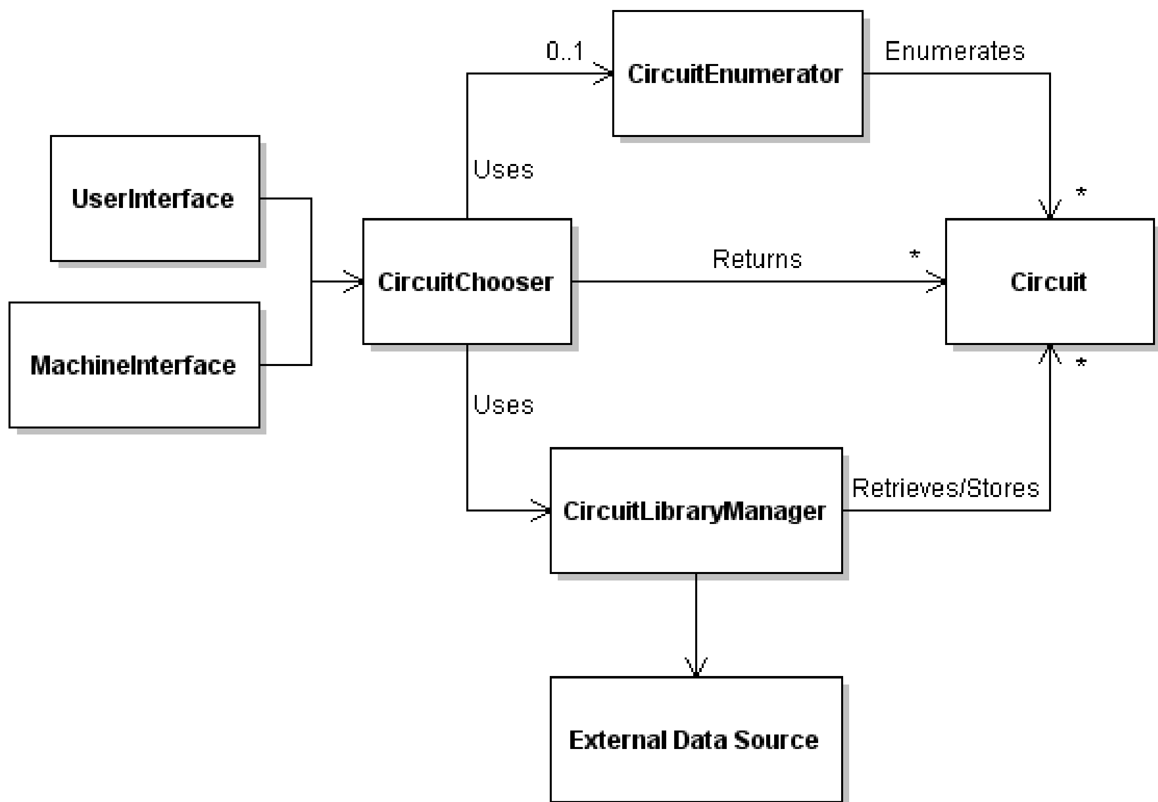

Figure A.2: Domain model

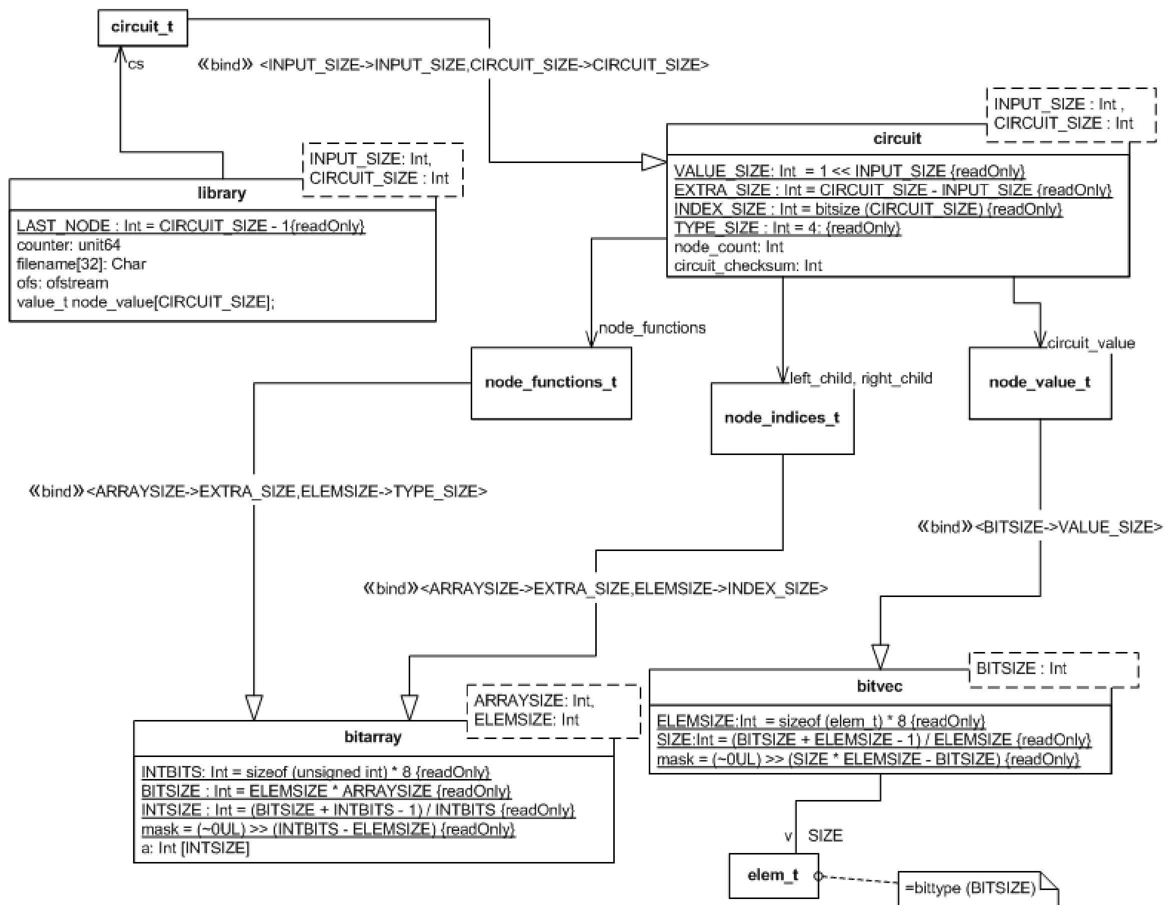

Figure A.3: Legacy CXL Class Diagram

```
    void generate_all (int unused, int node)
    {
     if (node > INPUT_SIZE)
     {
...
        // save generated circuit
        if ((unused == 0) || (node >= CIRCUIT_SIZE))
        {
        save_circuit ();
        }
...
     }
...
    }

    void save_circuit ()
    {
...
    cout << cs.get_value() << " " << cs << endl;
... }
```

Figure A.4: Portion of legacy source code

set characteristics defined in Chapter III. We implemented this component in the `CircuitType` class (see Figure A.5).

A.2.2 Circuit enumerator. Once Circuit and all related classes are established, the `CircuitEnumerator` class (Figure A.6), designed to enumerate all possible circuits of a given type is fairly simple.

The algorithm is essentially a multi-threaded version of the algorithm defined in Chapter III. Multi-threading allows the algorithm to take advantage of the multi-processor capabilities of modern computers.

The astute reader may note the use of an observer and realize that this allows much flexibility in exactly what the program will do with data generated by the circuit generator.

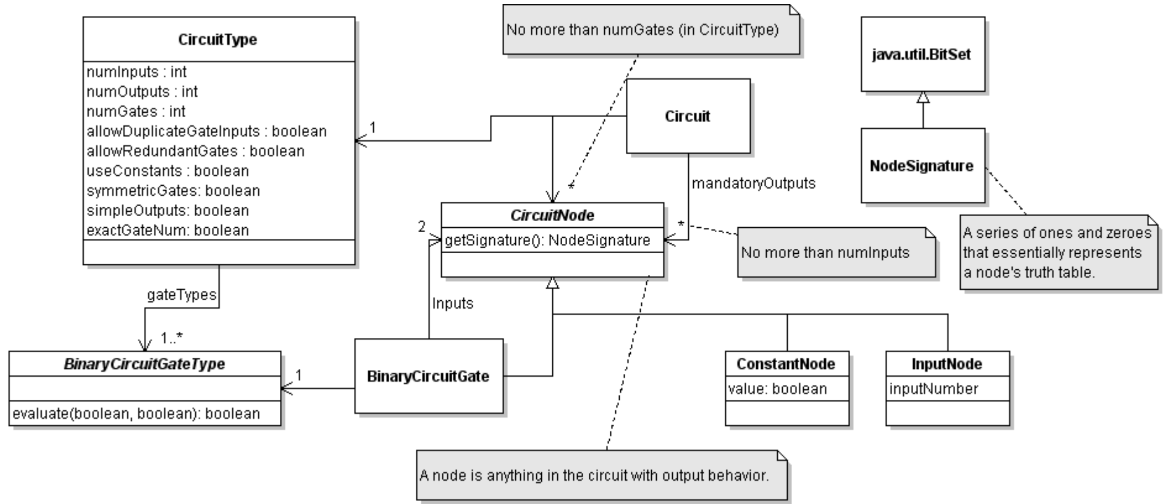

Figure A.5: The initial circuit class diagram

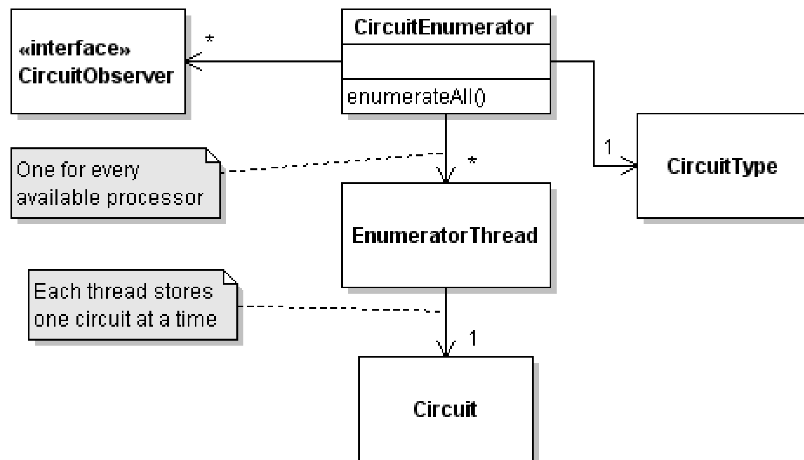

Figure A.6: Circuit enumerator class diagram

```
G3: 10001000 G2: 11110111 G0=I0==I1,G1=I0!&I1,G2=G1|I2,G3=G0&I1
G3: 10111011 G1: 10011001 G0=I0&I1,G1=I0==I1,G2=I0!&G0,G3=I1^G2
G3: 11111111 G1: 10011001 G0=I0&I1,G1=I0==I1,G2=I0!&G0,G3=I1|G2
G3: 00000000 G1: 10011001 G0=I0&I1,G1=I0==I1,G2=I0!&G0,G3=I1!|G2
G3: 01101111 G2: 01110111 G0=I0&I1,G1=I0==I1,G2=I0!&G0,G3=I2!&G1
G3: 10010110 G2: 01110111 G0=I0&I1,G1=I0==I1,G2=I0!&G0,G3=I2==G1
G3: 10010000 G2: 01110111 G0=I0&I1,G1=I0==I1,G2=I0!&G0,G3=I2&G1
G3: 01101001 G2: 01110111 G0=I0&I1,G1=I0==I1,G2=I0!&G0,G3=I2^G1
G3: 01111111 G0=I0!&I1,G1=I0!&I1,G2=G1!|I2,G3=G0^G2 G3: 01010101
G2: 11110111 G0=I0==I1,G1=I0!&I1,G2=G1|I2,G3=G0^I1 G3: 01111111
G0=I0!&I1,G1=I0!&I1,G2=G1!|I2,G3=G0|G2 G3: 11111001 G2: 01110111
G0=I0&I1,G1=I0==I1,G2=I0!&G0,G3=I2|G1 G3: 11011101 G2: 11110111
G0=I0==I1,G1=I0!&I1,G2=G1|I2,G3=G0|I1 G3: 10000000
G0=I0!&I1,G1=I0!&I1,G2=G1!|I2,G3=G0!|G2 G3: 10001000 G2: 00001000
G0=I0!&I1,G1=I0!&I1,G2=G1!|I2,G3=G1!&G0
```

Figure A.7: Part of the output of PrintCircuits running in unfiltered mode for a 3 input, 2 output, maximum of 4 gate circuit

A.3 User interface

The `PrintCircuits` class was designed as a command-line user interface. It is a simple application that registers an observer with the circuit generator in order to display information about generated circuits. It is capable of operation in two modes: filtered and unfiltered.

Unfiltered mode simply displays all circuits that match specified type along with the signatures of all of their open outputs. See Figure A.7 for an example. Note that some circuits have 1 open output and some have 2.

Filtered mode, on the other hand, displays only circuits where the specified outputs signature matches all open outputs and any remaining specified signatures match a gate within the circuit. See Figure A.8 for an example. Note that the program only displays circuits that can satisfy the specified signatures.

A.4 Development of persistence layer

The legacy software included persistence in the form of generated flat files. However, to ensure flexibility in types of persistence, this capability was defined as

```
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=I0==G0,G3=G2|G1
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=I0&G0,G3=G0|G1
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=I0&G0,G3=G1|G0
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=I0&G0,G3=G1|G2
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=I0&G0,G3=G2|G1
G3: 01111111 G1: 00100010 G0=I0==I1,G1=I1!|G0,G2=G1^I0,G3=I2!&G2
G3: 01111111 G1: 00100010 G0=I0==I1,G1=I1!|G0,G2=G1^I0,G3=G2!&I2
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=G0==I0,G3=G0|G1
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=G0==I0,G3=G1|G0
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=G0==I0,G3=G1|G2
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=G0==I0,G3=G2|G1
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=G0&I0,G3=G0|G1
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=G0&I0,G3=G1|G0
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=G0&I0,G3=G1|G2
G3: 01111111 G2: 00100010 G0=I0!&I1,G1=I2!&I0,G2=G0&I0,G3=G2|G1
```

Figure A.8: Part of the output of `PrintCircuits` running in filtered mode for a 3 input, 2 output, maximum of 4 gate circuit

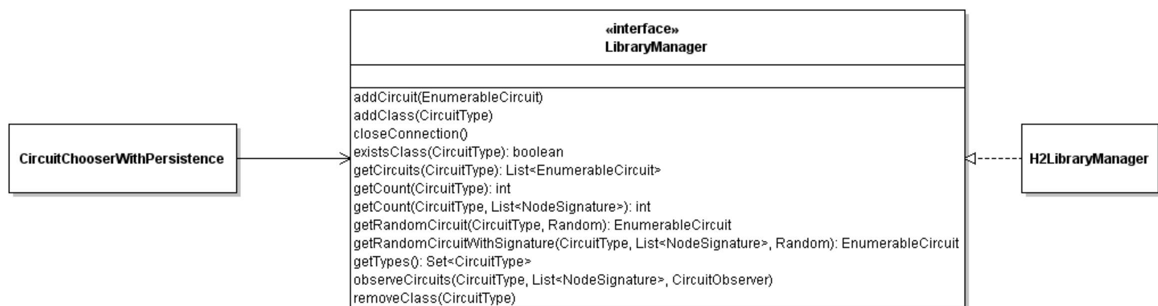

Figure A.9: The persistent circuit chooser and library manager

in interface in jCXL (see Figure A.9). We developed a sample implementation using the open source H2 Database because of claims that it was faster than other database products (see Figure A.10). However, the overhead associated with a database ensures that this implementation is not optimal in terms of speed and leaves open the possibility of a more efficient implementation in the future.

A.5 Development of machine interface

We developed a simple machine interface to work with Ken Norman's replacement algorithm known as "CORGI" [17]. The meat of this interface is an algorithm

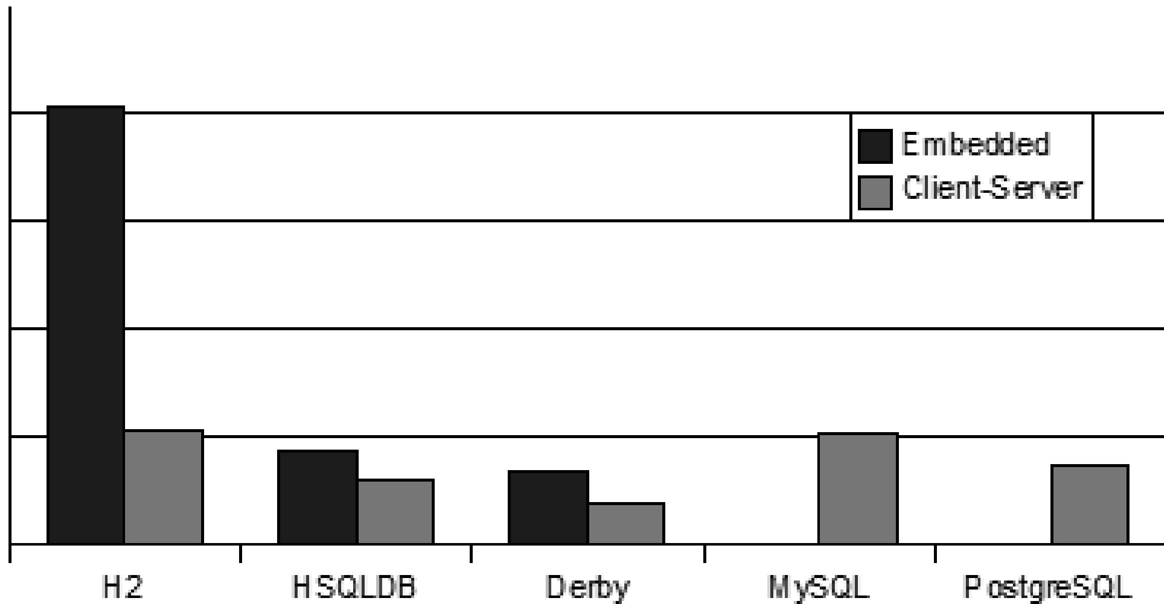

Figure A.10: Developer-provided operations/second benchmarks for H2 (Source: [9])

that converts the static, easily enumerable circuit type used by jCXL to the graph-based circuit type used by CORGI.

A.6 Refinement of algorithm runtime

Using a powerful performance analysis tool known as the Eclipse Test & Performance Tools Platform (TPTP) [5], we were able to locate and optimize some bottlenecks in the software. TPTP revealed that over half of the runtime of the enumeration software involved the `edu.afit.pet.cxl.nodes.gates.EnumerableGate-.generateSignature` method. A more in depth analysis revealed that inefficient representations of data and algorithms in the `NodeSignature` and various `Binary-CircuitGateType` classes were the cause of the slowdown. By changing the internal setup of `NodeSignature` and the `BinraryCircuitGateType` abstract class to use bitwise arithmetic operations (see Figure A.11, the total runtime of the `generate-Signature` method was reduced to less than 1/5 of the total runtime of the algorithm.

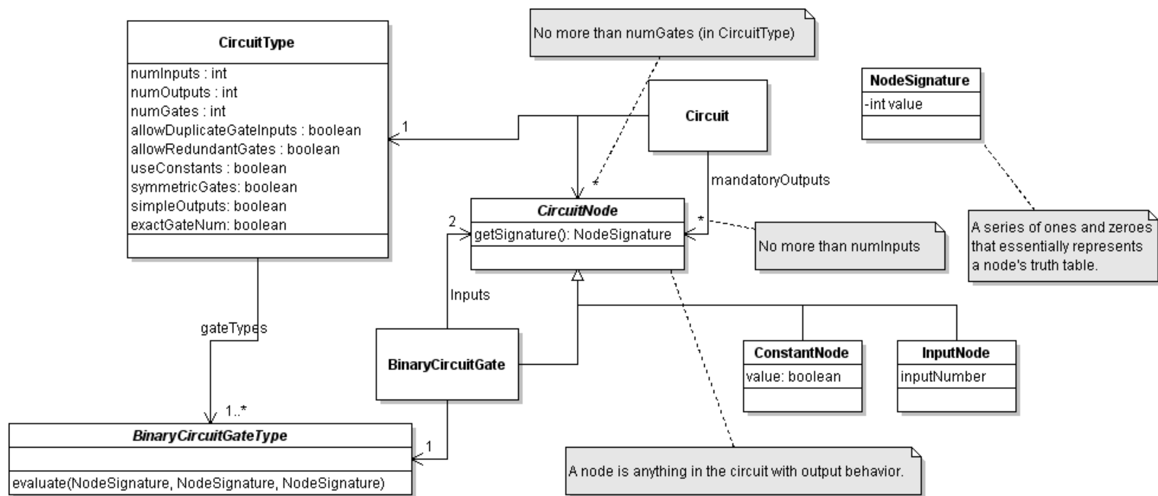

Figure A.11: The more efficient circuit class diagram

A.7 Software Testing

We used the JUnit[1] automated testing framework to conduct unit testing on a large number of features of the program, including verification of correctness of enumerated circuits and CORGI conversion.

[1]see http://www.junit.org

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
27–03–2008	Master's Thesis	Sept 2006 — Mar 2008

4. TITLE AND SUBTITLE

Obfuscation Framework Based on Functionally Equivalent Combinatorial Logic Families

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Moses C. James, Capt, USAF

5d. PROJECT NUMBER

08-183

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/08-12

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Dr. Robert L. Herklotz
Program Manager: Security and Information Operations AFOSR
Suite 325, Room 3112
875 N. Randolph Street Arlington, VA 22203-1768
email- robert.herklotz@afosr.af.mil
(703) 696-6565 fax (703) 696-8450

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This thesis aims to be a few building blocks in the bridge between theoretical and practical software obfuscation that researchers will one day construct. We provide a method for random uniform selection of circuits based on a functional signature and specific construction specifiers.
Additionally, this thesis includes the first formal definition of an algorithm that performs only static analysis on a program; that is analysis that does not rely on the input and output behavior of the analyzed program. This is analogous to some techniques used in real-world software reverse engineering.
Finally, this thesis uses the equivalent circuit library to empirically produce some statistical data about enumerated circuit families and explains how this data may be useful to future researchers.

15. SUBJECT TERMS

software obfuscation, information theory, cryptography, software engineering, software (computers)

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Lt Col J. Todd McDonald
					19b. TELEPHONE NUMBER *(include area code)*
U	U	U	UU	66	(937) 255–3636, jeffrey.mcdonald@afit.edu