Algorithms for White-box Obfuscation
Using Randomized
Subcircuit Selection and Replacement

Thesis
Kenneth E. Norman, Major, USAF
AFIT/GCS/ENG/08-17

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 expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.
Algorithms for White-box Obfuscation Using Randomized Subcircuit Selection and Replacement

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 in Computer Science

Major, USAF

27 March 2008

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
Algorithms for White-box Obfuscation Using Randomized Subcircuit Selection and Replacement

Major, USAF

Approved:

/signed/ 27 Feb 2008
__________________________
Lt Col J. Todd McDonald, Ph.D. (Chairman) Date

/signed/ 27 Feb 2008
__________________________
Dr. Yong C. Kim (Member) Date

/signed/ 27 Feb 2008
__________________________
Lt Col Stuart H. Kurkowski, Ph.D. (Member) Date
Abstract

Software protection remains an active research area with the goal of preventing adversarial software exploitation such as reverse engineering, tampering, and piracy. Heuristic obfuscation techniques lack strong theoretical underpinnings while current theoretical research highlights the impossibility of creating general, efficient, and information theoretically secure obfuscators. In this research, we consider a bridge between these two worlds by examining obfuscators based on the Random Program Model (RPM). Such a model envisions the use of program encryption techniques which change the black-box (semantic) and white-box (structural) representations of underlying programs.

In this thesis we explore the possibilities for white-box transformation. Under an RPM formulation, if an adversary cannot distinguish an original program from either its obfuscated version (whose black-box behavior has been strategically altered) or a randomly generated program of comparable size, then the white-box intent of the original program has been sufficiently protected. One proposed method of creating such random indistinguishability is by choosing (at random) a program from a size-bounded set of all semantically equivalent possibilities.

Since full enumeration of reasonably sized programs is not possible, in this work we focus on obfuscators which introduce random white-box structural variation based on iterative selection and replacement. We design and develop an obfuscation framework for programmatic logic expressed as combinatorial Boolean circuits and compare six unique approaches for sub-circuit selection. We analyze the relative behavior of random and guided-random sub-circuit selection algorithms while showing their utility in producing random white-box structural variation.
Acknowledgements

To my wife and son: Thank you for your love and support. My success is equally yours, and for your sacrifices, I owe you more than I can ever repay. I love you both very much.

Professionally, I owe a debt of gratitude to my thesis advisor, Lt Col Todd McDonald, and my research partner, Capt Moses James. As an electrical engineer in a computer science program, I know I taxed their patience with my many questions. Thank you.

Kenneth E. Norman
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>I. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem area</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Motivating scenario</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 Context</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Research objectives</td>
<td>4</td>
</tr>
<tr>
<td>II. Literature Review</td>
<td>5</td>
</tr>
<tr>
<td>2.1 What is obfuscation?</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Preliminary definitions</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2 Classifications of obfuscation</td>
<td>5</td>
</tr>
<tr>
<td>2.1.3 Theoretical definitions</td>
<td>6</td>
</tr>
<tr>
<td>2.1.3.1 Virtual Black Box Obfuscation</td>
<td>7</td>
</tr>
<tr>
<td>2.1.3.2 Indistinguishability Obfuscation</td>
<td>8</td>
</tr>
<tr>
<td>2.1.3.3 Best-Possible Obfuscation</td>
<td>9</td>
</tr>
<tr>
<td>2.1.4 Practical applications</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Shortfalls of current theoretical work</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Random Program Security Model</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1 Program encryption</td>
<td>12</td>
</tr>
<tr>
<td>2.3.2 Intent protection</td>
<td>14</td>
</tr>
<tr>
<td>III. Methodology</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Notation</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Assumptions</td>
<td>15</td>
</tr>
<tr>
<td>3.2.1 Programs represented as circuits</td>
<td>15</td>
</tr>
<tr>
<td>3.2.1.1 Combinational circuits</td>
<td>15</td>
</tr>
<tr>
<td>3.2.1.2 Directed acyclic multi-graphs</td>
<td>16</td>
</tr>
<tr>
<td>3.2.2 Iterative randomization</td>
<td>18</td>
</tr>
<tr>
<td>3.2.3 Circuit library exists</td>
<td>21</td>
</tr>
<tr>
<td>3.3 Obfuscation toolkit</td>
<td>23</td>
</tr>
<tr>
<td>3.3.1 CORGI: the circuit randomizer</td>
<td>23</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.3.1.1 Development environment</td>
<td>23</td>
</tr>
<tr>
<td>3.3.1.2 Subcircuit selection and replacement</td>
<td>24</td>
</tr>
<tr>
<td>3.3.2 CXL: the circuit library</td>
<td>25</td>
</tr>
<tr>
<td>3.4 Empirical Approach</td>
<td>26</td>
</tr>
<tr>
<td>3.4.1 Key concepts</td>
<td>26</td>
</tr>
<tr>
<td>3.4.2 Properties of obfuscated circuits</td>
<td>28</td>
</tr>
<tr>
<td>3.4.3 White-box obfuscation algorithms</td>
<td>30</td>
</tr>
<tr>
<td>IV. Results</td>
<td>32</td>
</tr>
<tr>
<td>4.1 Overview</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Limitations</td>
<td>33</td>
</tr>
<tr>
<td>4.2.1 Smart strategies</td>
<td>33</td>
</tr>
<tr>
<td>4.2.2 Introduced cycles</td>
<td>34</td>
</tr>
<tr>
<td>4.3 Analysis of subcircuit selection algorithms</td>
<td>35</td>
</tr>
<tr>
<td>4.3.1 Common functions</td>
<td>36</td>
</tr>
<tr>
<td>4.3.2 RandomSingleGate</td>
<td>36</td>
</tr>
<tr>
<td>4.3.3 RandomTwoGates</td>
<td>38</td>
</tr>
<tr>
<td>4.3.4 RandomLevelTwoGates</td>
<td>45</td>
</tr>
<tr>
<td>4.3.5 FixedLevelTwoGates</td>
<td>47</td>
</tr>
<tr>
<td>4.3.6 LargestLevelTwoGates</td>
<td>49</td>
</tr>
<tr>
<td>4.3.7 OutputLevelTwoGates</td>
<td>50</td>
</tr>
<tr>
<td>4.4 Runtime performance analysis</td>
<td>52</td>
</tr>
<tr>
<td>V. Conclusions</td>
<td>67</td>
</tr>
<tr>
<td>5.1 Contributions</td>
<td>67</td>
</tr>
<tr>
<td>5.2 Future work</td>
<td>68</td>
</tr>
<tr>
<td>Appendix A.</td>
<td>70</td>
</tr>
<tr>
<td>CORGI software</td>
<td>70</td>
</tr>
<tr>
<td>A.1 CORGI architecture</td>
<td>70</td>
</tr>
<tr>
<td>A.1.1 Functionality</td>
<td>70</td>
</tr>
<tr>
<td>A.1.1.1 JGraphT</td>
<td>70</td>
</tr>
<tr>
<td>A.2 Non-selection algorithms</td>
<td>70</td>
</tr>
<tr>
<td>A.3 Selection algorithm behavior</td>
<td>74</td>
</tr>
<tr>
<td>A.4 Selection algorithm results</td>
<td>78</td>
</tr>
<tr>
<td>A.4.1 C17 with all algorithms</td>
<td>78</td>
</tr>
<tr>
<td>A.4.2 C880 with OutputLevelTwoGates</td>
<td>78</td>
</tr>
<tr>
<td>Bibliography</td>
<td>84</td>
</tr>
<tr>
<td>Vita</td>
<td>86</td>
</tr>
<tr>
<td>Index</td>
<td>Index-1</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Program Encryption</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>The Random Program Model</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>RPM obfuscation</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Black box obfuscated program</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>The Random Program Model</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>ISCAS Benchmark Circuit C17</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Graph examples</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Iterative randomization</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Circuit hierarchy example</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Example histogram</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Improper subcircuit selection creates cycles</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Introduced control flow in ISCAS C17</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Diffusion of replacements in ISCAS C17</td>
<td>43</td>
</tr>
<tr>
<td>4.4</td>
<td>How a replacement subcircuit creates a new control flow</td>
<td>44</td>
</tr>
<tr>
<td>4.5</td>
<td>Runtime data for R1G on C17</td>
<td>55</td>
</tr>
<tr>
<td>4.6</td>
<td>Runtime data for R1G on C880</td>
<td>56</td>
</tr>
<tr>
<td>4.7</td>
<td>Runtime data for R2G on C17</td>
<td>57</td>
</tr>
<tr>
<td>4.8</td>
<td>Runtime data for R2G on C880</td>
<td>58</td>
</tr>
<tr>
<td>4.9</td>
<td>Runtime data for RL2G on C17</td>
<td>59</td>
</tr>
<tr>
<td>4.10</td>
<td>Runtime data for RL2G on C880</td>
<td>60</td>
</tr>
<tr>
<td>4.11</td>
<td>Runtime data for FL2G on C17</td>
<td>61</td>
</tr>
<tr>
<td>4.12</td>
<td>Runtime data for FL2G on C880</td>
<td>62</td>
</tr>
<tr>
<td>4.13</td>
<td>Runtime data for LL2G on C17</td>
<td>63</td>
</tr>
<tr>
<td>4.14</td>
<td>Runtime data for LL2G on C880</td>
<td>64</td>
</tr>
<tr>
<td>4.15</td>
<td>Runtime data for OL2G on C17</td>
<td>65</td>
</tr>
</tbody>
</table>
Figure | Page
--- | ---
4.16 Runtime data for OL2G on C880 | 66
A.1 CORGI UML class diagram | 71
A.2 Behavior data for all six selection algorithms | 75
A.3 Chart of behavior data: circuit height | 76
A.4 Chart of behavior data: circuit width | 77
A.5 Sample results of R1G and OL2G algorithms | 79
A.6 Sample results of R2G and FL2G algorithms | 80
A.7 Sample results of RL2G and LL2G algorithms | 81
A.8 Sample result 1 of OL2G applied to C880 | 81
A.9 Sample result 2 of OL2G applied to C880 | 82
A.10 Sample result 3 of OL2G applied to C880 | 83
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Notation for the Random Program Model</td>
</tr>
<tr>
<td>3.2</td>
<td>Features and benefits of JGraphT</td>
</tr>
<tr>
<td>3.3</td>
<td>Candidate circuit properties</td>
</tr>
<tr>
<td>3.4</td>
<td>Candidate subcircuit selection algorithms</td>
</tr>
<tr>
<td>4.1</td>
<td>Summary of runtime data</td>
</tr>
</tbody>
</table>
I. Introduction

Across the Department of Defense, it is increasingly difficult to find a weapon systems which does not rely upon software to perform its intended function. The United States Air Force in particular is reliant on software across every facet of its mission: air, space, and cyberspace. The ubiquity of software-based systems, and the interconnectedness of such systems, demands we protect them from our adversaries’ prying eyes. In many cases, physical security is sufficient to thwart anyone who seeks to gain access to our systems. When physical security fails to protect our critical software, we must turn to alternate means. One such alternative is software obfuscation.

1.1 Problem area

Software obfuscation is not a new concept, but neither is it a well-defined discipline in practice. The concept of software obfuscation is in many ways the unraveling of sound development principles. The objective in software engineering is to produce systems which are defect-free, modular, maintainable, and extensible. A well-engineered system will function as efficiently as possible, and perform the job the user expects, in the manner he expects it. The objective in software obfuscation is to produce highly coupled, difficult-to-understand, complex systems which, nevertheless, perform the job the user expects, in the manner he expects it (though perhaps with less efficiency by comparison).

1.1.1 Motivating scenario. In early 2001, the world watched as the US and China found themselves at odds after what became known as the Hainan Island
incident. In brief, a US EP-3 reconnaissance plane and a Chinese Shenyang J-8 collided, and the EP-3 was forced to make an emergency landing on Hainan Island off the south coast of China. According to a 2 April 2001 UPI press release [11],

“[t]he EP-3 could not have landed in a better place for China or a worse one for U.S. military intelligence. Hainan island is host to one of China’s largest electronic signals intelligence complexes and is manned by experts who can glean critical information on the aircraft’s capabilities if they gain access to the Navy’s EP-3” . . . Pentagon sources said.

The crew was held hostage for 12 days before being released. The plane, however, remained on Hainan Island for a total of 94 days, during which time China had unfettered access to the equipment on board. If the EP-3 crew was unable to entirely destroy all information storage devices (and the software they contain) before they landed, then the Chinese had ample opportunity to learn about US collection methods and targets of interest during the time the plane was in their control. Even if their examination would have taken more than 94 days, it would have been easy enough to copy the code (from undamaged equipment) and analyze it after they returned the aircraft to US custody.

1.1.2 Context. This research augments earlier work initiated by Lt Col Todd McDonald for his doctorate degree. In his dissertation, McDonald described software obfuscation as protecting program intent [12]. The concept of intent protection stands in contrast to traditional definitions of obfuscation, all of which require that a program’s functionality remain unchanged (without regards to some acceptable degradation of time and/or space efficiency). Instead, McDonald takes inspiration from the field of cryptography and likens intent protection to data encryption. The idea is to transform a program in two ways—structurally and functionally. If functionality (that is, input/output behavior) must change, then it must also be possible to recover the original behavior (see Figure 1.1). McDonald further requires that an intent protected program be indistinguishable from any other program, selected ran-
domly, which has a similar number of inputs, outputs, and is of similar size. This he calls the Random Program Model (RPM).

The difficult question is how to devise a random selection schema. Clearly, for any but the most basic of programs, software can be written in almost limitless ways to accomplish the same function. If the set is impossible (or at least infeasible) to create, an alternate means of “selection” is required.

Rather than attempt to enumerate entire sets of programs, then select a replacement in toto, we consider an alternate approach of iterative randomization. This process obfuscates a program by changing the structure of only a small portion of the program per iteration, but many iterations produce a randomized program.

For this nascent research, we narrow our focus to combinational Boolean circuits. This simplifies the problem domain by avoiding non-terminating programs and program state (memory). Additionally, circuits can be modeled using constructs from the mathematical discipline of graph theory.
1.2 Research objectives

We seek to accomplish two objectives with this research.

1. Develop a software architecture for developing and testing random selection schema for obfuscating a circuit’s structure.

2. Develop an initial set of selection algorithms and characterize their behavior with regards to white-box obfuscation.

The first objective above is a means to an end. In other words, to develop and analyze selection algorithms, we need an architecture which will import, export, and manipulate combinational Boolean circuits. No complete application is available to perform the operations we seek to employ, so we developed a software package (CORGI) to fill the void. Although CORGI is all new, it integrates an existing Java library (JGraphT) to represent the circuits as directed acyclic graphs.

For the second objective, we devised candidate algorithms which demonstrate the concept of random selection and replacement. The algorithms each produce an obfuscated version of an original circuit. Each circuit produced in this way is a randomly “selected,” semantically equivalent version of the original, with the selection occurring as a sequence of steps rather than a single-step selection from a large set.

Although this research is based on a new obfuscation paradigm, the next chapter explores the current theoretical understanding of obfuscation and how it relates to our current work.

\[1\] CORGI stands for Circuit Obfuscation via Randomization of Graphs Iteratively, and is discussed in more detail in Section 3.3.1.
II. Literature Review

Several key papers have been published which provide theoretical bases for why obfuscation is both impossible and, indeed, possible. Practical applications of these theories, however, do not appear in the literature. As such, one approach, the Random Program Security Model, proposes that practical obfuscation is indeed possible and that a program’s intent can be protected even if the adversary has access to the obfuscated version of the program. The Random Program Security Model is fundamentally an analog to data encryption, but applied to programs rather than data.

2.1 What is obfuscation?

2.1.1 Preliminary definitions. Before delving into the finer details of obfuscation, it is instructive to understand how the word obfuscation is used in several contexts. In generic speech, to obfuscate means to “make obscure” or “confuse” [13].

As applies to computing, to obfuscate means “to alter code while preserving its behavior but conceal its structure and intent” [19]. Alternately, obfuscation is “any efficient semantic-preserving transformation of computer programs aimed at bringing a program into such a form, which impedes the understanding of its algorithm and data structures or prevents the extracting of some valuable information from the plaintext of a program” [18]. These two definitions provide the context for our review of current theory and techniques for program obfuscation.

2.1.2 Classifications of obfuscation. Program development and execution involves several steps, and program obfuscation can be applied at one or more of these steps. Fundamentally, there are three classifications of program obfuscation: layout, data, and control [3]. Layout obfuscation involves such techniques as scrambling identifier names and removing layout formatting. Both of these techniques operate on the source code, and do nothing to alter control flow of the program.
Data obfuscation is also primarily focused on altering the source code. Techniques include (a) storage and encoding transformations, which alter the way data is encoded or manipulated (b) aggregation transformations, which operate on data structures, and (c) ordering transformations, which change the order of variables and methods (within classes) and parameters (within methods). To some extent, these techniques can have an impact on control flow within a program, but it is not the primary intent. Like layout obfuscation, many of the specific transformations do not change control flow (although some introduce new control mechanisms).

The final classification is control obfuscation, and its techniques include (a) control aggregation transformations, which break up computations that logically belong together or merge computations that do not, (b) control ordering transformations, which randomize the order in which computations are carried out, and (c) control computation transformations, which insert new (redundant or dead) code, or make algorithmic changes to the source application. Control obfuscation techniques, as described in [3], are not strictly limited to source code, which means it has more generic applicability (e.g., assembly language and machine code).

Among the three broad categories described above, general program (circuit) obfuscation must account for control flow. This becomes clear as we look at additional definitions of obfuscation.

2.1.3 Theoretical definitions. The first formalized theoretical definition of program (or circuit) obfuscation was introduced by Barak et al. in [1]. This was a watershed publication because it formally proved that universal obfuscators do not exist. It also had the effect of spawning alternate theoretically-based definitions of obfuscation in several publications which followed. We will look at several of these definitions here.
2.1.3.1 Virtual Black Box Obfuscation. “Informally, an obfuscator $\mathcal{O}$ is an (efficient, probabilistic) compiler that takes as input a program $P$ (or circuit $C$) and produces a new program $\mathcal{O}(P)$ that has the same functionality as $P$ yet is unintelligible in some sense” [1]. In lay terms, virtual black box (VBB) obfuscation can be thought of as some transformation to a program which completely hides all information about the program except input/output (i.e., black box) behavior, even though the obfuscated program is itself observable. In that sense, the obfuscated version provides virtually equivalent information as could be obtained with only black box access to the program.

Although informal, the definition above makes no distinction of what constitutes a program. No mention is made of “source code,” “assembly language,” or “machine code” anywhere in the paper (save one quote in a footnote). Thus, while there are clear differences between the three levels of a program, their fundamental nature is the same. Indeed, their equivalence is evidenced by the fact that programs can be viewed as boolean (specifically, combinational) logic circuits, and the Barak paper uses the terms program and circuit almost interchangeably. This is not to imply that obfuscated source code will necessarily yield object code that is obfuscated to the same degree (however measured). This remains an open question which, in part, will be addressed by this thesis.

Barak et al. formally define a (circuit) obfuscator as having these three properties:

1. **Functionality** property: For every circuit $C$, $\mathcal{O}(C)$ describes a circuit that computes the same function as $C$.

2. **Polynomial slowdown** property: There is a polynomial $p$ such that for every circuit $C$, $|\mathcal{O}(C)| \leq p(|C|)$. This property may apply to size, run time, or both. 

---

1Since this concept applies equally to programs and circuits, and since this thesis will specifically explore obfuscation of circuits, we will limit further discussion to circuit obfuscation. Therefore, substituting $C$ for $P$ does not alter the definition.
3. “Virtual black box (VBB)” property: For any probabilistic polynomial-time Turing machine (PPT) $A$, there is a PPT $S$ and a negligible function $\alpha$ such that for all circuits $C$,

$$|\Pr[A(O(C)) = 1] - \Pr[S^C(1^{\|C\|}) = 1]| \leq \alpha(|C|)$$ (2.1)

The obfuscator $O$ is efficient if it runs in polynomial time.

From this definition, Barak, et al. prove that no universal obfuscator exists. The basis of their proof is to show that, for any given obfuscator, there exists a family of circuits which cannot be obfuscated. “However, it does not mean that there is no method of making circuits ‘unintelligible’ in some meaningful and precise sense” [1]. To be clear, the impossibility result still allows for a given obfuscator $O$ to be able to protect some (though not all) families of circuits $C$. From this, Barak et al. offer a weaker notion of obfuscation: indistinguishability obfuscation.

**2.1.3.2 Indistinguishability Obfuscation.** An indistinguishability obfuscator is defined in the same way as a circuit obfuscator, except that the “virtual black box” property is replaced with the following:

- **Indistinguishability** property: For any PPT $A$, there is a negligible function $\alpha$ such that for any two circuits $C_1, C_2$ which compute the same function and are of the same size $k$,

$$|\Pr[A(O(C_1))] - \Pr[A(O(C_2))]| \leq \alpha(k)$$ (2.2)

Observe that the indistinguishability property compares the obfuscations of two different circuits, unlike the VBB property, which compares an obfuscated circuit to a simulator which has only black box access to the original circuit. By weakening the VBB definition in this way, it is provable that obfuscation (however inefficient) is not impossible.
2.1.3.3 Best-Possible Obfuscation. Goldwasser and Rothblum define an obfuscator as “a compiler that transforms any program (which we will view... as a boolean circuit) into an obfuscated program (also a circuit) that has the same input-output functionality as the original program, but is unintelligible” [6]. It is clear that this is the same definition found in [1], but it is nevertheless included because of the parenthetical comment that programs can be viewed as circuits. This concept is central to the research presented herein.

2.1.4 Practical applications. Obfuscation software, of varying sophistication, is widely available from both commercial vendors and open source developers. Among commercial products, there are several well-known titles. PreEmptive Solutions [16] produces two popular tools: Dotfuscator (for .NET) and DashO (for Java). Smardec [17], produces Allatori, a Java obfuscator. Yet another company, Semantic Designs, Inc. [15] has a suite of tools collectively called Thicket™. It provides tools to obfuscate several languages, including C, C++, C#, Java, JavaScript, Ada, and PHP. There are, of course, other vendors which offer products that purport to obfuscate software to some degree, but enumerating them all here is beyond the scope of this thesis.

On the open source side, the number of projects is as plentiful as on the commercial side. One in particular, ProGuard Java Optimizer and Obfuscator is one of the most popular projects on SourceForge.net.²

It is not surprising that these companies and open source developers reveal little about the inner workings of their obfuscation techniques, except to describe the results of applying a particular approach (e.g., name obfuscation, flow obfuscation, string encryption, etc.). Interestingly, however, Semantic Designs’ web site unequivocally states, “Warning: obfuscators do not stop reverse-engineering efforts by really determined opponents.” This statement is an acknowledgment of the theoretical work

---

²From its home page, “SourceForge.net is the world’s largest Open Source software development web site.” As of 16 Jan 2008, ProGuard was ranked 291 out of 166,996 projects listed.
of Barak et al. described above. Nevertheless, practical obfuscators are not in short supply, despite this limitation, which begs the question: “Why not?”

2.2 Shortfalls of current theoretical work

To begin to answer the question of why practical software obfuscators are even available, much less trusted, one must further ask, “what makes them useful despite the impossibility results asserted—‘proved’—by the theoreticians?” The answer is at least two-fold.

First, commercial and open source obfuscation tools are not typically employed, for the most part, to hide the purpose of the target software, but rather to hide the manner in which that purpose is achieved. For example, Microsoft may choose to obfuscate all or part of the source code for its spreadsheet program, Excel™. The obfuscated version would not hide the fact that the application is a spreadsheet. Rather, it would hide some portion of the code to prevent competitors from learning how part of the code is implemented, thus protecting Microsoft’s competitive advantage in the marketplace. In this way, the obfuscation would be useful, even if though it necessarily fails the VBB paradigm of perfectly secure obfuscation.

A second (perhaps more profound) reason may be that the tools do not address obfuscation from a theoretical perspective. In light of an absence in the literature that correlates theoretical results to practical implementations, it is difficult to make this claim definitively (i.e., “absence of proof is not proof of absence”). It is nonetheless intriguing that developers do not relate the strength of their obfuscation schema to results predicted by the theoretical models.

From a VBB perspective, no obfuscators of any ilk should be useful or beneficial. Although the VBB standard is not achievable in a general, efficient, universal sense, some amount of obfuscation, as pertains to some as-yet undefined metric of obfuscation, may be desirable. This is certainly the case with existing obfuscators, even if not explicitly stated or understood by the developers, because all such tools
both exist and fail the VBB test. Therefore, the VBB standard is not viable as a measure of practical obfuscation.

The other two theoretical results mentioned before—indistinguishability obfuscation and best-possible obfuscation—are similar. They both relate obfuscation to some property of the program, and use that to compare obfuscation results to each other (whereas VBB relates obfuscation to a black box version of a program). This distinction is subtle, but it opens the door to finding useful obfuscators even if they fail VBB scrutiny. Unfortunately, the underpinning theory behind indistinguishability obfuscation and best-possible obfuscation do not offer suggestions on what property or properties of a program should be the basis of comparison when deciding if an obfuscator yields indistinguishable results, or the best-possible level of obfuscation.

The research supporting this thesis was conducted to directly address what properties of a program might (or might not) be useful measures of obfuscation, and to provide a framework for empirically testing the efficacy of those properties. In other words, we seek to produce a “tangible” correlation to the theoretical work which has preceded this research. This objective is an outgrowth of the doctorate research conducted by Lt Col Todd McDonald. In his dissertation, he suggests a new paradigm of program obfuscation, the Random Program Security Model [12].

2.3 Random Program Security Model

Recall from [1] the theoretical benchmark definition of an obfuscator—the VBB paradigm—requires that three properties hold: functionality, polynomial slowdown, and the VBB property. Under the Random Program Security Model (or simply Random Program Model, RPM), McDonald replaces two of the three properties, functionality and VBB [12]. Only the polynomial slowdown property is retained.

For the functionality property, McDonald postulates instead that program obfuscation should apply both black-box and white-box obfuscation techniques. The principle is that neither approach on its own is sufficient to obfuscate a program.
When combined, however, they act synergistically to overcome the inherent weaknesses of each.

For the VBB property, McDonald reasons that if an obfuscated program is indistinguishable from another program randomly-selected from the same family of programs (based on inputs, outputs, and size of the program), then the intent of the original program is protected.

The RPM is similar to, and derived from, data cryptography. RPM models black-box obfuscation after data encryption, and white-box obfuscation is analogous to comparing cryptographic data ciphers to random bit strings. Figure 2.1 graphically depicts the RPM. The obfuscator function, \( \mathcal{O} \), uses both black-box and white-box transforms, as shown in Figure 2.2. These are described below in Sections 2.3.1 and 2.3.2.

2.3.1 Program encryption. Figure 2.3 illustrates the concept of black box obfuscation using program encryption. For an input \( x \) to program \( P \), the result, \( P(x) \) is the unobfuscated output of \( P \). Intermediate result \( P(x) \) becomes the input
Figure 2.2: RPM obfuscation combines both black-box and white-box transforms

Figure 2.3: A black box obfuscation $P''$ of program $P$. $P$ and $P''$ are not semantically equivalent because $P''$ includes a program, $E$, which encrypts the output of $P$.

of another component, $E$, which encrypts $P(x)$ based on some key $k$. The output $E(P(x), k)$ of $E$ is the overall output of $P''$. Since $P(x) \neq E(P(x), k)$ (i.e., $P(x) \neq P''(x)$) for a given input $x$, program $P''$ is thus said to be a black-box obfuscated version of $P$.

Program encryption might be sufficient to protect a program if an adversary never obtains white-box access to the obfuscated program, $P''$. If the adversary did have white-box access, the demarcation between $P$ and $E$ would be discernible, and $P$ would be revealed independent of $E$. Thus, RPM adds white-box protection to program encryption to achieve overall protection of the program’s intent.
2.3.2 Intent protection. As previously stated, perfect, efficient, universal VBB obfuscators do not exist. If an adversary has access to an obfuscated, semantically equivalent program, the adversary will eventually be able to understand the intent of the original program. McDonald theorizes that program encryption can be augmented in such a way as to prevent an adversary from being able to isolate \( P \) from \( E \) in an encrypted program \( P'' \). The goal is to hide the fact that there is a semantics-altering component \( E \). If this is possible, then even if the adversary is able to (eventually) predict the output of \( P'' \), such output will be meaningless with respect to \( P(x) \), and program intent will remain protected.

McDonald proposes that if \( P'' \) (which is not semantically equivalent to \( P \)) is replaced with a randomly chosen—or produced—program \( P' \) (which is semantically equivalent to \( P'' \)), then \( P \) is intent protected if the following hold:

- \( P' \) is such that the adversary cannot distinguish between the functional program \( P \) and the composite encryption program \( E \)
- \( P' \) is indistinguishable from a random program selected from the set of all programs the same size as \( P' \)
III. Methodology

The Random Program Model posits that an intent-protected program is indistinguishable from any other program with the same number of inputs and outputs, and of comparable size. This thesis specifically considers the white-box obfuscation component of the RPM. In this initial research, a program is modeled as a combinational boolean circuit. The circuit is white-box obfuscated by iteratively replacing random subcircuits with randomly-chosen, semantically-equivalent replacement subcircuits. Several algorithms are considered for selecting the subcircuits, and as well as candidate metrics with which to quantify the level of obfuscation achieved.

3.1 Notation

Since this research follows earlier work conducted by Lt Col Todd McDonald, we use his notation for the sake of consistency. Table 3.1 provides the notation used in the discussion which follows.

3.2 Assumptions

The current experimental environment relies on some simplifying assumptions, which are discussed here.

3.2.1 Programs represented as circuits. Software functionality, at its most fundamental level, can be represented as a sequence of Boolean expressions. For typical programs, which include loops (for, while, etc.), sequential boolean circuits map most directly to the program structure. In general, sequential (cyclic, in graph theory parlance) circuits can be converted to combinational (acyclic) circuits. Edwards [4] offers an algorithm which performs this transformation, but warns it is inefficient for anything but trivially small circuits (his algorithm ran for 51 seconds when operating on a 281-gate circuit). Despite potential intractability when converting large sequential circuits, we choose combinational logic over sequential logic because of its comparative simplicity.
Table 3.1: Notation used in describing the Random Program Model

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>A combinational Boolean circuit</td>
</tr>
<tr>
<td>$C'_i$</td>
<td>Original circuit $C$ after $i$ iterations of randomization</td>
</tr>
<tr>
<td>$C', C'_n$</td>
<td>Original circuit $C$ after $n$-iteration randomization is finished</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>Circuit basis. $\Omega$ is a set of Boolean functions such that $\Omega \subseteq {\text{AND, NAND, OR, NOR, XOR, XNOR, NOT}}$</td>
</tr>
<tr>
<td>$C_{X,Y,S,\Omega}$</td>
<td>The class of a circuit, indicating inputs ($X$), outputs ($Y$), size ($S =$ maximum number of gates), and basis ($\Omega$)</td>
</tr>
<tr>
<td>$\delta, \delta_{X,Y,S,\Omega}$</td>
<td>Circuit family, i.e., the set containing all circuits $C_{X,Y,S,\Omega}$</td>
</tr>
<tr>
<td>$\delta_C$</td>
<td>Family of circuits semantically equivalent to $C$ ($\delta_C \subset \delta$)</td>
</tr>
</tbody>
</table>

The Random Program Model applies not only to the program domain, but to the circuit domain as well. Figure 2.1 is given again (with only a notational change) in Figure 3.1 to show the parallel between the two.

3.2.1.1 Combinational circuits. Combinational circuits have no state, whereas sequential circuits are temporal, which is to say they have memory and feedback loops (cycles). Since sequential circuits can be decomposed into combinational components, it is sufficient at the outset of this research to forgo the former in favor of the latter. As an aside, combinational circuits sidestep the issue of non-terminating programs—another complication of sequential circuits.

Our decision to use combinational circuits is supported by [9] which points out in Chapter IV that a very simple grammar is all that is needed to compute everything that can be computed by large languages like $C$ and Java. In particular, the grammar, in Backus Naur form, is shown in Equation 3.1 where $B$ represents any Boolean expression and $E$ represents any integer expression. It is because of this
underlying simplicity that any software can be mapped to combinational logic form.

\[
B ::= \text{true}|\text{false}|(\neg B)|(B\&B)|(B \parallel B)|(E < E)
\]  

(3.1)

An obvious benefit of choosing combinational logic is that it is easy to understand. As demonstrated in Equation 3.1 above, only three logic functions are necessary: NOT (\(!\)), AND (\(&\)), and OR (\(\parallel\)). There are other commonly used logic functions (namely NAND, NOR, XOR, and XNOR), but these can be represented using various combinations of NOT, AND, and OR.

Combinational logic circuits are used across a broad spectrum of applications, within both the hardware and software domains. At the 1985 International Symposium of Circuits and Systems (ISCAS), the IEEE introduced a set of benchmark circuits, which are collectively referred to as ISCAS-85 benchmark circuits. [8] They are particularly useful to our purpose, even though they were initially targeted at the hardware community. A list of these circuits can be found at [2]. The smallest of these circuits, C17, is shown in Figure 3.2.
3.2.1.2 Directed acyclic multi-graphs. In order to manipulate circuits, they must be in a format suitable for that purpose. For this research, the discipline of graph theory provides a suitable application domain. Namely, we represent circuits as *directed acyclic multi-graphs*. We turn to Gross and Yellen [7] for a brief refresher on graph theory terminology to help describe the rationale for choosing graphs to represent circuits (reference Figure 3.3).

**graph:** A *graph* $G = (V, E)$ is a mathematical structure consisting of two finite sets $V$ and $E$. The elements of $V$ are called *vertices* (or *nodes*), and the elements of $E$ are called *edges*. Each edge has a set of one or two vertices associated to it, which are called *endpoints*. [Example: All graphs in Figure 3.3.]

The authors correctly allow for edges with only one endpoint, which “is an edge that joins a single endpoint to itself.” However, such a construct in a circuit would make it sequential, not combinational. For our purposes, we only consider edges with exactly two distinct vertices. See the definition for *cycle* below.

**directed edge:** A *directed edge* is an edge, one of whose endpoints is designated as the tail, and whose other endpoint is designated as the head. An edge is said to be directed from its tail to its head.

**directed graph:** A *directed graph* (or *digraph*) is a graph each of whose edges is directed. [Example: Figures 3.3(b), (d), and (f).]
Figure 3.3: Example graphs.
(a) An undirected graph with no cycles.
(b) A directed graph with no cycles.
(c) An undirected graph with one cycle ($1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$ and $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$).
(d) A directed graph with one cycle ($1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$ only).
(e) An undirected multi-graph with one cycle.
(f) A directed acyclic multi-graph.
We must limit the graphs we use to directed graphs because in a combinational circuit, a connection between gates is always from the output of one gate to an input of another gate.

**cycle:** A *cycle* is a nontrivial closed path.\[1\]

**acyclic graph:** An *acyclic graph* is a graph that has no cycles. [Example: Figures 3.3(a), (b), and (f).]

Combinational circuits do not have any feedback loops or memory, as do sequential circuits. Therefore, only an acyclic graph can represent a combinational circuit.

**multi-edge:** A *multi-edge* is a collection of two or more edges having identical endpoints. The *edge multiplicity* is the number of edges within the multi-edge.

**multi-graph:** A *multi-graph* is a graph that may contain multi-edges. [Example: Figures 3.3(c) and (f).]

In a combinational circuit, it is permissible for the output of one gate to be connected to more than one input of another single gate. The analogous construct in graph theory is a multi-graph.

**directed acyclic graph:** A *directed acyclic graph* (DAG) is a graph that is at the same time a directed graph and an acyclic graph. It may or may not be a multi-graph. [Example: Figures 3.3(b) and (f).]

For our purposes, we implicitly accept DAGs as also being multi-graphs. In other words, DAG and directed acyclic multi-graph carry the same meaning, thus Figures 3.3(b) and 3.3(f) are both DAGs.

---

1A *path* does not repeat any vertex (except possibly the initial/final vertex) or edge. *Nontrivial* means the path includes more than one vertex. *Closed* means the initial vertex is the same as the final vertex.
3.2.2 Iterative randomization. The RPM requires that an intent-protected circuit, \( C' \), be indistinguishable from a randomly selected circuit, \( C_R \). An interesting aspect of the RPM is that the comparison itself is not influenced by the choice of original circuit, \( C \). Consequently, if the obfuscator \( \mathcal{O} \) does not encrypt (i.e., semantically transform) a circuit, the indistinguishability comparison can still be performed. This fact allows us to segregate the white-box component of \( \mathcal{O} \) from its black-box component as we explore randomization methods for white-box obfuscation of circuits.

To perform white-box obfuscation, we consider the process of subcircuit selection and replacement. Two reasons drive us to this choice. First, to randomly select a white-box replacement of \( C \) would require enumeration of all circuits in \( \delta_C \). As circuit size increases, \( \delta_C \) becomes prohibitively large, and the obfuscator suffers greater-than-polynomial slowdown. Second, the separate steps of subcircuit selection and subcircuit replacement offer opportunities to inject randomness into the white-box obfuscation process.

Section 3.4.3 describes selection and replacement in greater detail, but we introduce here the basic of the concept (reference Figure 3.4). Given a circuit \( C \) which is to be white-box obfuscated, select a subcircuit, \( C_{sub} \). Retrieve a randomly chosen circuit \( C_{rep} \) from a library of circuits which contains a set of all circuits semantically equivalent to \( C_{sub} \) (the assumption that such a library exists will be discussed in Section 3.2.3). Finally, remove \( C_{sub} \) from \( C \) and insert \( C_{rep} \) in its place. As long as \( C_{sub} \) and \( C_{rep} \) are semantically equivalent (and the order of inputs and outputs is preserved), then semantic equivalence exists for \( C \), all \( C'_{i} \), and \( C'_{n} \).

3.2.3 Circuit library exists. A library of replacement circuits must exist in order for the process of iterative randomization to be possible. However, in Section 3.2.2 we said that enumerating all possible replacements for \( C \) would violate the polynomial slowdown condition of RPM. We overcome this apparent contradiction
Figure 3.4: Two representations of iterative white-box randomization.
(a) White-box obfuscation of circuit $C$ by iteratively replacing randomly selected subcircuits ($C_{\text{sub}}$) with a semantically equivalent subcircuit ($C_{\text{rep}}$) chosen randomly from a circuit library. $C$ is the unobfuscated circuit, $C'_i$ is $C$ after the $i^{th}$ iteration of replacement, and $C'_n$ is $C$ after an $n$-iteration obfuscation is complete.
(b) Depicts the sequential iterations of subcircuit selection and replacement.
by developing a library whose contents are limited to only small circuits, typically on the order of 5 or fewer gates. In this way, all semantically equivalent circuits in a particular family (i.e., all \( C \in \delta_C \)) can be enumerated. Therefore, in the iterative replacement process, a given \( C_{\text{rep}} \) can truly be selected from among all size-bounded circuits semantically equivalent to \( C_{\text{sub}} \).

### 3.3 Obfuscation toolkit

As this research is empirically based, a software tool was developed to perform the white-box circuit obfuscation portion of the RPM. Although the RPM calls for both black-box (program encryption) and white-box (randomization) techniques, they are performed independently from one another. This allows us to develop software which only performs the white-box function. The tool has two major components, CORGI and CXL.

#### 3.3.1 CORGI: the circuit randomizer

CORGI, which stands for Circuit Obfuscation via Randomization of Graphs Iteratively, was developed to empirically analyze the RPM. Its development was a major benefit of this research. The inner workings of the software are described in greater detail in Appendix A. Here, we briefly discuss the main features of CORGI.

##### 3.3.1.1 Development environment

CORGI is coded entirely in Java. Several factors influenced this choice. First, there is a strong emphasis on object-oriented design (OOD) at the Air Force Institute of Technology (AFIT), and Java is the de facto language of choice for the academic environment. Second, given the nature of the problem domain (i.e., circuits), OOD is a logical design choice. The third factor is based on our choice of application domain (i.e., to represent circuits as graphs), which allowed us to incorporate JGraphT into the development.

---

2 The circuit library used in this research is a product of concurrent research conducted by Capt Moses James. His research focuses on circuit randomization as a set selection problem.
Table 3.2: The most notable features and benefits JGraphT contributed to the development of CORGI.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit to CORGI development</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph package</td>
<td>Model CORGI circuits as graphs. In particular, JGraphT’s graph package included classes for all the types of graphs described in Section 3.2.1.2</td>
</tr>
<tr>
<td>Subgraph class</td>
<td>Manipulate subgraphs without modifying the base graph. This is a critical component of the subcircuit selection and replacement process.</td>
</tr>
<tr>
<td>Exporter classes</td>
<td>Export circuits to standard formats used by various graph software packages (e.g., yGraph, GraphVis, prefuse, etc.). Allows user to render circuits visually.</td>
</tr>
<tr>
<td>Algorithms package</td>
<td>Contains classes for standard algorithms used in graph theory. In particular, the CycleDetector class is a critical part of CORGI because it enforces the acyclic nature of DAGs.</td>
</tr>
</tbody>
</table>

JGraphT is an open source Java graph library [14]. Its free availability as an open source project shortened the time to develop CORGI by at least several weeks—possibly much more. JGraphT provides the means to easily generate graphs and apply to them many of the common graph theory techniques. It is the crux of what makes CORGI work. JGraphT not only provides the ability to model the underlying graph of a circuit, it also has methods and services which make circuit manipulation and analysis possible. [Table 3.2] shows the key features and benefits of JGraphT.

Despite the graph basis for circuit manipulation—as implemented by way of the JGraphT library—CORGI completely elides from the user any references to graphs or graph behavior. Thus, CORGI is effectively a translation between the two domains.

3.3.1.2 Subcircuit selection and replacement. Subcircuit selection and replacement is the principle function CORGI performs. From the user perspective, it is a single action, but as already described, this function is iterative. We describe in more detail here the mechanics of how CORGI carries out one iteration of the process (ref. Figure 3.1).
CORGI does not actually select subcircuits. Instead, it selects a subset of the circuit’s gates based on a selection strategy chosen by the user. This subset of the circuit’s gates corresponds to a subset of vertices in the underlying graph, by which a vertex-induced subgraph (or simply subgraph) is derived. CORGI then copies the subgraph (leaving the base graph unchanged) and uses it to construct a separate subcircuit representative of the gates selected.

Next, CORGI uses the new subcircuit’s truth table, along with other user inputs, to request a replacement from the circuit library (CXL). CXL selects a random, semantically equivalent, subcircuit replacement (i.e., its truth table is the same). The original subcircuit is removed from the circuit, and the replacement subcircuit is inserted in its place.

3.3.2 CXL: the circuit library. CXL is a component of CORGI which contains a library of circuits. In a sense, CXL is really a library of sets of circuits. Each set is a circuit family $\delta_C$ where $C$ is characterized by a particular class $C_{X,Y,S,\Omega}$ (ref. Table 3.1).

Because of the various equivalence relationships in Boolean logic, $|\delta_C|$ rapidly increases exponentially with even small increases in $S$ and/or $|\Omega|$. For practical reasons, we choose $S \leq 3$, although we do allow $\Omega \subseteq \{\text{AND, NAND, OR, NOR, XOR, XNOR, NOT}\}$ (i.e., $|\Omega| \leq 7$).

From a user perspective, CXL is not a separate component from CORGI. Indeed, CXL is accessed by CORGI via an interface, which is called from within the iterative function of subcircuit selection and replacement. The user provides parameters which are used by the interface, but the call itself is not controlled by the user. Because of this, we consider CXL to be an integrated component of CORGI, and this perspective is implicit in any further references to CORGI unless otherwise stated.

See [10] for more detailed information on the behavior of CXL.

---

3 The initial implementation of CORGI limits selection to only one or two gates, primarily for performance reasons, but also due to limitations imposed by the circuit library.
3.4 Empirical Approach

This research is predicated on the notion that we need empirical data to be able to demonstrate whether practical obfuscation might be possible in light of theoretic impossibility results. Perhaps there exist imperfect obfuscators that protect circuits to a useful, measurable degree. Inherent in the preceding conjecture are two questions:

- What properties of circuits are indicators of useful, measurable circuit protection?
- What methods of obfuscation produce such properties in circuits?

Since our standard of useful is the RPM, we are really asking what properties of circuits are indistinguishable between an obfuscated circuit, $C'$, and a randomly selected (generated) circuit, $C_R$. If we know which properties relate to indistinguishability under the RPM, our intuition is we should be able to easily find algorithms which produce those properties in $C'$. On the other hand, if we know that a particular obfuscator will produce a $C'$ which meets the RPM definition of indistinguishability, we can deduce which properties are indicators of well-obfuscated circuits.

In reality, we do not know a priori the answer to either of the two questions above. Our approach, therefore, is to work the problem incrementally to see where the results converge. We briefly consider several candidate properties with which to measure circuit obfuscation under RPM. Then we propose several algorithms for performing subcircuit selection as part of the iterative randomization process. These algorithms are applied to a circuit, $C$, and then the resulting white-box obfuscated circuit, $C'$, is examined for their effect on obfuscation under RPM. Next, we define some key concepts used in the discussion which follows.

3.4.1 Key concepts. First, a circuit property, as we shall use the term, is a descriptor of a single circuit. This is an important distinction since the white-box circuit obfuscation process we employ is iterative (ref. Figure 3.4), creating many intermediate circuits $C_i'$ before finishing with $C_n'$ ($C_n'$ is the same as $C'$ in
These intermediate circuits provide us the means to measure how a given property changes throughout the iterative process, but each $C_i'$ will have its own set of properties independent of any other circuit.

Second, since combinational circuits are modeled as DAGs, we look initially to graph theory for properties of graphs which may be candidate measures of circuit obfuscation. This choice leads us to also use graph terminology to describe some of the properties. When this occurs, equivalent terminology—if it exists—is included parenthetically.

Third, our use of the term path is limited to only those paths which begin at a circuit input and end at a circuit output. The intention is to describe control flow through a circuit.

Fourth, DAGs are by their nature hierarchical, thus combinational Boolean circuits are, too. A circuit’s gate hierarchy is dictated by the predecessor or successor relationships of the various gates in the circuit. By our convention, if a gate precedes another gate in some path through the circuit, then the preceding gate is at a higher level. Equivalently, if a gate succeeds another gate along some path through the circuit, then the succeeding gate is at a lower level. It is possible that a particular gate could be assigned to any one of several levels, but our convention is to assign the gate to the lowest level that preserves the hierarchy of the circuit.

Figure 3.5 demonstrates the concept of gate hierarchy. Note that gate B is at level 2, not level 1, as is gate C. This is because the longest path from inputs of gate B to the output of gate D is length 2. Similarly, gate A could have been assigned to a new level 3, but the addition of the extra level breaks the convention that gates should be assigned to the lowest level that preserves the hierarchy of the circuit.

Finally, certain proposed circuit properties are frequency distributions, represented graphically as histograms. An example might be the number of unique paths that transit each gate. In Figure 3.5, for example, gate A has two unique paths:
Figure 3.5: A simple example of circuit hierarchy.
(a) A simple circuit \((X = 4, Y = 1, S = 4, \Omega = \{\text{NAND}\})\) without hierarchical levels.
(b) The same circuit with lowest hierarchy level assigned to each gate.

\(i_0\)-A-C-D and \(i_1\)-A-C-D. Similarly, gate B has two, gate C has four, and gate D has six. The associated histogram is shown in Figure 3.6.

3.4.2 Properties of obfuscated circuits. A property of a circuit may be a single value (e.g., average path length), or a distribution of values (see Figure 3.6). In case of the latter, the property will be identified as such. We propose several circuit properties as candidate measures of circuit obfuscation, without consideration of the efficacy of each property (see Table 3.3).

To be clear, the properties listed in Table 3.3 serve two purposes. First, they are objects of the proposed algorithms (Section 3.4.3 below). Second, they are collectively a leaping-off point for future research on which circuit properties are strong indicators of effective obfuscation.
Figure 3.6: A simple example of a histogram of a circuit property. The chart represents the frequency of occurrence of gates having a particular number of unique paths passing through them. In this example circuit, two gates have two unique paths (gates A and B), one gate has four unique paths (gate C), and one gate has six unique paths (gate D).

Table 3.3: A set of candidate circuit properties for measuring circuit obfuscation.

<table>
<thead>
<tr>
<th>Circuit-level</th>
<th>Gate-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of vertices at each hierarchical level</td>
<td>Number of paths through each gate [distribution]</td>
</tr>
<tr>
<td>[distribution]</td>
<td></td>
</tr>
<tr>
<td>Set of input/output pairs as determined by paths through the circuit</td>
<td>Number of unique input/output pairs represented by paths through each gate [distribution]</td>
</tr>
<tr>
<td>Number of vertex (gate) types (</td>
<td>Ω</td>
</tr>
<tr>
<td>Number of each vertex type (e.g., AND, OR, etc.) [distribution]</td>
<td>Number of predecessors of each gate (i.e., gate fan-in) [distribution]</td>
</tr>
</tbody>
</table>
Table 3.4: A set of candidate subcircuit selection algorithms used to iteratively white-box obfuscate a circuit. Algorithm names are derived from the file name of the Java class which implements the algorithm in CORGI.

<table>
<thead>
<tr>
<th>Selection Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RandomSingleGate</td>
<td>Selects a single gate at random</td>
</tr>
<tr>
<td>RandomTwoGates</td>
<td>Selects two gates at random</td>
</tr>
<tr>
<td>RandomLevelTwoGates</td>
<td>Selects a hierarchical level at random, and limits replacement to two gates selected at random from that level (±1 level)</td>
</tr>
<tr>
<td>FixedLevelTwoGates</td>
<td>Same as RandomLevelTwoGates except the hierarchical level is specified</td>
</tr>
<tr>
<td>LargestLevelTwoGates</td>
<td>Same as FixedLevelTwoGates except the hierarchical level is the one containing the most gates</td>
</tr>
<tr>
<td>OutputLevelTwoGates</td>
<td>Same as FixedLevelTwoGates except the hierarchical level is 0 (level 0 contains all the output gates)</td>
</tr>
</tbody>
</table>

3.4.3 **White-box obfuscation algorithms.** CORGI was designed to use multiple, interchangeable subcircuit selection algorithms. Recall that under the RPM, an obfuscated circuit $C'$, which is semantically equivalent to circuit $C$, is indistinguishable from a random circuit $C_R$. We would like to be able to select $C'$ from a completely enumerated $\delta_{C'}$, but for large $|C|$, the size of $\delta_{C'}$ is prohibitively large to enumerate all circuits in the set. This limitation forces us to choose another method of random “selection” of $C'$: iterative randomized subcircuit selection and replacement.

The process of obfuscating a large circuit by iteratively randomizing small subcircuits provides opportunities and introduces challenges as compared to direct selection from $\delta_{C'}$. An advantage of the process is that a subcircuit selection algorithm can be chosen such that it optimizes a particular obfuscation metric. A disadvantage, due to the fact that the process is a metaheuristic, may be that a particular sequence of iterations will converge on a final $C'$ with a suboptimal value for the target property.
Table 3.4 lists a candidate set of randomization algorithms developed for this research with a brief description of each. In Chapter IV, we analyze these algorithms and how they were derived.
IV. Results

CORGI is an architecture for obfuscating combinational Boolean circuits via iterative subcircuit selection and replacement. Six strategies for subcircuit selection are implemented in CORGI as modular algorithms. When executed, these algorithms transform a circuit $C$ into a randomized (i.e., white-box obfuscated) but semantically equal circuit $C'$. The nature of the transformation is different for each algorithm. Also, the design of certain CORGI components degrades CORGI performance (runtime) when some selection algorithms are employed.

4.1 Overview

To perform white-box obfuscation under the RPM on circuit $C$, we would ideally like to enumerate all circuits in $\delta_C$, then select one at random as the semantically equivalent replacement circuit $C'$. Such enumeration is infeasible for large circuits, which means a replacement circuit cannot be directly selected at random. Instead, it must be built, but still yield a random $C' \in \delta_C$. The process of iterative subcircuit selection and replacement described in Section 3.2.2 provides two ways for introducing randomness into the process.

1. Random selection: Select a subcircuit $C_{\text{sub}} \subset C$ at random.

2. Random replacement: Select a replacement circuit $C_{\text{rep}} \in \delta_{C_{\text{rep}}}$ at random.

There may also be some intermediate circuit $C'_i$ for which non-random selection and replacement are preferred. Here, also, there are two such smart choices.

1. Smart selection: Only select subcircuits which have a particular property. If the subset of allowable selections contains more than one subcircuit, then one may be selected at random or based on another property.

2. Smart replacement: Similar to smart selection, only select replacement circuits from the library which have a particular property. If the subset of allowable selections contains more than one subcircuit, then one may be selected at random or based on another property.
4.2 Limitations

Our research exposed certain limitations on the development of subcircuit selection and replacement algorithms. *Smart* strategies often impinged upon temporal or spatial efficiency, and the problem domain (i.e., combinational Boolean circuits) reduced the randomness of *random* selection strategies as we seek to avoid creating sequential circuits.

4.2.1 Smart strategies. There are multiple ways to make smart subcircuit selections. Some examples include choosing only subcircuits with a particular input size \(X_{\text{sub}}\), output size \(Y_{\text{sub}}\), circuit size \(S_{\text{sub}}\), basis \(\Omega_{\text{sub}}\), and/or truth table. Selection can also be made based on particular subsets of the circuit’s gates. For example, select only subcircuits which have gates in a particular hierarchical level in the circuit. Other smart selection strategies require searching the underlying graph for isomorphic subgraphs, which is an NP-complete problem [5]. These can all pose intractability problems for our iterative randomization process when we have large circuit sizes, which limits the efficiency of the search algorithm.

Consider a smart selection strategy which is based on subgraph isomorphism. Since the search is NP-complete, and the search space can be quite large (circuits with thousands, perhaps millions of gates), the strategy becomes too computationally intensive to be efficient, as required by the RPM.

Two of the six algorithms (*RandomSingleGate* and *RandomTwoGates*) use a purely random selection strategy which are discussed in Sections 4.3.2 and 4.3.3. The other four algorithms (*RandomLevelTwoGates*, *FixedLevelTwoGates*, *LargestLevelTwoGates*, and *OutputLevelTwoGates*) use a blend of smart and random selection, as is described in Sections 4.3.4–4.3.7. None of the latter four algorithms use NP-complete selection strategies.

As for smart replacement, CXL currently has no means to employ such a strategy. The problem is more a limit on space than on time. Specifically, if all replacement circuits are stored with sufficient metadata, then finding a particular replacement is
basically a simple lookup in a database. However, as the size bound of candidate replacement circuits increases, the size of the library increases exponentially, thus limiting the set from which replacements can be selected.

4.2.2 Introduced cycles. The choice of combinational Boolean circuits places a particular limitation on which subcircuits may be selected for replacement, as stated in Axiom 1.

**Axiom 1.** In order to maintain the combinational structure of circuit $C$, the set of gates $G(C_{sub})$ in a selected subcircuit $C_{sub}$ must not contain any pair of gates $(G_i, G_j)$ such that (WLOG):

- (a) $G_i$ precedes $G_j$ along some directed path in $C$, and
- (b) the longest directed $G_i$-$G_j$ path in $C$ is $\geq 2$.

The results of improperly selecting $C_{sub}$ is shown in Figure 4.1. Figure 4.1(a) shows a 4-input, 1-output, 4-gate circuit. In Figure 4.1(b), a 3-input, 2-output, 2-gate subcircuit $C_{sub}$ is selected for replacement. However, $C_{sub}$ contains a pair of gates, B and D, which violate Axiom 1. Figure 4.1(c) shows that a cycle is created if $C_{sub}$ is replaced with any replacement circuit $C_{rep}$. The problem occurs because gate C receives an output from $C_{sub}$ but also provides an input to $C_{sub}$, thus creating a cycle. If $C_{sub}$ is improperly selected, there exists no $C_{rep}$ such that a cycle is not created.

As a result of this limitation, the manner of subcircuit selection in CORGI requires a sequential selection of gates for those algorithms which select multi-gate subcircuits. There are differences in how this is performed for each algorithm, which are discussed below. The important point here is that, once the set of gates is selected, the subcircuit is defined (induced) by the set of selected gates, as well as all connections ("wires") leading into or out of those gates. It is not necessary that selected gates be connected directly to each other in $C$. 

34
4.3 Analysis of subcircuit selection algorithms

For this research, six subcircuit selection algorithms were developed. All algorithms adhere to a standard selection interface in CORGI, which does not actually select a subcircuit $C_{\text{sub}}$ directly from a circuit $C'$. Instead, the interface requires each algorithm to return a set of gates $G$. CORGI then uses $G$, together with JGraphT’s `DirectedSubgraph` class to create the subcircuit $C_{\text{sub}}$ induced by the selected gates in set $G$. Thus, each algorithm returns a set of gates, not a subcircuit. The sections that follow describe the manner of selection and the “behavior” each algorithm exhibits.

The development of these algorithms was itself an iterative process. As each new algorithm was developed and tested, the results would suggest alternate strategies for selection. Therefore, the algorithms are presented below in roughly the same order in which they were developed.
4.3.1 Common functions. The overall process of randomization is also presented in Appendix A (Algorithm 7). For the sake of brevity here, we defer to Appendix A, Section A.2 for the details of two functions used by the six selection algorithms discussed below: SelectRandomGate (Algorithm 8) and RejectGates (Algorithm 9). It is sufficient to know that SelectRandomGate randomly selects a gate from a set of gates, and RejectGates populates a set of gates which should not be part of the input to SelectRandomGate.

A third function, EstablishGateHierarchy, is used only by the so-called level algorithms (those for which selection is based on a circuit’s hierarchical level—all have “Level” in their name). The details of EstablishGateHierarchy are presented in Appendix A (Algorithm 10), but its basic functionality is to assign each gate to the lowest allowable level in the circuit’s hierarchy. The details of why this function is required will be presented in Section 4.3.4, where we introduce the first of the level algorithms, RandomLevelTwoGates.

4.3.2 RandomSingleGate. RandomSingleGate was the first selection algorithm developed for CORGI. As the name implies, all subcircuit selections $C_{sub}$ are of the class $C_{X_{sub}^{1-1-1-\Omega_{sub}}}$ where

$$
\Omega_{sub} \subset \{\text{AND, NAND, OR, NOR, XOR, XNOR}\}
$$

$$X_{sub} \geq 2$$

$$Y_{sub} = S_{sub} = |\Omega_{sub}| = 1$$

Since all $C_{sub}$ contain only one gate, any gate can be selected from $C_i'$ and replaced without creating cycles in $C_{i+1}'$ (as long as $C_i'$ is combinational). The selection procedure is simple, as shown in Algorithm 1.

RandomSingleGate was developed initially as a simple algorithm by which CORGI functionality could be tested. The function of removing a subcircuit from a circuit, then replacing it with a different subcircuit is a non-trivial activity. Se-
Algorithm 1 RandomSingleGate($C'_i$)

1: $G_{sub} \leftarrow \emptyset$ \{set of gates (1 in this case) to induce $C_{sub}$\}
2: $G(C'_i) \leftarrow$ set of all gates in $C'_i$
3: $g_k \leftarrow$ call SelectRandomGate($G(C'_i)$)
4: $G_{sub} \leftarrow G_{sub} \cup \{g_k\}$
5: return $G_{sub}$

Selecting a single-gate subcircuit for replacement, while simple to do, provides multiple dimensions by which to test the process of iterative randomization. When the subcircuit is replaced, gate properties such as type (NAND, NOR, etc.), fan-in (number of adjacent predecessors), fanout (number of adjacent successors), and whether the selected gate is a circuit output, must all be accounted for.

RandomSingleGate is a purely random (as opposed to smart) selection process. No knowledge of the target circuit is needed other than the set of gates in the circuit. The iterative process cannot be guided in any way.

There are three results produced by RandomSingleGate. First, no new external control flows are introduced in the circuit; second, the size $S$ of $C'_{i+1}$ is never smaller than $C'_i$; and third, the circuit becomes very “tall” ($\ell \propto n$, where $\ell = \text{number of hierarchical levels, and } n = \text{number of replacement iterations}$).

The first result is contingent on how we use the term control flow. If we have access to the structure of a circuit, then every unique path through a circuit is a control flow. If, however, we only have black-box access to a circuit, then no distinction can be made between unique paths which share a common source (input) and sink (output). We will refer to the former as internal control flow and the latter as external control flow. RandomSingleGate will never introduce new external control flows because all subcircuit inputs connect to a single subcircuit output. However, RandomSingleGate will always introduce new internal control flows. The only way RandomSingleGate does not produce new internal control flows is the trivial case where the selected single-gate subcircuit is replaced with itself. All other semantically equivalent replacements
have more than one gate, with connections between them; thus new internal control flows are always introduced.

The second result is a function of the replacement subcircuits $C_{\text{rep}}$ returned by CXL. In order for a replacement $C_{\text{rep}}$ of a single gate subcircuit $C_{\text{sub}}$ to change $C'_i$, $C_{\text{rep}}$ must have more than one gate. The reason for this is there is no non-trivial single-gate equivalence between any pair of gates $(g_i, g_j)$ in $\Omega = \{\text{AND, NAND, OR, NOR, XOR, XNOR}\}$.

The third result is a natural consequence of the first two. A subcircuit $C_{\text{sub}}$ comprised of a single Boolean logic gate has only one hierarchical level ($\ell = 1$). All nontrivial replacements $C_{\text{rep}}$ of $C_{\text{sub}}$ have at least two gates. If $C_{\text{rep}}$ has $n$ gates, then it can have $1 \leq \ell \leq n$ hierarchical levels. If $\ell \geq 2$, then $C'_{i+1}$ could “grow”—relative to $C'_i$—by as much as $\ell - 1$ levels (although it may not grow at all). The rate of growth over many iterations is a function of which gates are selected and the average number of levels in each $C_{\text{rep}}$ selected from CXL.

Reference Figure A.5(b) for a sample result of applying this algorithm to IS-CAS benchmark circuit C17. As we see from the data presented in Section A.3, RandomSingleGate produces the tallest $C'$ on average among all the circuits.

4.3.3 RandomTwoGates. RandomTwoGates is meant to be a two-gate version of RandomSingleGate. All subcircuit selections $C_{\text{sub}}$ are of the class $C_{X_{\text{sub}}}Y_{\text{sub}}2-\Omega_{\text{sub}}$, where

$$\Omega_{\text{sub}} \subset \{\text{AND, NAND, OR, NOR, XOR, XNOR}\}$$

$$X_{\text{sub}} \geq 2$$

$$Y_{\text{sub}} = 1 \text{ or } Y_{\text{sub}} = 2$$

$$S_{\text{sub}} = 2$$

$$|\Omega_{\text{sub}}| = 1 \text{ or } |\Omega_{\text{sub}}| = 2$$
The selection of $C_{\text{sub}}$ is accomplished by sequentially selecting the two gates. The first gate, $g_1$, is selected entirely randomly, in exactly the same fashion as the gate $g_k$ was selected by the RandomSingleGate algorithm. The second gate, $g_2$, must be selected more carefully, however, in order not to introduce cycles after replacement. Specifically, $g_2$ can only be selected from a specific subset of gates in $C_i'$ that remains after $g_1$ was selected (ref. Section 4.2.2). The procedure is shown in Algorithm 2.

Algorithm 2: RandomTwoGates($C_i'$)

1: $G_{\text{cand}} \leftarrow \emptyset$ \{set of candidate gates to select from randomly\}
2: $G_{\text{cand}} \leftarrow G_{\text{cand}} \cup G(C_i')$ \{set of all gates in $C_i'$\}
3: $g_1 \leftarrow \text{call SelectRandomGate}(G_{\text{cand}})$
4: $G_{\text{cand}} \leftarrow G_{\text{cand}} - \{g_1\}$ \{$g_1$ cannot also be $g_2$\}
5: $G_{\text{cand}} \leftarrow G_{\text{cand}} - \{\text{call RejectGates}(g_1, \text{true})\}$ \{remove predecessors of $g_1$\}
6: $G_{\text{cand}} \leftarrow G_{\text{cand}} - \{\text{call RejectGates}(g_1, \text{false})\}$ \{remove successors of $g_1$\}
7: $g_2 \leftarrow \text{call SelectRandomGate}(G_{\text{cand}})$
8: $G_{\text{sub}} \leftarrow \emptyset$ \{set of gates to induce $C_{\text{sub}}$\}
9: $G_{\text{sub}} \leftarrow G_{\text{sub}} \cup \{g_1, g_2\}$
10: return $G_{\text{sub}}$

There were two motivations for developing RandomTwoGates. We wanted to continue testing the capabilities of CORGI to determine if the selection/replacement process will work for $C_{\text{rep}}$ with more than one output. We also had the intuition that a replacement for a multi-input, multi-output subcircuit would introduce new external control flows.

RandomTwoGates is (almost) purely a random selection algorithm. The only caveat is that not every pair of gates $(g_1, g_2) \subset G(C_i')$ are “legal” selections since some pairs introduce cycles when replaced. Despite the fact that the candidates for selecting $g_2$ is a subset of $G(C_i') - \{g_1\}$, RandomTwoGates is in no way a smart selection algorithm.

There were three results from analyzing the behavior of RandomTwoGates. First, we confirmed our intuition that new external control flows can indeed be introduced. Second, similar to RandomSingleGate, the circuit becomes very tall, with few gates in any single hierarchical level. Third, RandomTwoGates runs slower than
RandomSingleGate because CXL must select from a larger store of semantically equivalent replacements as the number of inputs increases. We will discuss each result separately.

By far the most profound discovery was that new external (and internal) control flows can be introduced (but it does not always occur). The reason it can is because the subcircuit selected can be (and often is) comprised of two gates, $g_1$ and $g_2$, which are not adjacent to each other (i.e., $g_1$ is not a predecessor of $g_2$). If $g_1$ and $g_2$ are adjacent, then the resulting subcircuit $C_{sub}$ will have only one output, and RandomTwoGates will behave like RandomSingleGate for that single iteration.

The probability $P$ that RandomTwoGates creates a new control flow during any given replacement iteration $i$ is described by

$$P(i) \propto \left( 1 - \frac{n_e}{X \times Y} \right) \times p_c \times p_a$$

where $i$, $n_e$, $X$, $Y$, $p_c$, and $p_a$ are described below:

- $i$: A particular iteration of the algorithm
- $n_e$: Number of external control flows in $C_i'$ before selection
- $X$, $Y$: Number of inputs and outputs, respectively, of $C_i'$
- $p_c$: Probability that CXL returns a replacement subcircuit $C_{rep}$ with more control flows than $C_{sub}$ (ref. Figure 4.4 for an example)
- $p_a$: Probability that RandomTwoGates will choose two gates adjacent to one another (i.e., the output of one gate feeds an input of the other—ref. Section 4.3.2)

The foregoing can best be demonstrated with an actual circuit. Figure 3.2 depicts ISCAS benchmark C17, which was the target circuit for an experiment to demonstrate how selection algorithm RandomTwoGates can introduce new external control flows. A series of 20-iteration trials were performed until the final circuit $C'$ had more external control flows than the original $C$ (i.e., ISCAS benchmark C17). After only seven trials, a $C'$ was found with a path from input 1 to output 23, which was not present in $C$. CORGI has the capability to output the results of each iteration
of randomization, and by looking back through the data, we found that the seventh iteration produced the desired effect. Figure 4.2 shows the transition from $C'_6$ to $C'_7$ (iteration #7 in this example).

The second result for RandomTwoGates—the fact that it also makes circuits grow very tall—was somewhat unexpected. In retrospect, it probably should not have been since the same relationship between the hierarchical levels of $C_{sub}$ and $C_{rep}$ described in Section 4.3.2 exists for RandomTwoGates. As circuit size increases, the probability that $C_{sub}$ will have two hierarchical levels ($\ell = 2$) decreases since the number of adjacent gate pairs in $C'_i$ is exponentially smaller than the number of all gate pairs in $C'_i$. The rate at which a circuit obfuscated with RandomTwoGates grows taller is, on average, slightly slower than for RandomSingleGate since there is a non-zero probability that a one-output $C_{sub}$ is selected during a given iteration of subcircuit selection and replacement.

The third result has to do with a non-intuitive property of circuit families. The size of a given family $\delta$ is a function of several factors, including input quantity, output quantity, basis, and gate quantity. But it is also a function of the signature (truth table) of elements of $\delta$. Some families have circuit signatures such that there are relatively few (sometimes zero) elements. Others families may have thousands of elements. When a subcircuit $C_{sub}$ is selected such that $|\delta|$ is large, the selection of a replacement $C_{rep}$ takes longer. RandomTwoGates selects $C_{sub}$ such that $\delta_{C_{sub}}$ (from which CXL must choose a $C_{rep}$) is, on average, larger than it is when RandomSingleGate is the selection algorithm. See [10] for more details on the relationship of a circuit’s signature (truth table) to the size of its circuit family.

Reference Figure A.6(b) for a sample result of applying this algorithm to IS-CAS benchmark circuit C17. Again, from the data presented in Section A.3, RandomTwoGates produces $C'$ which are, on average, about half the height of circuits produced by RandomSingleGate.
Figure 4.2: Subcircuit selection and replacement using RandomTwoGates on ISCAS C17, which creates a new external control flow in the circuit (input 1 [In1] to output 23 [Out23]).

(a) Gates 31 and 32 ($C_{\text{sub}}$) will be removed from $C_6'$. Note there is no control flow from In1 to Out23.

(b) New circuit $C_7'$ is created after $C_{\text{sub}}$ in circuit $C_6'$ is replaced with semantically equivalent $C_{\text{rep}}$ (gates 41, 42, and 43). A new control flow now exists from In1 to Out23 (path: In1→35→41→42→29→40→Out23).
Figure 4.3: Subcircuit selection and replacement using RandomTwoGates on ISCAS C17, which replaces a two gates, each added during different iterations.

(a) Gates 39 and 43 ($C_{sub}$) will be removed from $C_7'$. Note that gate 39 was not in the original circuit.

(b) New circuit $C_8'$ is created after $C_{sub}$ in circuit $C_7'$ is replaced with semantically equivalent $C_{rep}$ (gates 44, 45, 46, and 47). Because of the structure of the selected replacement circuit, gates 38 and 35 are each elevated to the next higher layer in the circuit hierarchy.
Figure 4.4: An example of how a replacement subcircuit $C_{rep}$ can introduce a new control flow where none existed in the selected subcircuit $C_{sub}$ (reference Figure 4.2).

(a) No control flow exists between gate 35 and gate 29 in $C_{sub}$.
(b) Subcircuit $C_{rep}$ has a control flow from gate 35 to gate 29.
(c) The truth table of both circuits.
4.3.4 RandomLevelTwoGates. The RandomLevelTwoGates selection algorithm functions the same as RandomTwoGates except that \( G_{\text{cand}} \) only contains gates which are in at most three contiguous levels of the circuit hierarchy. All subcircuit selections \( C_{\text{sub}} \) are of the class \( C_{X_{\text{sub}} Y_{\text{sub}} 2\Omega_{\text{sub}}} \) where

\[
\Omega_{\text{sub}} \subset \{\text{AND, NAND, OR, NOR, XOR, XNOR}\}
\]

\( X_{\text{sub}} \geq 2 \)

\( Y_{\text{sub}} = 1 \) or \( Y_{\text{sub}} = 2 \)

\( S_{\text{sub}} = 2 \)

\( |\Omega_{\text{sub}}| = 1 \) or \( |\Omega_{\text{sub}}| = 2 \)

\( \ell_{g_2} = \ell_{g_1} \pm 1 \)

The similarity between RandomLevelTwoGates and RandomTwoGates is in how it selects the first gate, \( g_1 \). In both cases, \( g_1 \) is selected entirely at random. Since gate \( g_1 \) occupies some hierarchical level \( \ell_{g_1} \), then level \( \ell_{g_1} \) is a de facto random selection. The difference between these two algorithms is in how gate \( g_2 \) is selected. With RandomLevelTwoGates, gate \( g_2 \) must be selected from within levels \( \ell_{g_1} \), \( \ell_{g_1} + 1 \), or \( \ell_{g_1} - 1 \). As we saw with RandomTwoGates, its gate \( g_2 \) selection can be any gate that does not introduce a cycle. Note that RandomLevelTwoGates selects two gates which are within one level of each other. Therefore, no call to RejectGates is required since it is impossible to introduce a cycle.

Having observed that RandomTwoGates increases the number of hierarchical levels (i.e., the height) at approximately half the rate of RandomSingleGate (see Section A.3 for discussion), we developed RandomLevelTwoGates to see if we could further reduce the rate of height increase relative to the number of iterations performed, yet retain as much randomness as possible otherwise. The conjecture was that, by limiting subcircuit selection to gates in a single “band” of at most three hierarchical levels,
Algorithm 3 RandomLevelTwoGates($C_i'$)

1: call EstablishGateHierarchy() {Assigns each gate to a hierarchical level}
2: $G_{cand} \leftarrow \emptyset$
3: $g_1 \leftarrow$ call SelectRandomGate($G(C_i')$) {A random gate from any level}
4: $\ell_{g_1} \leftarrow$ hierarchy level of gate $g_1$
5: $G_{cand} \leftarrow G_{cand} \cup G(\ell_{g_1})$ {all gates in level $\ell_{g_1}$}
6: $G_{cand} \leftarrow G_{cand} - \{g_1\}$
7: if $\ell_{g_1} > 0$ then
8: $G_{cand} \leftarrow G_{cand} \cup G(\ell_{g_1} - 1)$ {all gates one level below $g_1$}
9: end if
10: if $\ell_{g_1} < \ell_{MAX}$ then
11: $G_{cand} \leftarrow G_{cand} \cup G(\ell_{g_1} + 1)$ {all gates one level above $g_1$}
12: end if
13: $g_2 \leftarrow$ call SelectRandomGate($G_{cand}$)
14: $G_{sub} \leftarrow \emptyset$
15: $G_{sub} \leftarrow G_{sub} \cup \{g_1, g_2\}$
16: return $G_{sub}$

The propensity of a replacement circuit $C_{rep}$ to increase the circuit’s height should be further mitigated.

The fact that we specifically disregard particular levels when choosing $g_2$ makes RandomLevelTwoGates a smart selection algorithm. Gate $g_1$ is still selected randomly, but since the subset of gates from which $g_2$ is chosen is dependent on $g_1$, we expect to be able to better control how RandomLevelTwoGates modifies a circuit. The reason that $g_2$ is not restricted only to $\ell_{g_1}$ is because of the nature of subcircuits $C_{rep}$ returned by CXL. As previously discussed in Section 4.3.2 (page 38), $C_{rep}$ can—and often does—have more than one hierarchical level. If it occurs that $C_{sub}$ has the same height as $C_{rep}$, then the overall circuit $C_i'$ will not grow in height during iteration $i$.

There were four results from analyzing the behavior of RandomLevelTwoGates. First, we confirmed our hypothesis that RandomLevelTwoGates produces shorter circuits than either RandomSingleGate or RandomTwoGates. Second, we demonstrated that a smart selection strategy can be employed to guide the behavior of a white-box obfuscator to a particular end. In this case, we took our observations of how single
iterations of random selection strategies impacted circuit growth to develop a smart algorithm.

The third result is the nature of the internal circuit structure. Unlike the two random selection algorithms, \texttt{RandomLevelTwoGates} has connections (edges) which span fewer hierarchical levels. This can be observed by comparing Figures A.5(b) and A.6(b). In the former image, many connections span more than half the length of the circuit, whereas in the latter image, connections spanning more than eight levels appear much less frequently. The implication of this finding is that level-based algorithms could be useful if connection length is a circuit property that correlates to the degree of obfuscation.

The fourth result has to do with algorithm efficiency. The function \texttt{EstablishGateHierarchy} is a component of this (indeed, all four) level-based algorithm. It must be invoked at the beginning of every iteration, as shown in algorithms 3, 4, 5, and 6 (line 1 in each). In its current implementation, \texttt{EstablishGateHierarchy} is inefficient\footnote{The details of why this is the case are discussed in Appendix A}. For \( C' \) with small size \(|S|\), this is not a problem; but as the number of iterations increase, the circuit size also increases, and \texttt{EstablishGateHierarchy} slows down the selection algorithm. Future versions of CORGI must take this performance factor into account in order that level-based selection algorithms are efficient for large circuits.

\subsection*{4.3.5 \texttt{FixedLevelTwoGates}}

The \texttt{FixedLevelTwoGates} selection algorithm functions the same as \texttt{RandomLevelTwoGates} except for two differences. Whereas in \texttt{RandomLevelTwoGates} the target level is based on the selection of gate \( g_1 \), the opposite is true here. \texttt{FixedLevelTwoGates} must first have a level \( \ell_F \) to target (user input), and from that level, it selects gate \( g_1 \) (the numerical value of \( \ell_F \) remains constant for all iterations). In addition, \texttt{FixedLevelTwoGates} selects gate \( g_2 \) only from levels \( \ell_F \) or \( \ell_F + 1 \) (not from \( \ell_F - 1 \)). All subcircuit selections \( C_{\text{sub}} \) are of the class \( C_{X_{\text{sub}}, Y_{\text{sub}}, 2-\Omega_{\text{sub}}} \).
where

\[ \Omega_{sub} \subset \{ \text{AND, NAND, OR, NOR, XOR, XNOR} \} \]

\[ X_{sub} \geq 2 \]

\[ Y_{sub} = 1 \text{ or } Y_{sub} = 2 \]

\[ S_{sub} = 2 \]

\[ |\Omega_{sub}| = 1 \text{ or } |\Omega_{sub}| = 2 \]

\[ \ell_{g2} = \ell_{g1} \text{ or } \ell_{g2} = \ell_{g1} + 1 \]

\[ \text{Algorithm 4 FixedLevelTwoGates}(C'_i) \]

1: call EstablishGateHierarchy()
2: \( G_{cand} \leftarrow \emptyset \)
3: \( \ell_F \leftarrow \text{fixed level where } 0 \leq F \leq \ell_{MAX} \{\text{user inputs } F\} \)
4: \( G_{cand} \leftarrow G_{cand} \cup G(\ell_F) \{\text{all gates in level } \ell_F\} \)
5: \( g_1 \leftarrow \text{call SelectRandomGate}(G_{cand}) \)
6: \( G_{cand} \leftarrow G_{cand} - \{g_1\} \)
7: if \( \ell_F < \ell_{MAX} \) then
8: \( G_{cand} \leftarrow G_{cand} \cup G(\ell_{g1} + 1) \)
9: end if
10: \( g_2 \leftarrow \text{call SelectRandomGate}(G_{cand}) \)
11: \( G_{sub} \leftarrow \emptyset \)
12: \( G_{sub} \leftarrow G_{sub} \cup \{g_1, g_2\} \)
13: return \( G_{sub} \)

The first three algorithms developed successively improved control over circuit growth as measured by circuit height, yet they each created a wide range of height results. In other words, over many trials, the data shows a large standard deviation (\( \sigma \)) for circuit height (see Figure A.2). Our motivation for developing FixedLevelTwoGates next was to observe whether targeting a single level for subcircuit selection would cause the circuit to grow wider than it did with RandomLevelTwoGates.

There were two findings regarding FixedLevelTwoGates, one negative, and one positive. First, it produces circuits which are (on average) both taller and
narrower than those produced by \textit{RandomLevelTwoGates}. This is the opposite of what we expected, but the circuits did exhibit one similarity to those produced by \textit{RandomLevelTwoGates}; namely, there are relatively few connections that span more than 10\% of the circuit’s height.

Second, however, \textit{FixedLevelTwoGates} achieved more predictable behavior relative to the number of iterations performed (i.e., smaller standard deviation, $\sigma$). We attribute that fact to limiting the selection of gate $g_2$ to only two, rather than three, contiguous levels in $C_i'$. This substantially limits the set of gates from which gate $g_1$ may be selected (the previous three algorithms select gate $g_1$ at random from among all gates in $C_i'$). As a result, this \textit{smart} selection algorithm has much less randomness, which may be the basis of the tight coupling between circuit height and number of iterations.

4.3.6 \textbf{LargestLevelTwoGates}. With \textit{LargestLevelTwoGates}, we combine the \textit{variable} level selection of \textit{RandomLevelTwoGates} with the \textit{targeted} level selection of \textit{FixedLevelTwoGates}. This algorithm is procedurally the same as \textit{FixedLevelTwoGates} except that the selected largest (widest) level, $\ell_W$, is calculated for every iteration.

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{call} EstablishGateHierarchy() \Comment{EstablishGateHierarchy}
\State $G_{\text{cand}} \leftarrow \emptyset$ \Comment{Initialize candidate gate set}
\State $\ell_W \leftarrow$ widest (largest) level where $0 \leq \ell_W \leq \ell_{\text{MAX}}$ \{tiebreaker: smallest $\ell_W$\}
\State $G_{\text{cand}} \leftarrow G_{\text{cand}} \cup G(\ell_W)$ \{all gates in level $\ell_W$\}
\State $g_1 \leftarrow \text{call SelectRandomGate}(G_{\text{cand}})$ \Comment{Select Random Gate}
\State $G_{\text{cand}} \leftarrow G_{\text{cand}} - \{g_1\}$ \Comment{Remove selected gate}
\If{$\ell_W < \ell_{\text{MAX}}$} \Comment{Check Level Bound}
\State $G_{\text{cand}} \leftarrow G_{\text{cand}} \cup G(\ell_{g_1} + 1)$ \Comment{Add Gates at Next Level}
\EndIf
\State $g_2 \leftarrow \text{call SelectRandomGate}(G_{\text{cand}})$ \Comment{Select Random Gate}
\State $G_{\text{sub}} \leftarrow \emptyset$ \Comment{Initialize subset}
\State $G_{\text{sub}} \leftarrow G_{\text{sub}} \cup \{g_1, g_2\}$ \Comment{Add Gates to Subset}
\State \Return $G_{\text{sub}}$ \Comment{Return Subset}
\end{algorithmic}
\end{algorithm}
Our objective in developing \texttt{LargestLevelTwoGates} is an algorithm that is agile enough to “chase” the largest (widest) level as $C'_i$ grows. The nature of subcircuit replacement, combined with the rigidity of predecessor relationships in a combinatorial Boolean circuit, causes gates to migrate to higher levels in the circuit hierarchy. When a gate moves from one level to the next, the population of the level it originally occupied decrements by one. To combat this tendency, \texttt{LargestLevelTwoGates} always selects gates from the largest level. If multiple levels are largest, choose the lowest level among them.

Two results from \texttt{LargestLevelTwoGates} are clearly evident in Figure A.7(c). First, the algorithm provides more control over circuit growth than any of the previous selection algorithms. From the data in Section A.3, we see the average height of $C'$ produced by \texttt{LargestLevelTwoGates} is approximately 54% the average height of $C'$ produced by its nearest competitor, \texttt{RandomLevelTwoGates}. Circuits produced using \texttt{LargestLevelTwoGates} are also much wider than any of the other circuits. \texttt{RandomLevelTwoGates} is again the closest competition, but \texttt{LargestLevelTwoGates} produces $C'$ more than twice as wide.

A second result is the fact that \texttt{LargestLevelTwoGates} can introduce external control flows, just as we first saw with \texttt{RandomTwoGates}. The $C'$ circuit represented in Figure A.7(c) has external control flows In1–Out23 and In7–Out22, whereas the original circuit $C$ in Figure A.7(a) does not. Thus, \texttt{LargestLevelTwoGates} provides a high degree of control over circuit growth, yet retains the potential to introduce control flows.

4.3.7 \textit{OutputLevelTwoGates}. The last of the six algorithms is another variation on a theme. \texttt{OutputLevelTwoGates} is, in fact, only a special case of \texttt{FixedLevelTwoGates} where the target level contains the circuit outputs. In other words, using \texttt{FixedLevelTwoGates} with $\ell_F = 0$ is the same as using \texttt{OutputLevelTwoGates}. Algorithm 6 shows this special case.
Algorithm 6 OutputLevelTwoGates($C'_i$)

1: call EstablishGateHierarchy()
2: $G_{cand} \leftarrow \emptyset$
3: $\ell_0 \leftarrow$ level 0 which only contains circuit output gates
4: $G_{cand} \leftarrow G_{cand} \cup G(\ell_0)$ \{all gates in level $\ell_0$\}
5: $g_1 \leftarrow$ call SelectRandomGate($G_{cand}$)
6: $G_{cand} \leftarrow G_{cand} - \{g_1\}$
7: $G_{cand} \leftarrow G_{cand} \cup G(\ell_1)$ \{all gates in level $\ell_1$\}
8: $g_2 \leftarrow$ call SelectRandomGate($G_{cand}$)
9: $G_{sub} \leftarrow \emptyset$
10: $G_{sub} \leftarrow G_{sub} \cup \{g_1, g_2\}$
11: return $G_{sub}$

OutputLevelTwoGates was developed purely out of curiosity, and it proved to be a worthwhile endeavor. When $\ell_F > 0$ in FixedLevelTwoGates, there is the possibility that the width of $\ell_F$ can increase. Conversely, the width of a circuit’s output level ($\ell_0$) is fixed, so any replacement of a subcircuit that contains an output gate must not increase the number of circuit outputs\(^2\) The net effect on circuit growth is best described by way of analogy, followed by three example $C''$ circuits produced by OutputLevelTwoGates.

The behavior of OutputLevelTwoGates resembles the manufacturing process of extrusion which creates long objects of a fixed cross-sectional profile. In this case, the cross-sectional profile is circuit width. However, unlike the random algorithms, OutputLevelTwoGates produces circuits in which all levels are approximately the same width as output level $\ell_0$. Regardless of how many outputs the circuit has, or how many iterations are performed, the widest level will contain only a few more gates than the output level, $\ell_0$.

To see the extrusion effect, reference $C'$ in Figure A.5(c) which has height of 93 levels and widest layer only of 4 gates. But on average, all layers are not 4 gates.

\(^2\)Under the concept of black-box refinement, adding decoy outputs—and inputs—is desirable. This research does not address the concept, however, so we restrict ourselves to preserving circuit input and output quantities.

\(^3\)An exception to this is when, prior to iteration 1, the widest level of $C$ is substantially wider than $\ell_0$.  

51
gates wide; they are only 2 gates wide, which is equal to the number of outputs. For another demonstration, we apply \texttt{OutputLevelTwoGates} to a different ISCAS benchmark circuit, C880, which has 26 outputs. The results, for different numbers of iterations, are shown in Figures A.8–A.10. Again, the average width of all layers is approximately 26, which is the same as $\ell_0$.

From these results, we must revisit our observations for \texttt{FixedLevelTwoGates}. Basically, \texttt{FixedLevelTwoGates} behaves the same as \texttt{OutputLevelTwoGates}, but the extrusion occurs at some user-defined level. In essence, the target circuit $C$ will be “split” at the chosen level $\ell_F$ which has a particular number of gates, $n_F$. All levels 0 through $\ell_F - 1$ will remain unchanged and an extruded subcircuit will connect the top and bottom of $C$ in the final randomized circuit $C''$.

4.4 \textit{Runtime performance analysis}

We conclude with a brief discussion on the runtime performance of the six algorithms. This is not intended to be a rigorous examination of CORGI performance, but instead it will provide an understanding of what factors influence run times as well as compare the performance of each of the six selection algorithms relative to one another. Figures 4.5–4.16 contain runtime performance data for the six algorithms as applied to two different ISCAS BENCH circuits: C17 and c880. Each figure displays representative results from two trials for each combination (i.e., selection algorithm and circuit). In all cases, each trial is 1000 iterations. Table 4.1 provides a summary of the data.

The time required for each iteration of randomization is comprised of the time needed to perform the selection and the time required for CXL to produce a replacement subcircuit. For all algorithms, the latter is independent of both the structure of $C'_i$ and the time required for CORGI to select a subcircuit from $C'_i$; however, the runtime of CXL is dependent on whether it generates an equivalent subcircuit at runtime, or simply selects an equivalent subcircuit from a static store. For the data presented here, we used the runtime option. Even though that choice is more time-
intensive, the average per-iteration time will remain constant over many iterations. Therefore, by comparing the results of one selection algorithm to those of another (or the same selection algorithm applied to different circuits), we can deduce the relative performance characteristics of CORGI.

RandomSingleGate is the fastest of the six selection algorithms. Each subcircuit only has one gate, thus the subcircuit only has one output. As a result, CXL can more quickly return a replacement. The slower times for RandomSingleGate when applied to C880 vs. C17 is because C880 initially has 437 gates to only 6 gates for C17.

The remaining five selection algorithms are substantially slower than RandomSingleGate primarily because selected subcircuits contain two outputs. Therefore, the library of equivalent subcircuits in CXL is substantially larger. Since, for our experiments, CXL generates the equivalent subcircuits at runtime, the per-iteration times increase substantially as compared to RandomSingleGate.

RandomTwoGates is the slowest of the six selection algorithms. RandomTwoGates is the only selection algorithm that calls RejectGates (Algorithm 9), which is a recursive DFS.

For the four level-based selection algorithms, average run times over 1000 iterations are all less than times for RandomTwoGates. Whereas RandomTwoGates

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>C17 Trial 1</th>
<th>C17 Trial 2</th>
<th>C880 Trial 1</th>
<th>C880 Trial 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RandomSingleGate</td>
<td>38 ms</td>
<td>37 ms</td>
<td>44 ms</td>
<td>44 ms</td>
</tr>
<tr>
<td>RandomTwoGates</td>
<td>441 ms</td>
<td>447 ms</td>
<td>476 ms</td>
<td>504 ms</td>
</tr>
<tr>
<td>RandomLevelTwoGates</td>
<td>290 ms</td>
<td>282 ms</td>
<td>430 ms</td>
<td>436 ms</td>
</tr>
<tr>
<td>FixedLevelTwoGates</td>
<td>404 ms</td>
<td>393 ms</td>
<td>420 ms</td>
<td>438 ms</td>
</tr>
<tr>
<td>LargestLevelTwoGates</td>
<td>350 ms</td>
<td>359 ms</td>
<td>400 ms</td>
<td>373 ms</td>
</tr>
<tr>
<td>OutputLevelTwoGates</td>
<td>331 ms</td>
<td>359 ms</td>
<td>438 ms</td>
<td>445 ms</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of runtime data for the six selection algorithms. The data show the average per-iteration time (in milliseconds) after 1000 iterations for two trials on each of two circuits: C17 and C880. Times are rounded to the nearest millisecond.
calls RejctGates, the four level-based selection algorithms all call EstablishGate-Hierarchy. This, too, is a DFS, but employs pruning. Pruning is a graph theory technique for limiting the search space, and in part accounts for the relative speedup of these four algorithms as compared to RandomTwoGates. Another factor that contributes to the increased speed of these four algorithms is the frequency of selecting single output subcircuits. Specifically, these algorithms select gates in adjacent hierarchical layers, which means the second gate selected by the algorithms is more likely to be a predecessor or successor of the first gate selected. The result of such a selection is a one-output subcircuit, which CXL more quickly produces than it does two-output subcircuits.
Figure 4.5: Sample per-iteration runtime data from applying selection algorithm RandomSingleGate to circuit C17.
Figure 4.6: Sample per-iteration runtime data from applying selection algorithm RandomSingleGate to circuit C880.
Figure 4.7: Sample per-iteration runtime data from applying selection algorithm RandomTwoGates to circuit C17.
Figure 4.8: Sample per-iteration runtime data from applying selection algorithm RandomTwoGates to circuit C880.
Figure 4.9: Sample per-iteration runtime data from applying selection algorithm RandomLevelTwoGates to circuit C17.
Figure 4.10: Sample per-iteration runtime data from applying selection algorithm RandomLevelTwoGates to circuit C880.
Figure 4.11: Sample per-iteration runtime data from applying selection algorithm FixedLevelTwoGates to circuit C17.
Figure 4.12: Sample per-iteration runtime data from applying selection algorithm FixedLevelTwoGates to circuit C880.
Figure 4.13: Sample per-iteration runtime data from applying selection algorithm LargestLevelTwoGates to circuit C17.
Figure 4.14: Sample per-iteration runtime data from applying selection algorithm LargestLevelTwoGates to circuit C880.

(a) Trial 1

(b) Trial 2
Figure 4.15: Sample per-iteration runtime data from applying selection algorithm OutputLevelTwoGates to circuit C17.
Figure 4.16: Sample per-iteration runtime data from applying selection algorithm OutputLevelTwoGates to circuit C880.
V. Conclusions

The work described in the foregoing chapters comprises only the beginning of a much larger effort. Going forward, we expect a steep learning curve given the “obstacle” of the impossibility result presented in [1]. However, this research—combined with that which will follow—seeks to set intent protection (which alters structure and function) apart from the common understanding of obfuscation (which only alters structure). In this research, we focused only on the process of white-box obfuscation, a necessary but not sufficient component of program intent protection. We further narrowed our scope to white-box obfuscating combinational Boolean circuits. We developed an architecture for manipulating circuits, and developed an initial set of algorithms for white-box obfuscating circuits via subcircuit selection and replacement.

5.1 Contributions

Perhaps our biggest contribution to our area of study is CORGI, the tool upon which this and future research is based. As with any new software, its development was not without difficulty. However, without CORGI, the process of subcircuit selection and replacement would have been entirely manual which would have yielded little data: calculations by-hand would simply take too long. On the other hand, the time spent to develop a stable architecture clearly impacted the number and complexity of selection algorithms that were produced. We view this tradeoff as appropriate since it will allow future research to focus on the process of obfuscation rather than the tool that performs the task.

The six subcircuit selection algorithms we produced yielded some surprising results, and they gave us new insights into the heretofore untested process of subcircuit selection and replacement. The RandomTwoGates algorithm alone provided two valuable results. First, it demonstrates that the gates of a subcircuit need not be connected to be selected. Additionally, RandomTwoGates also demonstrates how a circuit library (CXL) can provide replacement subcircuits that introduce new control
flows in the circuit. These results mean that completely disparate portions of a circuit can be intertwined, both from a black-box (functional) and a white-box (structural) perspective.

All six of the algorithms revealed that circuit size always increases when only one or two gates are selected for replacement. For single-gate subcircuits, all replacements have at least two gates. For a two-gate subcircuit, if its function is not semantically equivalent to a basic gate (AND, NAND, OR, NOR, XOR, or XNOR), then all replacement circuits in the circuit library will be, on average, larger than two gates. Unless and until we devise algorithms that select three or more gates can we expect to reduce circuit size. The ability to either increase or decrease circuit size is how the process of subcircuit selection and replacement will be able to produce a truly random circuit from a particular circuit family.

Finally, the three “smart” algorithms, especially, LargestLevelTwoGates, show how circuit growth can be controlled and predicted, even when the selection algorithm produces ever-increasing circuit size.

5.2 Future work

As alluded to above, we see 3-gate selection algorithms as the most important next step in devising an intent protection framework. One approach is to extend RandomTwoGates to select a third gate at random. This may be the easiest to do, but our insight is that it will provide results which will guide the development of other algorithms. In particular, as another approach, it may be advantageous during some iteration of selection to chose only subcircuits for which there is a large population of replacements in the circuit library. Such a strategy will require the algorithm to find subcircuits with a particular truth table. In graph theory, this is known as subgraph isomorphism, and is an NP-complete problem. Depending on circuit size, it may nonetheless be a feasible approach.

This assumes the library has a cache of metadata on its stores of circuit libraries which can be quickly and easily searched.
There are at least two ways CORGI can be augmented which have nothing to do with the algorithms directly. Currently, CORGI maintains no historical log of what steps and in what order were performed to obfuscate a circuit. A future version of CORGI with this capability would benefit the notion that an original circuit can be recovered from an obfuscated version. In a sense, such a log file would be analogous to a data encryption key for the white-box portion of the obfuscator. It remains to be seen what advantages might accrue for the cost of this operation, but it’s a question worth exploring.

Finally, CORGI is a solid proof-of-concept tool, but to make it better suited to the research, two major augmentations need to occur. An obvious shortcoming is the need for a better user interface. Although not addressed in this text, the tool functionality was accessed for this research entirely through test cases since the textual user interface was too cumbersome for repeated experimentation. Ideally, a graphical user interface will be developed so that rapid selection of input parameters and selection algorithm(s) will further keep the focus on experimentation rather than coding. CORGI also needs a review of the efficiency of some of its processes (not the selection algorithms themselves). Under the hood, there are several methods which employ recursive search algorithms that are not very efficient. They become even less efficient as circuit size increases. By instituting some optimization techniques, and limiting calls to these methods only when necessary, CORGI will be more likely to achieve, at worst, polynomial slowdown for large circuits.
Appendix A. CORGI software

A.1 CORGI architecture

A.1.1 Functionality. CORGI is a Java application which employs a model-view-controller (MVC) architecture. In Figure A.1 (page 71), the model is the Circuit, which is composed of Gate objects. The controller is CircuitController. The view is the UserCommandParser, which provides the user a text-based user interface.

A.1.1.1 JGraphT. The Java graph library JGraphT, introduced in Section 3.3.1.1 is the “engine under the hood” of CORGI. Recall, the ‘G’ in CORGI stands for graphs, and JGraphT is what allows us to manipulate circuits as DAGs, yet elide that fact from the user. Every circuit has an underlying graph (DAG), so Circuit is really a façade for a JGraphT DirectedGraph.

All circuit modifying behavior is contained in Circuit; however, the mechanism of subcircuit selection and replacement is modularized as a separate class, ... (more to come)

A.2 Non-selection algorithms

For the sake of brevity in the main text, the discussion of the non-selection algorithms is presented here. The entire process of subcircuit selection and replacement is given in Algorithm 7. The procedures for removeSubCircuit, fetchReplacement, and insertReplacement are elided since they are purely “mechanical” in the sense that they do not impact the selection process. Once a subcircuit $C_{sub}$ is selected from circuit $C'_i$, then these three methods will, respectively, remove $C_{sub}$, get a replacement circuit $C_{rep}$ from CXL, then insert $C_{rep}$ in place of $C_{sub}$.

Algorithm 8 (SelectRandomGate) and Algorithm 9 (RejectGates) are helper methods used by the six selection algorithms discussed in Chapter IV. SelectRandomGate simply selects a single gate at random from among a set of gates. This capability is needed since subcircuit selection relies on a sequence of random gate selections.
Figure A.1: The UML class diagram which shows the CORGI architecture.
Algorithm 7 performReplacement(Selection($C'_i$))
1: $C'_i$ ← circuit $C$ after $i$ iterations of randomization
2: $G_{sub} ← \emptyset$ \{subset of gates in $C'_i$: $G_{sub} \subseteq G(C'_i)$\}
3: $G_{sub} ←$ call Selection($C'_i$) \{the interface for the selection algorithms\}
4: $C_{sub} ←$ call RemoveSubCircuit($G_{sub}$)
5: $C_{rep} ←$ call FetchReplacement($C_{sub}$) \{this is the CXL interface\}
6: $C'_{i+1} ←$ call InsertReplacement($C_{rep}$)
7: return $C'_{i+1}$ \{circuit $C'_i$ after replacing $C_{sub}$ with $C_{rep}$\}

Algorithm 8 SelectRandomGate($G$)
\textbf{Require:} \hspace{.2cm} $G$ is a non-empty set of gates
1: $k ←$ uniform random number such that $0 \leq k < |G|$
2: $g_k ←$ the $k^{th}$ gate in $G$
3: return $g_k$

RejectGates identifies the set of all gates which lie on all paths through a
particular gate and which are more than one hierarchical level removed from said
gate. RejectGates is the means by which performReplacement prevents cycles from
being introduced in $C'_{i+1}$ when replacing a subcircuit that contains more than one
gate.
Algorithm 9 RejectGates($g_k, P$)

Require: $P$ true for predecessors of $g_k$, false for successors of $g_k$

1: $G_{rej} \leftarrow \emptyset$ \{set of rejected gates\}
2: $G_{curr} \leftarrow \emptyset$ \{set of gates being considered for rejection\}
3: $G_{prev} \leftarrow \emptyset$ \{set of gates already considered for rejection\}
4: $G_{next} \leftarrow \emptyset$ \{set of gates to be considered for rejection\}
5: $G_{adj} \leftarrow \emptyset$ \{set of predecessors (successors) of a gate\}
6: $G_{curr} \leftarrow G_{curr} + g_k$
7: if $P = \text{true}$ then
8: $G_{adj} \leftarrow$ predecessors of $g_k$
9: else
10: $G_{adj} \leftarrow$ successors of $g_k$
11: end if
12: for all gates $g_a$ in $G_{adj}$ do
13: if difference between hierarchy levels of $g_a$ and $g_k$ is $> 1$ then
14: $G_{curr} \leftarrow G_{curr} \cup \{g_a\}$
15: end if
16: end for
17: $G_{adj} \leftarrow \emptyset$
18: while $G_{curr} \neq \emptyset$ do
19: $G_{next} \leftarrow \emptyset$
20: for all gates $G_c$ in $G_{curr}$ do
21: if $P == \text{true}$ then
22: $G_{adj} \leftarrow$ predecessors of $G_c$
23: else
24: $G_{adj} \leftarrow$ successors of $G_c$
25: end if
26: $G_{next} \leftarrow G_{next} \cup G_{adj}$
27: end for
28: $G_{prev} \leftarrow G_{prev} \cup G_{curr}$
29: $G_{curr} \leftarrow \emptyset$
30: $G_{curr} \leftarrow G_{curr} \cup G_{next}$
31: end while
32: return $G_{rej}$

EstablishGateHierarchy is a circuit function that sets the hierarchy attribute for all gates in the circuit. When there are multiple paths between a particular pair of gates, and when one path is shorter than the other (in terms of number of gates along the path), then one or more of the gates on the shorter path could legally occupy any one of several levels in the hierarchy. We choose to assign gates to the
lowest possible level that adheres to this convention: every gate in the circuit will always occupy a level that is lower (smaller) than the level of any of its predecessors.

Algorithm 10 EstablishGateHierarchy()
1: label all gates as \( \ell_0 \)
2: \( \ell_G \leftarrow 0 \) \{initialize global maximum level\}
3: \( \ell_L \leftarrow 0 \) \{initialize local (output) maximum level\}
4: for all circuit output gates \( g_{out} \) do
5: \( \ell_L \leftarrow \text{call SetGateHierarchies}(g_{out}, 0, 0) \)
6: \( \ell_G \leftarrow \text{MAX}(\ell_L, \ell_G) \)
7: end for

None of the so-called level-based selection algorithms would function properly without EstablishGateHierarchy. EstablishGateHierarchy, in turn, relies upon the recursive function SetGateHierarchies (described in Algorithm 11). The way it works is to perform a DFS beginning at each circuit output, explore that output’s predecessor tree (in the underlying DAG), and set the hierarchy attribute for all gates along the way. Some pruning is performed, but there will invariably be gates that are visited at least twice, which makes EstablishGateHierarchy inefficient. Since so much of CORGI relies on gates having a correct hierarchy attribute, future versions of CORGI will benefit greatly from optimizing EstablishGateHierarchy.

Algorithm 11 SetGateHierarchies\((g_i, \ell_L, \ell_G)\)
1: for all predecessor gates \( g_j \) of gate \( g_i \) do
2: if \( \ell(g_j) \leq \ell(g_i) \) then
3: \( \ell(g_j) \leftarrow \ell(g_j) + 1 \)
4: \( \ell_G \leftarrow \text{call SetGateHierarchies}(g_j, \ell_L + 1, \ell_G) \)
5: end if
6: return \( \text{MAX}(\ell_L, \ell_G) \)
7: end for

A.3 Selection algorithm behavior

Figures A.2, A.3, and A.4 give insight into the behavior of the six selection algorithms.
Figure A.2: Experimental results from performing ten trials of 200 iterations each using all six selection algorithms, with ISCAS circuit C17 as the target C. To provide a common mode of comparison, all three tables are sorted in decreasing order of $H_{avg}$.

(a) The number of hierarchical levels in $C'$ (maximum, average, minimum, and standard deviation).

(b) The number of gates in the widest hierarchical level of $C'$ (maximum, average, minimum, and standard deviation).

(c) Height-to-width ratio and rate at which number of hierarchy levels increase per iteration.
Figure A.3: Chart of data from Figure A.2(a).
Figure A.4: Chart of data from Figure A.2(b).
A.4 Selection algorithm results

A.4.1 C17 with all algorithms. Figures A.5, A.6, and A.7 display examples of the results achieved when each of the six algorithms are applied to a simple ISCAS benchmark circuit, C17. In each case, the algorithm ran for 200 iterations. The images are DAGs which represent the various circuits. While they are not strictly circuits, they demonstrate the behavior of each algorithm. All images are drawn to relative scale for ease of comparison.\footnote{When viewing this document electronically in PDF format, the circuit details can be seen by zooming in to at least 1600\% magnification.}

A.4.2 C880 with OutputLevelTwoGates. Figures A.8, A.9, and A.10 shows how circuit C880 changes over time when randomized using the OutputLevelTwoGates selection algorithm. Compare Figure A.8 to Figure A.5(c). Note that C880, which has 26 outputs, grows in height much more slowly than does C17, which has 2 outputs, when OutputLevelTwoGates is applied for 200 iterations.
Figure A.5: Comparison of original circuit (ISCAS C17) to sample results of R1G and OL2G algorithms (200 iterations; circuits represented as DAGs).
(a) $C =$ ISCAS benchmark circuit C17 (height = 3 levels, width = 3 gates).
(b) $C'$ after applying RandomSingleGate to $C$ (height = 189 levels, width = 7 gates).
(c) $C''$ after applying OutputLevelTwoGates to $C$ (height = 93 levels, width = 4 gates).
Figure A.6: Comparison of original circuit (ISCAS C17) to sample results of R2G and FL2G algorithms (200 iterations; circuits represented as DAGs).
(a) $C =$ ISCAS benchmark circuit C17 (height = 3 levels, width = 3 gates).
(b) $C'$ after applying RandomTwoGates to $C$ (height = 93 levels, width = 6 gates).
(c) $C''$ after applying FixedLevelTwoGates to $C$ (height = 67 levels, width = 6 gates).
Figure A.7: Comparison of original circuit (ISCAS C17) to sample results of RL2G and LL2G algorithms (200 iterations; circuits represented as DAGs).
(a) $C = $ ISCAS benchmark circuit C17 (height = 3 levels, width = 3 gates).
(b) $C'$ after applying RandomLevelTwoGates to $C$ (height = 61 levels, width = 8 gates).
(c) $C'$ after applying LargestLevelTwoGates to $C$ (height = 32 levels, width = 15 gates).

Figure A.8: $C'$ after applying 200 iterations of OutputLevelTwoGates to ISCAS benchmark circuit C880 (height = 42 levels, width = 38 gates).
Figure A.9: $C'$ after applying 400 iterations of OutputLevelTwoGates to ISCAS benchmark circuit C880 (height= 60 levels, width= 31 gates).
Figure A.10: $C''$ after applying an additional 800 iterations of \texttt{OutputLevelTwoGates} to the circuit $C''$ in Figure A.9 (height= 98 levels, width= 33 gates).
Bibliography


Vita

Major Kenneth E. Norman graduated from Fayette County High School in Fayetteville, Georgia. He entered undergraduate studies at the Georgia Institute of Technology in Atlanta, Georgia where he graduated with a Bachelor degree in Electrical Engineering in 1992. He was commissioned through Officer Training School in 1993. In 2002, he earned his first Masters degree in Engineering Management at the Florida Institute of Technology.

Major Norman was first assigned to HQ Standard Systems Group, Maxwell AFB, Alabama in July 1993 as officer in charge of software development. In October 1996, he was assigned to the National Air Intelligence Center, Wright-Patterson AFB, Ohio where he served as an intelligence analyst. His third assignment began in October 1999 when he was selected to stand up a new joint interoperability program office at the US Army’s Communications-Electronics Command, Fort Monmouth, New Jersey. Next, he became an assignment officer for the developmental engineer career field at HQ Air Force Personnel Center, Randolph AFB, Texas in July 2002. Maj Norman was next assigned to the National Reconnaissance Office in Chantilly, Virginia in August 2004 as a systems engineer. While there, he was selected for in-residence Intermediate Developmental Education, which precipitated his assignment to attend the Air Force Institute of Technology in August 2006. Upon graduation, he will remain at Wright-Patterson AFB for his assignment to Air Force Research Laboratory.

Permanent address: Air Force Institute of Technology
2950 Hobson Way
Wright-Patterson AFB, OH 45433-7765
The index is conceptual and does not designate every occurrence of a keyword. Page numbers in bold represent concept definition or introduction.

acyclic graph, 17
black-box obfuscation, 8
circuit class, 13
CORGi, 20
CXL, 20
digraph, see directed graph
directed acyclic graph, 17
directed graph, 15

Index

graph, 15
multi-graph, 17
Random Program Model, 8
RPM, see Random Program Model

subcircuit replacement, 18
subcircuit selection, 18

VBB, see virtual black box
virtual black box, 4

white-box obfuscation, 8
Software protection remains an active research area with the goal of preventing adversarial software exploitation such as reverse engineering, tampering, and piracy. Heuristic obfuscation techniques lack strong theoretical underpinnings while current theoretical research highlights the impossibility of creating general, efficient, and information theoretically secure obfuscators.

In this research, we consider a bridge between these two worlds by examining obfuscators based on the Random Program Model (RPM). Such a model envisions the use of program encryption techniques which change the black-box (semantic) and white-box (structural) representations of underlying programs. In this thesis we explore the possibilities for white-box transformation. Under an RPM formulation, if an adversary cannot distinguish an original program from either its obfuscated version (whose black-box behavior has been strategically altered) or a randomly generated program of comparable size, then the white-box intent of the original program has been sufficiently protected. One proposed method of creating such random indistinguishability is by choosing (at random) a program from a size-bounded set of all semantically equivalent possibilities. Since full enumeration of reasonably sized programs is not possible, in this work we focus on obfuscators which introduce random white-box structural variation based on iterative selection and replacement. We design and develop an obfuscation framework for programmatic logic expressed as combinatorial Boolean circuits and compare six unique approaches for sub-circuit selection. We analyze the relative behavior of random and guided-random sub-circuit selection algorithms while showing their utility in producing random white-box structural variation.