**SANDIA REPORT**
SAND2018-11010
Printed September 2018

Sandia National Laboratories

# Diversity for Microelectronics Lifecycle Security

Jason R. Hamlet, Jackson R. Mayo, Mitchell T. Martin, David Torres, and Jonathan W. Cruz

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico
87185 and Livermore,
California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.



SAND2018-11010
Printed September 2018

# Diversity for Microelectronics Lifecycle Security

Jason R. Hamlet, Mitchell T. Martin, and David Torres
Systems Security Research
Sandia National Laboratories
P. O. Box 5800
Albuquerque, New Mexico 87185-MS0671

Jackson R. Mayo
Scalable Modeling and Analysis Systems
Sandia National Laboratories
P. O. Box 969
MS9158
Livermore, California 94551-0969

Jonathan W. Cruz
Center for Cyber Defenders
Sandia National Laboratories
P. O. Box 5800
Albuquerque, New Mexico 87185-MS0927

## Abstract

In this work we examine approaches for using implementation diversity to disrupt or disable hardware trojans. We explore a variety of general frameworks for building diverse variants of circuits in voting architectures, and examine the impact of these on attackers and defenders mathematically and empirically. This work is augmented by analysis of a new majority voting technique. We also describe several automated approaches for generating diverse variants of a circuit and empirically study the overheads associated with these. We then describe a general technique for targeting functional circuit modifications to hardware trojans, present several specific implementations of this technique, and study the impact that they have on trojanized benchmark circuits.

This page left blank

## TABLE OF CONTENTS

## FIGURES

**TABLES**

## EXECUTIVE SUMMARY

The use of untrusted design tools, components, and designers, coupled with untrusted device fabrication, introduces the possibility of malicious modifications being made to integrated circuits (ICs) during their design and fabrication. These modifications are known as hardware trojans. The widespread use of commercially purchased circuit designs, known as 3rd party intellectual property (3PIP) and commercial design tools extends even into trusted design flows. Unfortunately, due to the complexity of modern digital circuits, one cannot prove that a design is free of trojans or exhaustively test it to ensure that no trojans are present. Furthermore, there is no guarantee that trojan detection approaches will succeed at finding trojans. Consequently, we desire methods for mitigating or eliminating the impact of hardware trojans as an additional, complementary protection measure.

We investigate the use of implementation diversity to disrupt hardware trojans in ICs or FPGAs. We draw our inspiration for using diversity from the existing literature on system reliability, which has established that diversity in implementation, particularly when used in combination with majority voting, can be effective at mitigating the impacts of common-mode faults. Introducing implementation diversity can eliminate the presence of some of these common-mode faults, increasing system reliability. If we take the view that any deviation from the desired system behavior is a fault, then the source of these faults, be they natural, accidental, or malicious, is irrelevant. Hardware trojans have some similarity to common mode failures since a trojan introduced in a design file, whether at the register transfer or gate level, will be present in all copies of the circuit produced from that design file. It follows that implementation diversity should be effective at mitigating some malicious fault behavior, such as arises from hardware trojans.

We present several architectures for introducing diversity into digital circuits, and provide models and mathematical analysis of the expected impact of these architectures on attackers. We then empirically evaluate the cost associated with implementing these architectures by finding the area and performance overhead that results from applying them to a collection of benchmark circuits. We also provide several approaches for automatically diversifying or randomizing circuits at the netlist level, and also provide overhead results for these. Many of the diversity architectures require some method of combining the results from several variants of a circuit into a final output. We investigate tradeoffs between two traditional majority voter approaches, and describe a new approach that preserves the benefits of the traditional approaches while reducing their drawbacks. We also provide a general framework for selectively identifying and targeting portions of a design suspected of containing trojans for diversification. We describe three specific implementations of this framework. The first of these uses machine learning, the second uses template matching in netlists, and the third identifies structures in hardware description level representations of circuits. We describe implementations of these approaches and demonstrations of the last two approaches to a benchmark circuit. We also apply the machine learning approach to a collection of more than twenty benchmarks, and show that it is able to eliminate or disrupt trojans over 80% of the time.

## ACRONYMS AND DEFINITIONS

| Abbreviation | Definition |
|---|---|
| **3PIP** | Third Party Intellectual Property |
| **FPGA** | Field Programmable Gate Array |
| **IC** | Integrated Circuit |
| **LSB** | Least Significant Bit |
| **PI** | Primary Input |
| **PO** | Primary Output |
| **PRNG** | Pseudo Random Number Generator |
| **RTL** | Register Transfer Level |

# 1. INTRODUCTION

The use of untrusted design tools, components, and designers, coupled with untrusted device fabrication, introduces the possibility of malicious modifications being made to integrated circuits (ICs) during their design and fabrication. These modifications are known as hardware trojans. The widespread use of commercially purchased circuit designs, known as 3rd party intellectual property (3PIP) and commercial design tools extends even into trusted design flows. Unfortunately, due to the complexity of modern digital circuits, one cannot prove that a design is free of trojans or exhaustively test it to ensure that no trojans are present. Furthermore, there is no guarantee that trojan detection approaches will succeed at finding trojans [32, 33, 34]. Consequently, we desire methods for mitigating or eliminating the impact of hardware trojans as an additional, complementary protection measure.

In this work we investigate the use of implementation diversity to disrupt hardware trojans in ICs or FPGAs. We draw our inspiration for using diversity from the existing literature on system reliability, which has established that diversity in implementation, particularly when used in combination with majority voting, can be effective at mitigating the impacts of *common-mode* faults. Introducing implementation diversity can eliminate the impacts of some of these common-mode faults, increasing system reliability [29, 30, 31]. If we take the view that *any* deviation from the desired system behavior is a fault, then the source of these faults, be they natural, accidental, or malicious, is irrelevant. Hardware trojans have some similarity to common mode failures since a trojan introduced in a design file, whether at the register transfer or gate level, will be present in all copies of the circuit produced from that design file. It follows that implementation diversity should be effective at mitigating some malicious fault behavior, such as arises from hardware trojans.

In this report we will first introduce hardware trojans and then provide a brief overview of the broad mitigation approaches we have investigated. Then, in Section 3, we present several approaches for automatically diversifying digital circuits. Section 4 presents results from abstract modeling and analysis of the impact of diversity attackers, as well as hardware structures capable of fulfilling the assumptions of those models and experimentally determine overheads for implementing the resulting structures in hardware. Many diversity techniques make use of majority voters, so in section 5 we present an analysis of different majority voter structures, including a new one developed by this project. Section 6 describes several approaches for selectively targeting and randomizing or diversifying portions of a circuit likely to contain hardware trojans. We evaluate the voters both probabilistically and empirically using data from our diversified circuits. Section 7 presents results from formal analysis of majority voter, comparative redundancy, and comparator circuits. Finally, Section 8 describes some additional thoughts on protecting circuits from hardware trojans. These ideas were developed during the course of the project, but were not fully evaluated. They are potential avenues for future work in this area.

This page left blank

# 2. HARDWARE TROJANS

Hardware trojans typically consist of two components: some activation mechanism, usually called a *trigger,* and a *payload* that modifies the behavior of the circuit in some way. Triggers activate the trojan in response to some input to the circuit or internal circuit condition. Triggers can be combinational or sequential. They can also be time-based, sometimes referred to as a *time bomb*, in which case they activate after the circuit has operated for some specified duration of time. Trojans that are not triggered are said to be *always on*. The trojan payload may change some functional characteristics of the circuit, such as by inverting a signal in the original design. They may also consist of new circuitry, such as a mechanism for leaking information through a side-channel. Hardware trojans can have implementations as simple as changes in the dopant of existing transistors or in the geometry of wires within an IC [35]. These trojans do not require the introduction of any additional circuitry. In this work, we are primarily concerned with trojans that modify the behavior of a design or that introduce new behavior. These trojans will typically exist in register transfer level (RTL) descriptions of circuits or in circuit netlists, and could be introduced by malicious designers or design tools.

## 2.1. Hardware Trojan Taxonomies

Several hardware trojan taxonomies have been proposed [24,25]. These taxonomies typically differentiate trojans on the basis of their physical characteristics, activation or triggering conditions, and characteristics of their payload. Some further classify trojans on the basis of where in the lifecycle the trojan is inserted into the design and at what level of abstraction the trojan is implemented [23]. We find that the effectiveness of trojan mitigations are often constrained by when in the lifecycle the trojan is inserted and at what level of abstraction this occurs, so we prefer a taxonomy that includes these characteristics. Ideally, designers could achieve "coverage" of hardware trojans by applying one (or more) mitigations for each type of trojan within a comprehensive taxonomy. To this end, we map our mitigations onto the taxonomy illustrated in Figure 1.

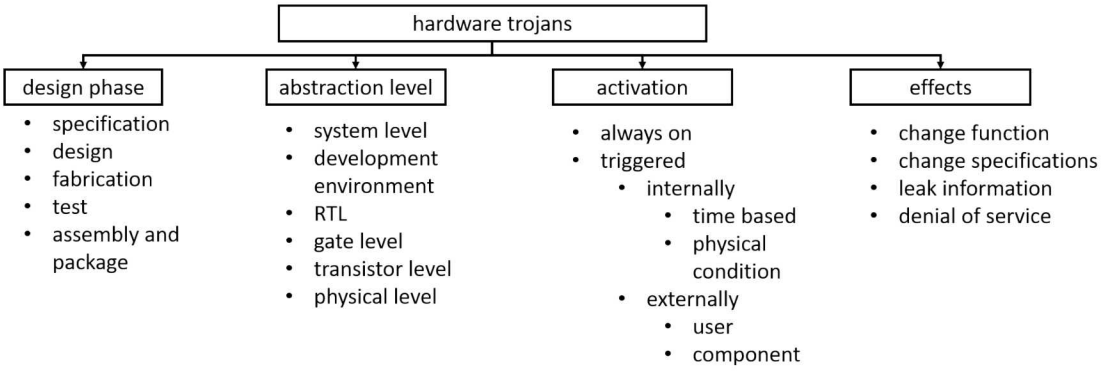


**Figure 1.** A modified version of the hardware trojan taxonomy from [23]

## 2.2. Benchmarks

### *2.2.1. Standard Benchmarks*

In Section 4 we study architectures for introducing diversity to arbitrary digital circuits. The approaches presented there preserve the logical behavior of the circuit, but may reduce an attacker's ability to successfully introduce or take advantage of a vulnerability in the circuit. They may also disrupt trojan payloads that operate on analog characteristics of the circuit, such as by leaking secret information through a side channel. For these diversity architectures we evaluate the overhead of adding diversity to the circuit. We measure overhead by the increase in the number of combinational functions required to implement the circuit in an Altera Cyclone IV FPGA and by the reduction in maximum operating frequency of the diversified circuit when compared to the original circuit. For these evaluations we use standard combinational and sequential ISCAS benchmarks [27, 28].

### *2.2.2. Hardware Trojan Benchmarks*

There are few open sources of hardware trojan benchmarks available to the research community. The results presented in this work make exclusive use of a small set of benchmarks available from [26]. Of these benchmarks, we have primarily made use of the 21 AES-128 benchmark circuits and 10 of the RS-232 benchmarks. These benchmarks and their characteristics are listed in Table 1. All of these benchmarks are provided as Verilog RTL. We use the AES-128 benchmarks for testing our mitigations, and the RS-232 benchmarks are used as training data for building machine learning models to target trojan nets in the AES-128 circuits for the mitigation described in Section 6.2.

Some of our mitigations operate at the RTL level, but the majority of them operate on gate level netlists. For the netlist-based mitigations, we synthesize and map the RTL to a small gate library consisting of 2-input NOR, NAND, OR, AND, XOR, and XNOR gates, an inverter, a buffer, and a latch. We selected this simple gate library for ease of implementation, but note that the mitigations could be implemented to target any gate library of interest. Additionally, our netlist-level mitigations can be applied to trojans inserted at the RTL or gate level. We were unable to test our mitigations on benchmarks provided by [26] as netlists since we do not have access to the gate library used in the benchmarks. Due to this, we test our netlist-level mitigations on RTL benchmarks mapped to netlists.

**Table 1.** Overview of Trojan Benchmarks

| Benchmark | Design | Abstraction | Activation | Effects |
|---|---|---|---|---|

| | **Phase** | | | |
|---|---|---|---|---|
| AES-T100 | Design | RTL | always on | leak information |
| AES-T200 | Design | RTL | always on | leak information |
| AES-T300 | Design | RTL | always on | leak information |
| AES-T400 | Design | RTL | conditional | leak information |
| AES-T500 | Design | RTL | conditional (sequential) | denial of service |
| AES-T600 | design | RTL | conditional | leak information |
| AES-T700 | design | RTL | conditional | leak information |
| AES-T800 | design | RTL | conditional (sequential) | leak information |
| AES-T900 | design | RTL | time based | leak information |
| AES-T1000 | design | RTL | conditional | leak information |
| AES-T1100 | design | RTL | conditional (sequential) | leak information |
| AES-T1200 | design | RTL | time based | leak information |
| AES-T1300 | design | RTL | conditional | leak information |
| AES-T1400 | design | RTL | conditional (sequential) | leak information |
| AES-T1500 | design | RTL | time based | leak information |
| AES-T1600 | design | RTL | conditional (sequential) | leak information |
| AES-T1700 | design | RTL | time based | leak information |
| AES-T1800 | design | RTL | conditional | denial of service |
| AES-T1900 | design | RTL | time based | denial of service |
| AES-T2000 | design | RTL | conditional | leak information |
| AES-T2100 | design | RTL | conditional (sequential) | leak information |
| RS232-T100 | design | RTL | time based | denial of service |
| RS232-T200 | design | RTL | conditional | reduce reliability |
| RS232-T300 | design | RTL | time based | leak information |
| RS232-T400 | design | RTL | conditional | leak information |
| RS232-T500 | design | RTL | time based | denial of service |
| RS232-T600 | design | RTL | conditional | leak information, denial of service |
| **Benchmark** | **Design Phase** | **Abstraction** | **Activation** | **Effects** |

| | | | | |
|---|---|---|---|---|
| RS232-T700 | design | RTL | conditional | denial of service |
| RS232-T800 | design | RTL | conditional | denial of service |
| RS232-T900 | design | RTL | conditional | denial of service |
| RS232-T901 | design | RTL | conditional | denial of service |

## 2.3. Trojan Mitigation Approaches

In this report we consider security solutions that can be used to mitigate trust problems that arise from the use of untrusted design tools, 3rd party IP or untrusted designers, and untrusted fabrication. We investigate two general approaches that apply to trojans inserted at different times during the development process. Here, we briefly introduce those approaches before entering a more detailed discussion in the following sections. Additionally, we also map these methods to the hardware trojan taxonomy of Figure 1 to provide an understanding of which threats may be addressed by these techniques, and which require mitigation by other means. This should help designers identify suites of security controls appropriate for addressing the concerns within their programs.

### *2.3.1. Trojans Introduced into Design Files*

Trojans can be introduced into RTL circuit descriptions or netlists. This can be done by malicious designers, whether they are outsiders providing 3PIP or insiders, or by the design tools used to translate the RTL into netlists. Although more challenging, trojans can also be inserted directly into netlists by malicious parties.

Identifying and correcting undesirable behavior in a circuit at a particular level of abstraction requires access to a correct description of that circuit at some other, typically higher, level of abstraction. For example, it is in principle possible to use a (correct) specification document to verify an RTL implementation, or a correct RTL implementation to verify a netlist. This is known as formal equivalence checking [36]. However, equivalence checking is not always possible. A common situation is to purchase 3PIP as a netlist. In this scenario there is no suitable alternative description of the circuit to perform equivalence checking against.

To address trojans of this nature we propose several approaches for targeting circuit modifications to portions of a design likely to contain trojans. After each modification the circuit undergoes a comprehensive simulation to ensure that the desired circuit functionality has not been impacted. If the simulated behavior is acceptable, then the circuit modification is retained. Otherwise, the modification is reverted and the next suspect portion of the circuit is modified. The modifications themselves involve selectively deleting or randomizing the implementation of some nets within a design. This approach does require access to a suitable simulation testbench for the circuit. If such a testbench is not available it may be possible to generate one by fuzzing the original design [37]. Note that we assume that the testbench only exercises the desired behavior of the circuit, and that it does not exercise any trojan behavior. This is

reasonable since trojans are assumed to be hard to trigger or identify, and those that are triggered by a simulation would presumably be detected and mitigated by other means. Our targeted circuit modification based mitigations are described in detail in Section 6. They are appropriate for addressing the following portions of the trojan taxonomy from Figure 1:

**Design phase**: design

**Abstraction level**: development environment, RTL, gate level

**Activation**: always on, triggered (all types)

**Effects**: change function, leak information, denial of service

### *2.3.2. Trojans Introduced after Design is Complete*

It is also possible for circuits to be trojanized after the design process has completed. These modifications could be made to netlists or configuration files for FPGAs, or could be introduced during the IC fabrication process. Since these trojans are inserted after the circuit designer has completed the design, there is no further opportunity to modify or randomize the circuit to disable or disrupt trojans. However, there are structures that can be built into a design to make it more difficult for a malicious actor to successfully insert or make use of trojans. Circuit obfuscation and approaches for "locking" circuits may be useful for these purposes [38-45]. We consider the introduction of diverse, redundant structures in a design to introduce redundancy, complexity, and uncertainty for attackers. These mitigations do not change the behavior of the circuit (from inputs to outputs) in any way, and so they cannot directly disrupt trojans embedded in RTL or netlists that modify the circuit functionality. However, it is possible for them to disrupt trojans that leak information through side channels because, while these approaches do preserve the logical behavior of the circuit, they do not preserve incidental analog characteristics such as timing or power consumption. Additionally, by providing a diversity of components to select from at run time these techniques can make it more difficult for attackers to ensure that a compromised subset of the components will be selected. These mitigations are described and analyzed in detail in Section 4. They are appropriate for addressing the following portions of the trojan taxonomy from Figure 1:

**Design phase**: design, fabrication, test

**Abstraction level**: development environment, RTL, gate level

**Activation**: always on, triggered (all types)

**Effects**: change function, leak information, denial of service

This page left blank

# 3. AUTOMATED DIVERSIFICATION OF DIGITAL CIRCUITS

Here, we describe several approaches for automatically diversifying gate level netlists. In this work, we use these diversification approaches in two separate ways. First, these techniques can be used to generate functionally equivalent, but physically distinct, variants of a circuit. These variants have the same logical behavior, but differ in the way that behavior is implemented. This implies that the variants also have differing analog characteristics, such as timing and power consumption, and that their fault behavior will also be distinct. This further provides the possibility of changing vulnerabilities or trojan behavior related to the specific implementation of the circuit, although it cannot impact the logical function of a trojan at the output of the circuit since logical behavior is preserved. Using these techniques to generate functionally equivalent circuits is appropriate for implementing the diversity architectures studied in Section 4. Second, these techniques can be used to modify the logical behavior of a circuit. In this approach the functional behavior of the circuit is changed, and so it becomes possible to alter not only the analog characteristics of the circuit, and any trojans within it, but also the functional behavior of the circuit and its trojans. Many, and perhaps even most, functional changes to a circuit's behavior will result in modifications to the intended behavior of the circuit and will prevent the circuit from performing its desired function. Since we desire to preserve the intended circuit functionality while selectively disrupting the functionality of trojans or other portions of the circuit that are extraneous to the desired functionality, such as might arise when not all portions of a complex IP core are needed in a particular implementation, modifying the logical behavior of the circuit is more appropriate when used in conjunction with one of the targeting techniques described in Section 6.

With these diversification approaches we first select a small logic cone within the circuit, and then apply the modification approach to the small sub-circuit defined by this logic cone. These diversification approaches all result in some change to the functionality of the logic cone as defined from its primary inputs to its primary outputs. Consequently, when we are creating functionally equivalent variants we need to determine the truth table for the original cone before any modification is made. Then, after the modification has been made we identify a reconvergence circuit, which has as its inputs the primary inputs from the original logic cone and the primary outputs from the modified cone, and produces as its output the expected behavior from the original logic cone. We add this reconvergence circuit to the logic cone to undo any functional modifications to the primary outputs of the original logic cone, recovering the intended behavior. After these modifications are made the updated logic cone is reincorporated into the circuit and we check for combinational and sequential equivalence between the original and modified circuits. If either of these fail, then the modification is reverted. If we are not creating functionally equivalent variants then the reconvergence circuit is not needed.

For ease of implementation, in this work we always map circuits to a reduced gate library consisting only of an inverter and two input AND, NAND, OR, NOR, XOR and XNOR gates, as well as a buffer and a latch.

### 3.1. Gate Addition

For gate addition we randomly select a two input logic gate and incorporate it into a logic cone, as illustrated in Figure 2 [17]. We randomly select an existing gate within the logic cone and use its output as one of the inputs to the new gate. The second input to the new gate is a randomly selected node from the circuit that is not already in the logic cone. If we are preserving the original behavior of the logic cone, then an appropriate reconvergence circuit is added as well.

We experimentally determine the area and performance overhead for gate addition by applying the technique to a collection of ISCAS benchmark circuits*. We measure area overhead by the change in the number of logic elements required to implement the circuit in an Altera Cyclone IV FPGA, and the performance overhead by the change in maximum operating frequency ($f_{max}$) of the circuit. To implement the gate addition we first select a random node in the circuit and trace its transitive fanout for three levels of logic. If the fanout exceeds 20 gates then a new random node is selected. If the fanout cone has fewer than 20 gates, then we add a new gate to the logic cone. Then we find an appropriate reconvergence circuit and incorporate it and the modified logic cone back into the circuit. We repeat this process ten times, so that ten gates are added to the circuit. The overhead results are presented in Figure 3 and Figure 4. From these figures we see that adding 10 gates and corresponding reconvergence circuits to these benchmarks increases area by an average of about 5% and has negligible impact on $f_{max}$, with the average circuit seeing about a 3% reduction in $f_{max}$ after gate addition.

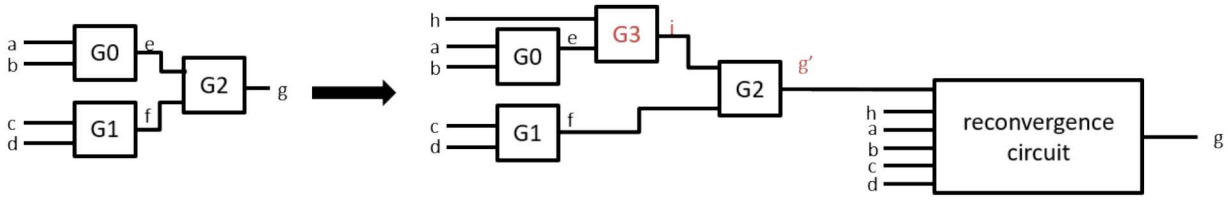


**Figure 2.** Gate addition involves adding a randomly selected gate to a logic cone. Then, a reconvergence circuit is added at the output of the cone to recover the cone's original functionality.

*b01, b03, b04, b06, b07, b08, b09, b10, b11, b12, b13, b14_1, b15_1, b17_1, b20_1, b21_1, b22_1, c432, c499, c880, c1355, c1908, c3540, c5315, c6288, s208, s298, s344, s349, s382, s386, s386a, s400, s420, s444, s510, s526, s526a, s641, s713, s820, s832, s953, s1196, s1196a, s1196b, s1238, s1238a, s1423, s1488, s1494, s5378, s9234, s13207, s15850, s35932, s38417, s38584

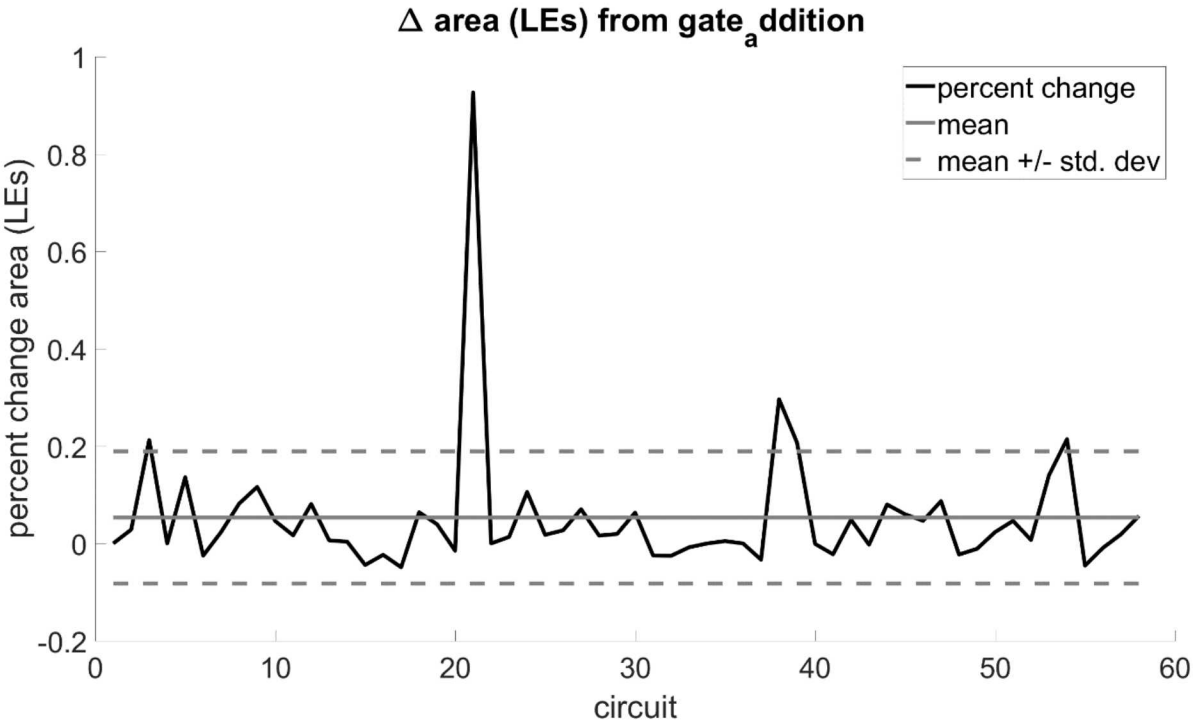


**Figure 3**. Area overhead from gate addition

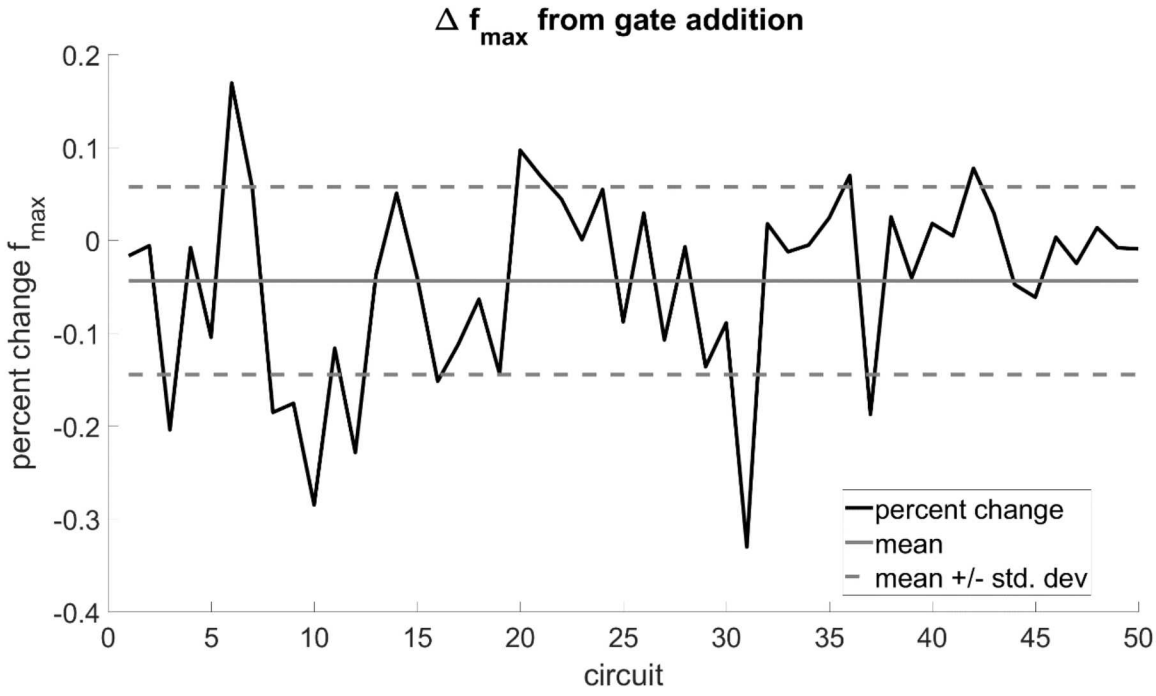


**Figure 4**. Performance overhead from gate addition

## 3.2. Gate Replacement

For gate replacement we randomly select a two-input gate within the logic cone and replace it with a different randomly selected two-input gate [17]. If we are preserving the behavior of the original logic cone then we also find an appropriate reconvergence circuit to append to the logic cone. This approach is shown in Figure 5.

To implement gate replacement we choose a random node in the circuit and then trace its transitive fanout cone for three levels of logic. If this transitive fanout cone exceeds 20 gates, then a new initial node is randomly selected. After extracting the logic cone, we then randomly select three two-input gates from it and replace each of them with a different two-input gate. If there are three or fewer two-input gates in the logic cone then we replace all of the input gates. After making these replacements we find an appropriate reconvergence circuit and add it to the logic cone. Then this modified cone is reincorporated into the original circuit. We repeat this process ten times, so up to thirty gates are replaced.

We experimentally determine the area and performance overhead for gate addition by applying the technique to the same collection of ISCAS benchmark circuits as used in Section 3.1. Overhead results are shown in Figure 6 and Figure 7. From these figures we see that gate replacement has similar area overheads as gate addition, with an average increase of about 3%, and also has negligible impact on $f_{max}$, with the average circuit showing about a 3% decrease in $f_{max}$ after gate replacement.

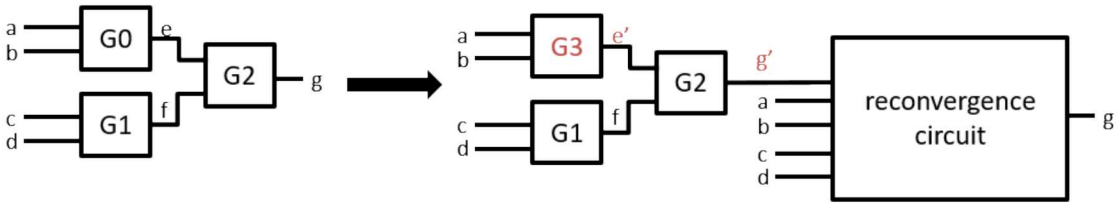


**Figure 5**. Gate replacement involves replacing a gate within a logic cone with a randomly selected gate. Then, a reconvergence circuit is added at the output of the cone to recover the cone's original functionality.

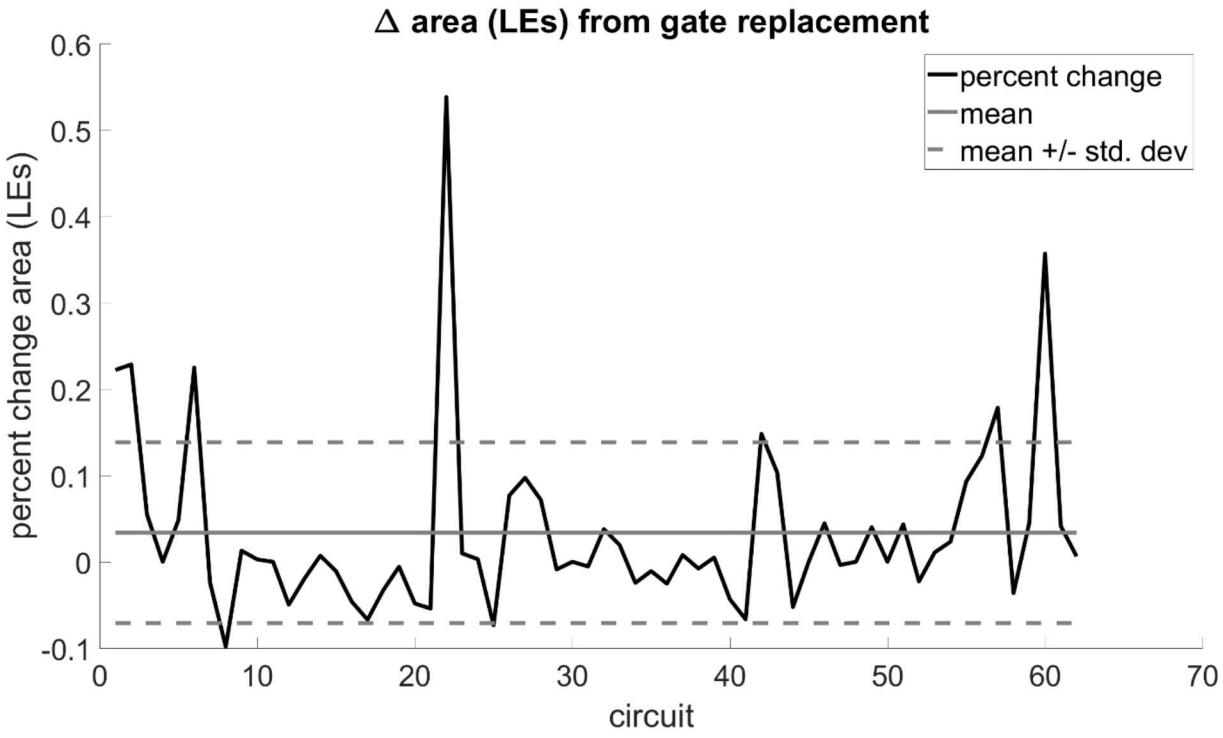


**Figure 6.** Area overhead from gate replacement

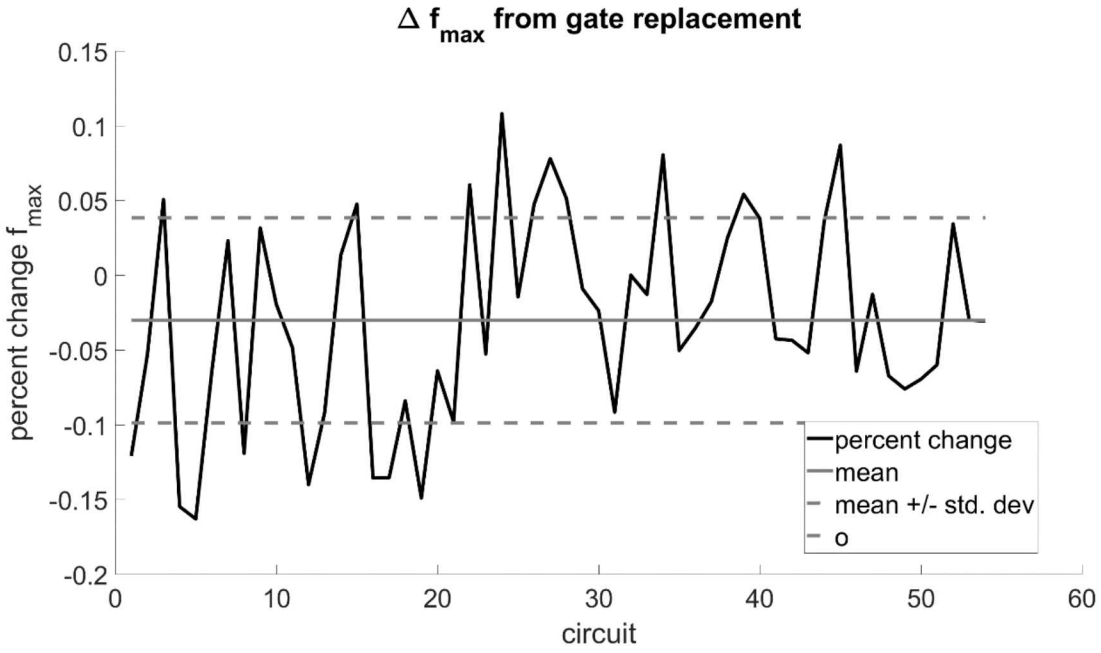


**Figure 7.** Performance overhead from gate replacement

### 3.3. Dynamic Output Inversion

In the dynamic output inversion technique we randomly select a target gate within the logic cone and attach its output to one input of an XOR gate [18]. The second input to the XOR gate is a random function of 2 to 4 of the primary inputs to the logic cone. For this random function we select an appropriate number of random two-input logic gates to implement a function of the 2 to 4 inputs to one output. This results in the original output of target gate being inverted whenever the output of the randomly selected function is '1'. If we choose to preserve the original behavior of the logic cone then an appropriate reconvergence circuit is added. This technique is illustrated in Figure 8.

To implement dynamic output inversion we pick a random node in the circuit and trace its transitive fanout cone for three levels of logic. If the resulting logic cone has more than 20 gates then we discard it and choose a different starting node. After finding a suitable logic cone we then partition it into two disjoint subcircuits. We then find an output from the first subcircuit that is an input to the second. This is the signal that we will selectively invert. We also find which inputs to the original logic cone are also inputs to the first subcircuit. If there is only one such input then we start over by picking a new random node in the circuit and then finding its transitive fanout cone. If there are two such inputs, then we create a random function of them by combining them with a random two-input gate. If there are three such inputs, then we choose random gates. Two of the inputs are attached to the first random gate, and then the output from this gate and the third input are combined with the second gate. If there are four or more such inputs then we combine the first four of these with three randomly selected gates. The first two inputs are combined with the first gate, and the second two with the second gate. Then the outputs from these are combined with the third gate. In any case, we XOR the output of this random circuit with the previously signal that is an output from the first subcircuit and an input to the second. We then find an appropriate reconvergence circuit and add it to the modified logic cone. Finally, we reincorporate the modified cone and reconvergence circuit into the original circuit. We repeat this entire process ten times.

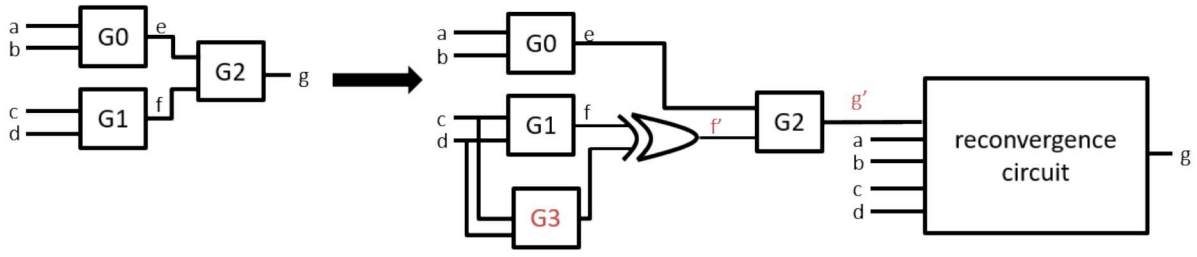


**Figure 8.** In dynamic output inversion the output of a gate within a logic cone is selectively inverted as a function of the inputs to the logic cone. Then, a reconvergence circuit is added at the output of the cone to recover the cone's original functionality.

We experimentally determine the area and performance overhead for dynamic output inversion by applying the technique to the same collection of ISCAS benchmark circuits as used in Section 3.1. Overhead results are shown in Figure 9 and Figure 10. From these figures we see that this approach has larger area overheads than gate addition and gate replacement, with an average increase of about 100%. This technique also has a larger impact on $f_{max}$ than gate addition and gate replacement, with the average circuit's $f_{max}$ decreasing by more than 30%.

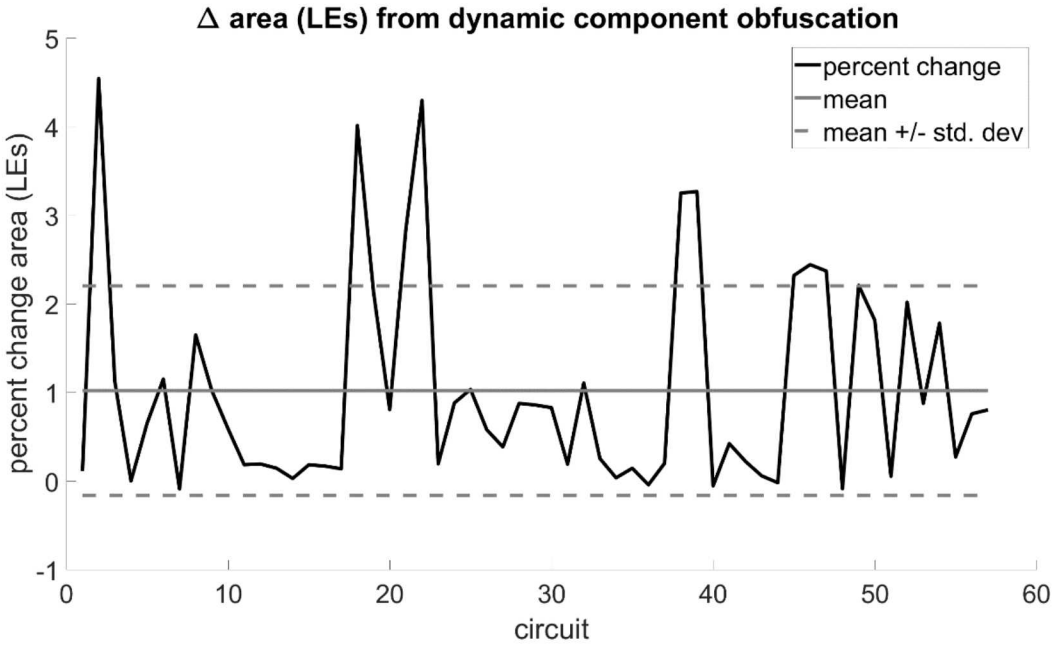


**Figure 9.** Area overhead from dynamic output inversion

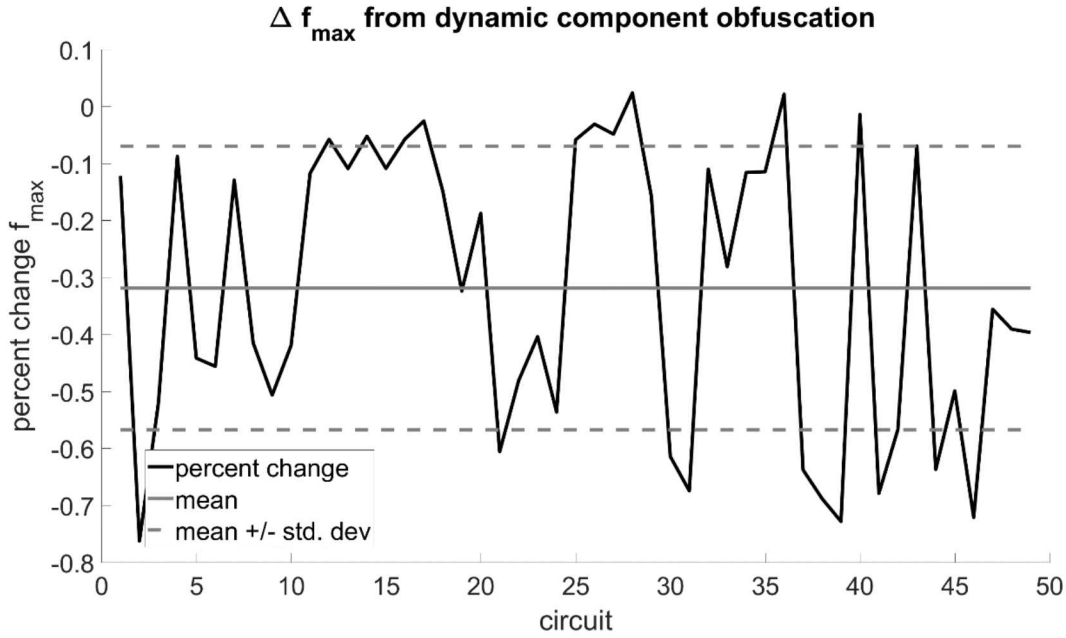


**Figure 10.** Performance overhead from dynamic output inversion

## 3.4. Column Exchange

The column exchange technique is effectively a bus permutation [18]. It is implemented by selectively swapping one output for another under some input conditions, as shown in Figure 11. The technique can be implemented with controlled crossbar switches that swap the output bits when the control condition is met.

To implement column exchange we pick a random node in the circuit and trace its transitive fanout cone for three levels of logic. If the resulting logic cone has more than 20 gates or fewer than 8 then we discard it and choose a different starting node. After finding a suitable logic cone we partition it into two subcircuits by placing the first half of the gates in one subcircuit and the second half of the gates in the second. Then, we verify that there are at least two gates in the first circuit whose outputs are inputs to the second circuit. If this is not the case, then we abandon this cone and choose a new random starting node. If there are at least two such gates, then we introduce a crossbar switch that swaps the outputs of these gates as a function (randomly chosen XOR or XNOR) of two of the primary inputs to the logic cone that are also inputs to the first subcircuit. We then find a suitable reconvergence circuit to append to the cone, and incorporate the modified cone back into the original circuit. We repeat this process ten times, so that ten logic cones in the original circuit are modified by the column exchange technique.

We found the area and performance overhead for column exchange by applying it to the same collection of ISCAS benchmark circuits as used in Section 3.1. Overhead results are shown in Figure 12 and Figure 13. From these figures we see that this approach has larger area overheads than gate addition and gate replacement, with an average increase of about 60%. This technique also has a larger impact on $f_{max}$ than gate addition and gate replacement, with the average circuit's $f_{max}$ decreasing by almost 30%. These results are similar to those for dynamic output inversion.

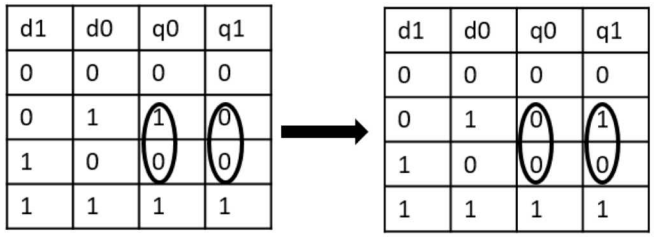


**Figure 11.** In the column exchange approach output bits are selectively swapped

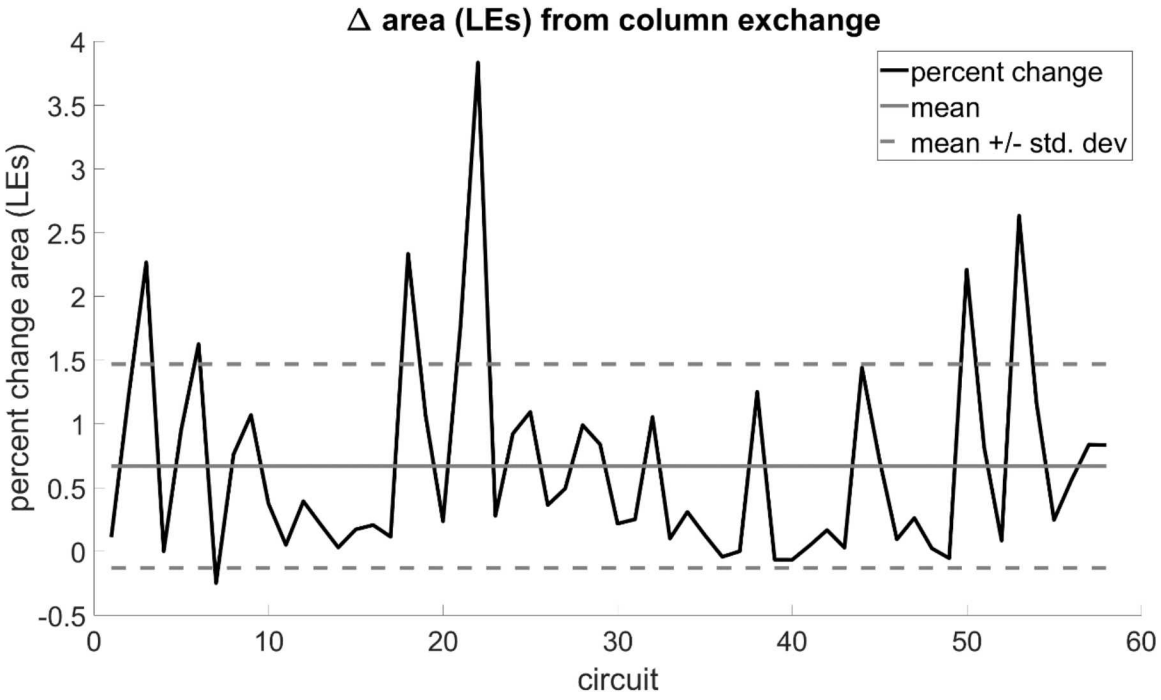


**Figure 12.** Area overhead from column exchange

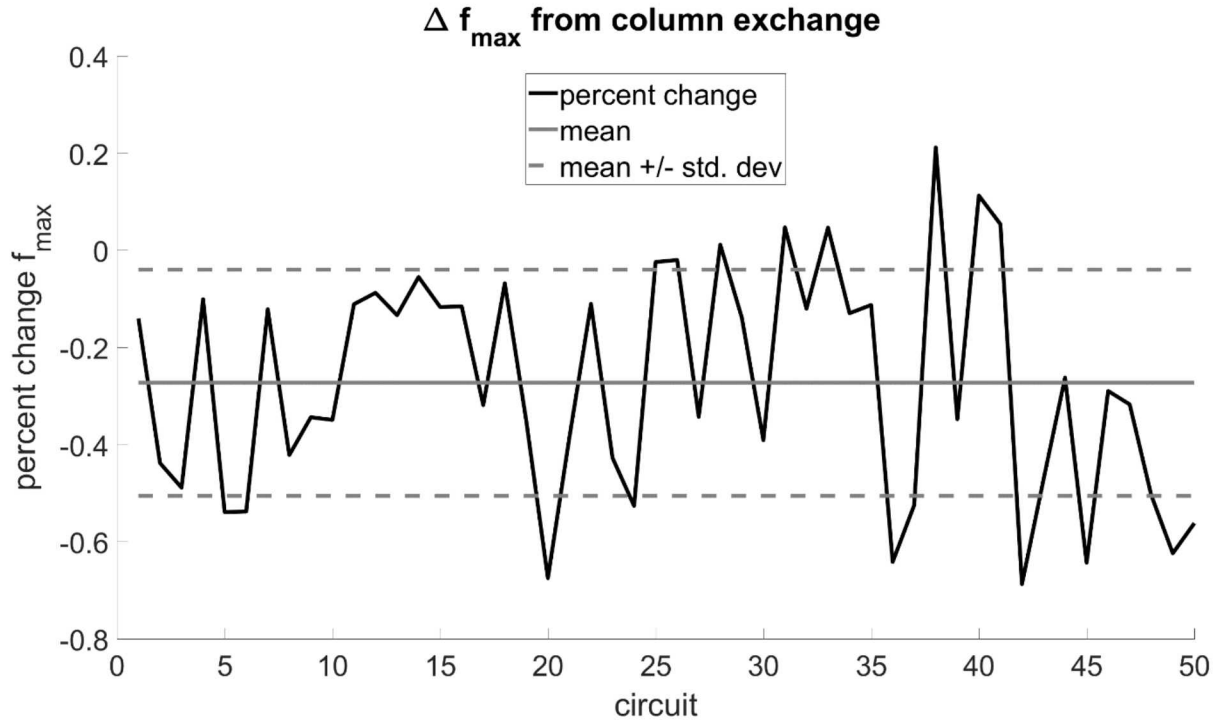


**Figure 13.** Performance overhead from column exchange

## 3.5. Approximate Circuits

In the approximate circuits approach a circuit with intended function $F$ is implemented by creating a collection of circuits $F_1, F_2 ... F_{N-1}$, each of which deviates from the functionality of $F$ for a small number of input conditions. Since these functions deviate from the intended behavior for some portion of their inputs we say that they approximate $F$. We constrain the $F_1, F_2 ... F_{N-1}$ by requiring that a majority vote on their outputs results in the intended function $F$. This approach was previously studied as a mechanism for enhancing circuit reliability [15].

To implement the approximate circuits approach we choose a random node in the circuit and trace its transitive fanout cone for three levels of logic. If the resulting logic cone has more than 40 gates then we discard it and choose a different starting node. Similarly, we also discard the cone if it has fewer than 5 gates or more than 12 inputs. After extracting a suitable logic cone we find its truth table and create three copies of the cone. We then create the approximate circuits by inverting the outputs of the first copy of the cone for the first third of the rows in the truth table. Similarly, the second copy of the cone has its outputs flipped for the second third of the rows in the truth table, and the third copy has its outputs flipped for the final third of the rows of the truth table. These variants of the cone are then combined by introducing a majority-3 voter for each output bit of the logic cone. Since for every row in the truth table the outputs are modified in a single copy of the cone, this approach ensures that the output of the majority-3 voter will be the intended function. We repeat this entire process ten times, so that ten logic cones in the original circuit are replaced by approximate circuits implementations.

We found the area and performance overhead for the approximate circuits approach by applying it to the same collection of ISCAS benchmark circuits as used in Section 3.1. Overhead results are shown in Figure 14 and Figure 15. From these figures we see that this approach has much larger area overhead than the other techniques, with the average diversified circuit requiring almost 9X as many logic elements as the original circuit. This technique also has significant impact on $f_{max}$, with the average circuit's $f_{max}$ decreasing by almost 40%. This reduction in $f_{max}$ is similar to those for dynamic output inversion and column exchange. We note that generating the approximate circuits in this way will usually result in sub-optimal implementations, and that more sophisticated approaches for determining which outputs to invert might result in considerably less overhead.

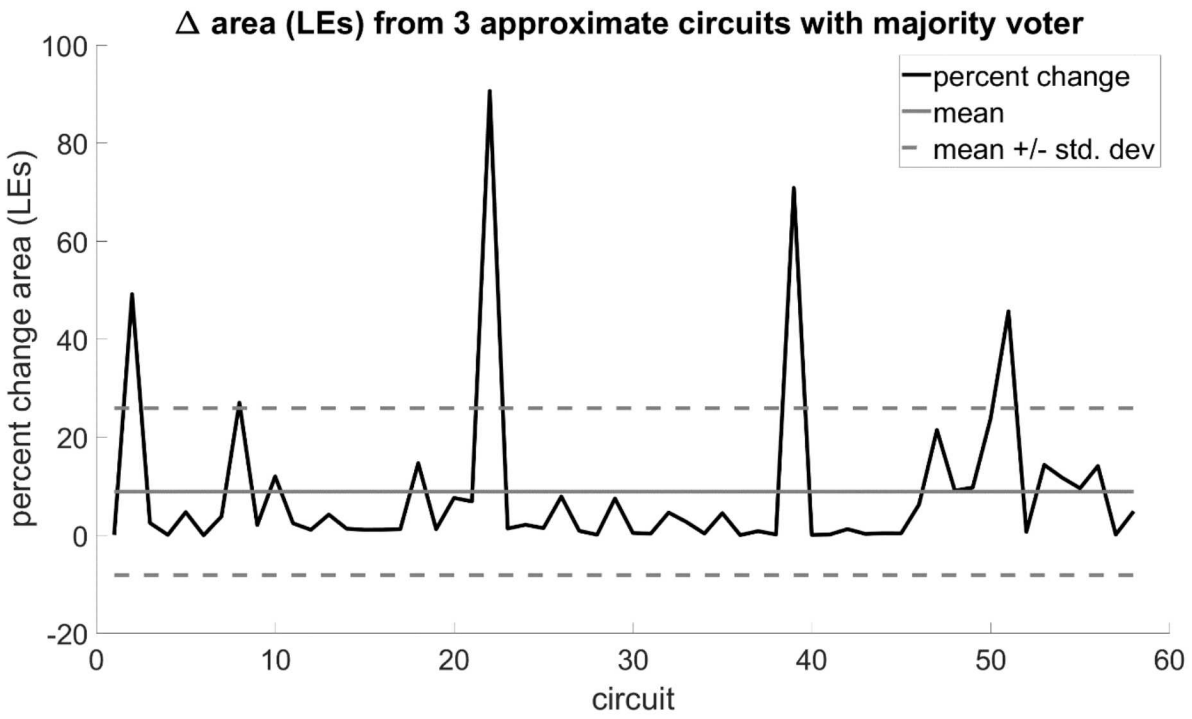


**Figure 14**. Area overhead from approximate circuits

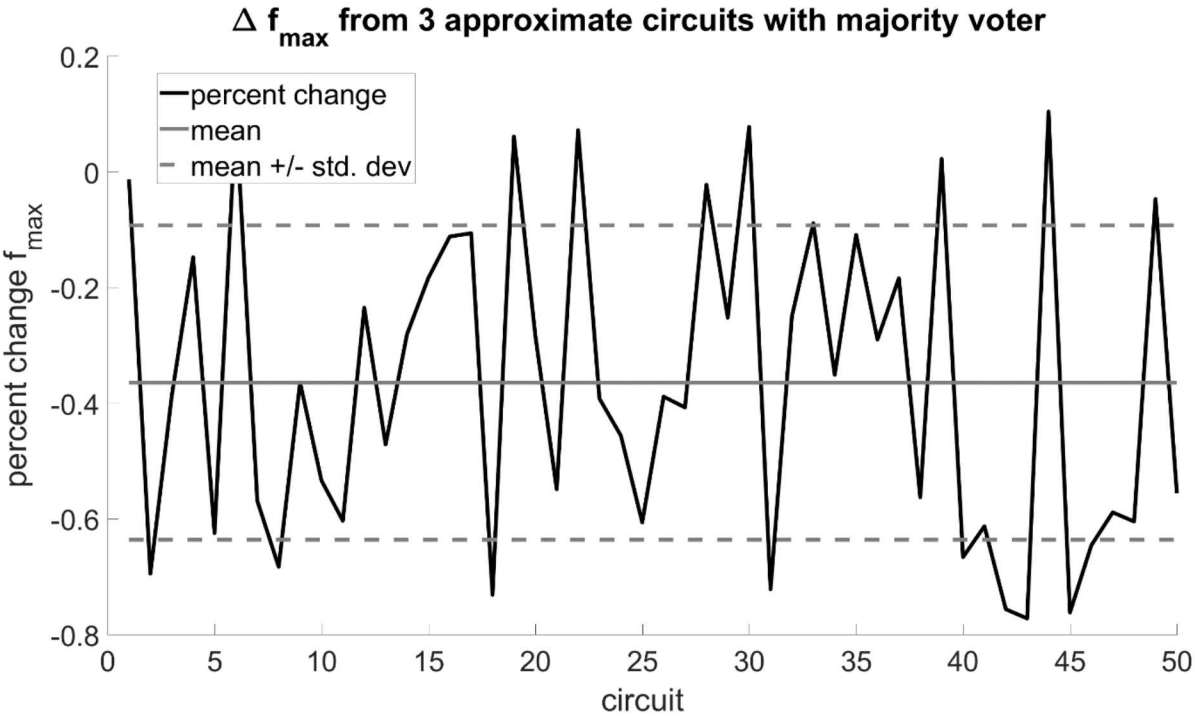


**Figure 15.** Performance overhead from approximate circuits

## 3.6. Polymorphic Gates

A polymorphic logic gate is a gate that can change its functions in response to some external condition. We consider functionally polymorphic gates, which are simply logic gates whose behavior is determined by a set of configuration bits [39]. While there are many implementations of a functionally polymorphic gate, it is easiest to think of a simple 4:1 MUX structure, which can implement any logical function of two variables by assigning the input variables to the two select bits of the MUX and then defining the desired function at the four data inputs. In practice we use a more complicated implementation that defines the polymorphic gates with a truth table. To do this we define the polymorphic gate we create a function with five inputs and one output. Two of the inputs are the inputs to the original logic gate. The remaining three inputs are key bits that can are randomly assigned, and are the same for each row in the truth table. The value of the output is determined by the logic gate that we wish to implement. After defining this truth table we then implement the polymorphic gate by finding a circuit that implements the truth table. An example truth table for a polymorphic AND gate with key bits $k_0$, $k_1$, $k_2$, input bits $d_0$ and $d_1$, and output bit $q_0$ is:

| $k_2$ | $k_1$ | $k_0$ | $d_1$ | $d_0$ | $q_0$ |
|---|---|---|---|---|---|
| $k_2$ | $k_1$ | $k_0$ | 0 | 0 | 0 |
| $k_2$ | $k_1$ | $k_0$ | 0 | 1 | 0 |
| $k_2$ | $k_1$ | $k_0$ | 1 | 0 | 0 |
| $k_2$ | $k_1$ | $k_0$ | 1 | 1 | 1 |

To implement the polymorphic gates we pick a random two-input gate from the circuit. We then define a random polymorphic implementation for that gate as described above by randomly choosing the key bits and generating the corresponding truth table. We then replace the selected gate with the polymorphic implementation. The key bits can be added as new primary inputs to the circuit or hardcoded. We choose to hardcode them in our implementation. We repeat this process ten times, so that ten gates are replaced with keyed polymorphic implementations.

We found the area and performance overhead for the polymorphic approach by applying it to the same collection of ISCAS benchmark circuits as used in Section 3.1. Overhead results are shown in Figure 16 and Figure 17. From these figures we see that this approach has only a small area overhead, averaging about 2%. This technique also has a negligible impact on $f_{max}$, with the average circuit's $f_{max}$ changing by less than 0.5%. This approach has the smallest area overhead of the diversification approaches we considered, and essentially no impact on performance.

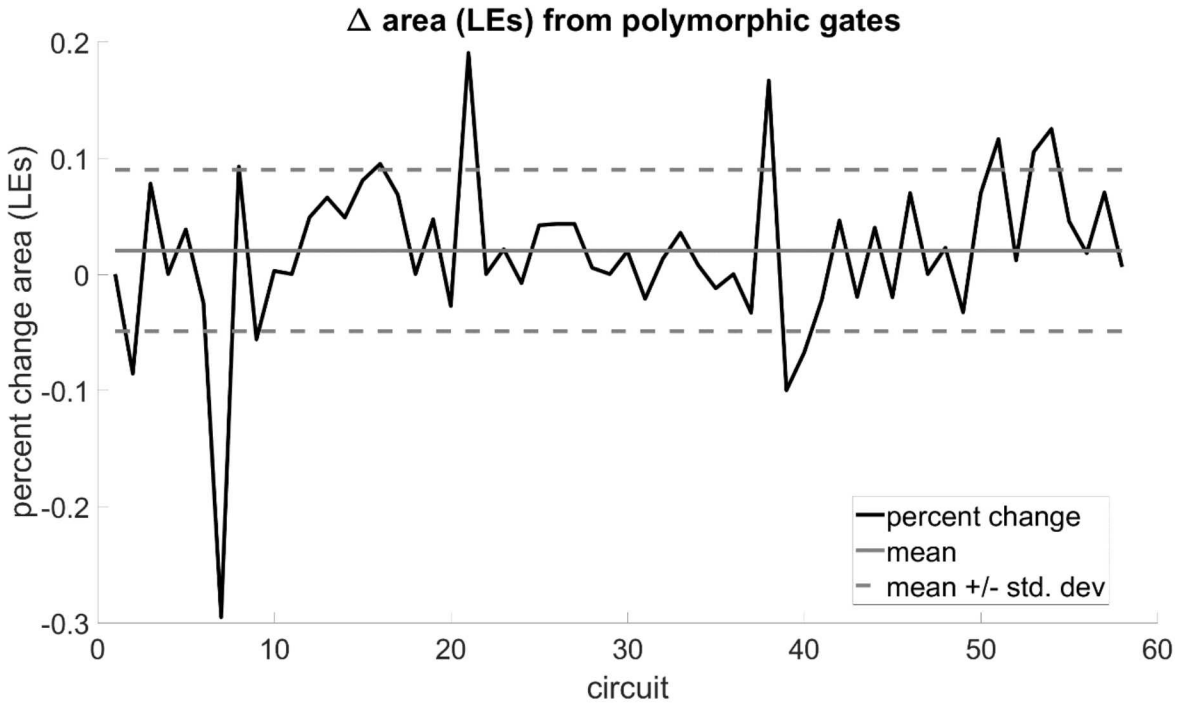


**Figure 16.** Area overhead from polymorphic gates

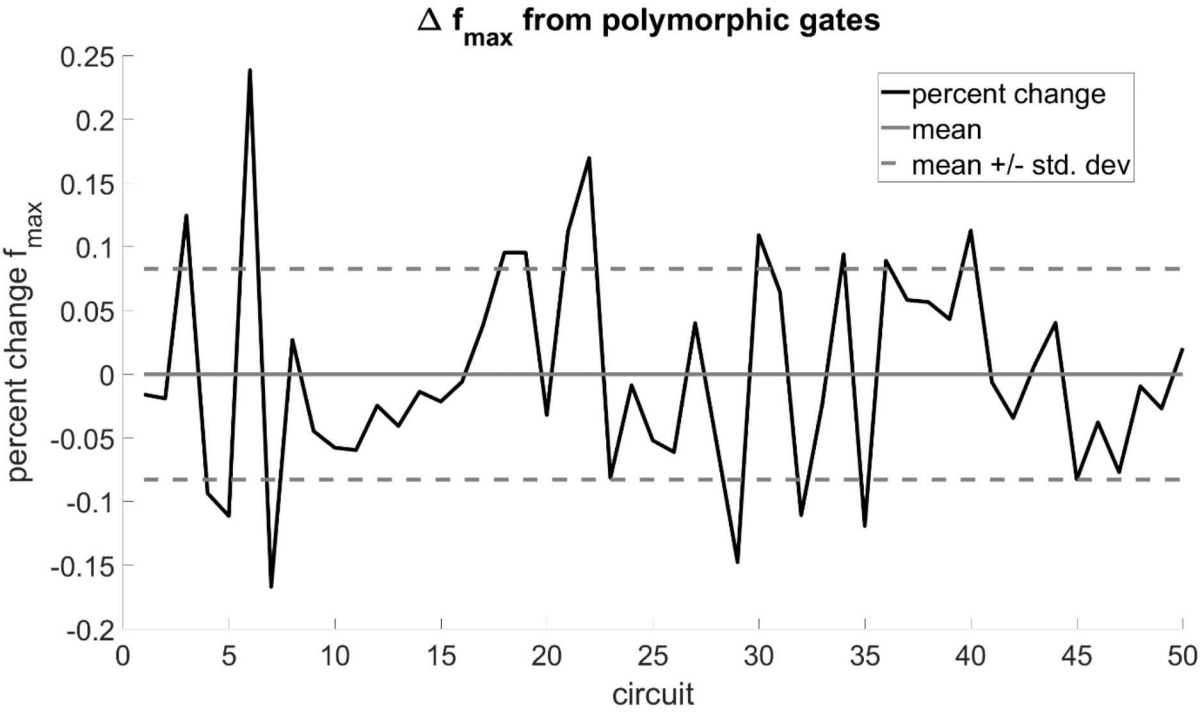


**Figure 17.** Performance overhead from polymorphic gates

Table 2 provides a relative comparison of the area and performance overheads for these six diversification approaches. Gate addition, gate replacement, and polymorphic gates have the lowest overall overheads. Dynamic output inversion and column exchange have moderately larger overheads than gate addition and gate replacement. The approximate circuits approach has performance overhead comparable to that of dynamic output inversion and column exchange, but the area overhead in our current simplistic implementation is an order of magnitude larger than that from dynamic output inversion or column exchange.

**Table 2.** Relative comparison of overhead from various circuit diversification approaches

| | **Gate Addition** | **Gate Replacement** | **Dynamic Output Inversion** | **Column Exchange** | **Approximate Circuits** | **Polymorphic Gates** |
|---|---|---|---|---|---|---|
| **Area** | low | low | medium | medium | very high | low |
| $\mathbf{f_{max}}$ | low | low | medium | medium | medium | low |

# 4. MODELING AND ANALYSIS OF THE IMPACT OF DIVERSITY ON ATTACKERS

Diversity and redundancy in hardware and software systems have previously been studied as a means of enhancing the security of systems by limiting the impact of vulnerabilities in those systems. Diversity in implementation can eliminate some vulnerabilities and make it uncertain whether a given implementation will have a particular vulnerability [1]. We consider the more general scenario of an attacker purposefully inserting trojans or other malicious artifices. Diversity is of interest because, while some properties of a system's design can be proved exhaustively [2], other "incidental" vulnerabilities still exist due to the particular way the system is implemented, and possibly subverted, at a lower level [3]. A general technique to quantifiably mitigate such remaining, *a priori* unknown vulnerabilities is to disrupt adversaries' ability to analyze and attack a specific implementation at their leisure, by introducing design elements that are random or otherwise cannot be anticipated by the attacker. This is known as a moving target. The case we focus on here is voting several diverse implementations of the same function, so that a majority must be compromised simultaneously to successfully attack the system. There is broad evidence from previous work that moving targets, including diverse voting systems, can increase the difficulty of attacks [4].

Our focus is on practical aspects of using diversity to reduce the utility of hardware trojans. We place our emphasis on design-time constructs. Such approaches can be repeated from design to design and are relatively easily implemented and studied when compared to incorporating diversity during other portions of the lifecycle, such as fabrication. We also have broad control over our own design processes. It is also likely that design-time diversity can be automated, reducing the impact of diversity on system designers. To incorporate design-time diversity we need to identify frameworks for incorporating diversity into our systems and methods for creating that diversity. In this section we present and analyze several such frameworks, and evaluate the area and timing overheads that result when they are applied to a collection of benchmark circuits.

We have developed models for abstractly studying the impact of various methods of incorporating diversity into circuit designs, and have used these to craft probabilistic expressions for studying the utility of the diversity frameworks over a range of conditions. We begin with a routing model in which data is processed from the input to the output of a circuit through several "tiers" of subcircuits. For example, tiers may represent pipeline stages or distinct processing units. If we permit several diverse implementations of each tier then we obtain a diversity of potential processing paths. A single path is realized by selecting one unit from each tier. The attack succeeds if any unit in the selected path has been subverted. Data can be processed along more than one path in parallel to create a diverse voting system. By taking a majority vote on the output of the distinct paths we can ensure that the output of the voter is correct as long as fewer than half of the paths contain a subverted unit. This initial analysis assumes that only one node in the system is compromised. If two or more distinct processing paths pass through the same vulnerable node, then we interpret this as a common-mode failure in which insufficient diversity was applied to restrict the

subversion to a single implementation. We present a probability relation that models this scenario.

We next present a model in which the circuit is again divided into a number of components, each of which can be diversified. We consider the scenario in which three diverse implementations of the circuit are constructed and the output of the circuit is voted on, which is conceptually similar to the previous model, in addition to the scenario in which individual subcircuits are diversified and the subcircuit outputs are voted on locally. We then generalize the second scenario to one in which only a subset of the subcircuits is diversified. We present probability relations for these models as well.

Finally, we consider a moving target model in which the attacker must be successful at least $m$ times within $n$ tries for the attack to succeed, and in which the system is modified after each of the attacker's attempts. We present a broadly applicable probability relation to model this scenario.

For each of these models, we also provide abstract hardware architectures that realize the framework. We then apply these architectures to a collection of benchmark circuits to augment our probabilistic understanding of the frameworks' benefits with an experimental understanding of the cost and performance overheads required to implement them.

## 4.1. Theoretical Security Effectiveness of Diverse Voting and Moving Targets in Component Architectures

### *4.1.1. Assumptions about Component Vulnerabilities*

A digital system is intended to implement a specific mapping of inputs to outputs. Incorrect function of the system for a particular input is interpreted as a vulnerability. The system is made up of $m$ components, and correct function of the system requires correct function of all the components. Each component responds deterministically to the system's input, and functions incorrectly for a fraction $f$ of the possible inputs due to implementation defects. We assume that implementations of the component can be generated that are vulnerable to the same extent but on different, random subsets of inputs.

Thus, for any chosen input, a fraction $f$ of random implementations (variants) of the component, sampled uniformly, will be vulnerable. This is the key condition we rely on, and as long as it is satisfied, we can relax the assumption that each variant is vulnerable on exactly a fraction $f$ of inputs. Instead, the vulnerability fraction could be greater for some variants of the component and less for others, provided that it averages to $f$ and is equally distributed over all inputs.

As an example of a more specific scenario with these properties, we can suppose: $f = 2^{-k}$, a component vulnerability is triggered by a particular $k$-bit segment of the input matching a particular $k$-bit pattern, and this pattern is uniformly randomized among variants. We call $k$ the key length.

Given these assumptions, we wish to compute the probability that an attacker succeeds in compromising the system, i.e., the system produces an incorrect output for the attacker's input. The most basic relation is the following: For an attacker trying one input, on a system constructed using a random (unknown to the attacker) variant of each component, the probability of attacker success is the probability that at least one component functions incorrectly for the chosen input, which is

$$p = 1 - (1 - 2^{-k})^m \quad (1)$$

For $2^{-k}m \ll 1$, this reduces to

$$p \approx 2^{-k}m \quad (2)$$

The most basic security benefit of randomized implementations is seen because, if instead the attacker did know the particular component implementations used, the attacker could choose an input to which one component is vulnerable and succeed with certainty $(p = 1)$.

### 4.1.2. Routing Model

Diversity is of interest because, while some properties of a system's design can be proved exhaustively [12], other "incidental" vulnerabilities still exist due to the particular way the system is implemented, and possibly subverted, at a lower level [13]. The only existing general technique to quantifiably mitigate such remaining, *a priori* unknown vulnerabilities is to disrupt adversaries' ability to analyze and attack a specific implementation at their leisure, by introducing design elements that are random or otherwise cannot be anticipated by the attacker. This is known as moving target defense. The case we focus on here is voting several diverse implementations of the same function, so that a majority must be compromised simultaneously to successfully attack the system. There is broad evidence from previous work that moving target, including diverse voting systems, can increase the difficulty of attacks [14, 15].

Diversity can be applied in a digital system design process in a variety of ways, and correspondingly the specific modeling of that diversity can take many forms. We initially sought a simple and tractable example. Our model is described and analyzed below in terms of a design for routing of messages in a communications fabric, with diversity implemented as randomized routes between nodes to mitigate the consequences of subverting any particular node. However, we view this routing model of diversity as a pattern that could be relevant at various stages in system lifecycles involving chained dependencies that can be satisfied, either at design time or at run time, by different combinations of partially redundant, possibly subverted resources – e.g., multiple servers, compilers, or testers.

The structure of our notional communications fabric is shown in Figure 18. The concept is that a message being transmitted from Origin to Destination passes through each "tier" of nodes in between. A specific route is a choice of which node in each tier receives and passes along the message. The number of possible routes for the example in Figure 18 is $10^6$, because any of 10 nodes can be used in each of the 6 tiers. We

assume that the message will be corrupted, as desired by the attacker, if any of the nodes it passes through has been subverted.

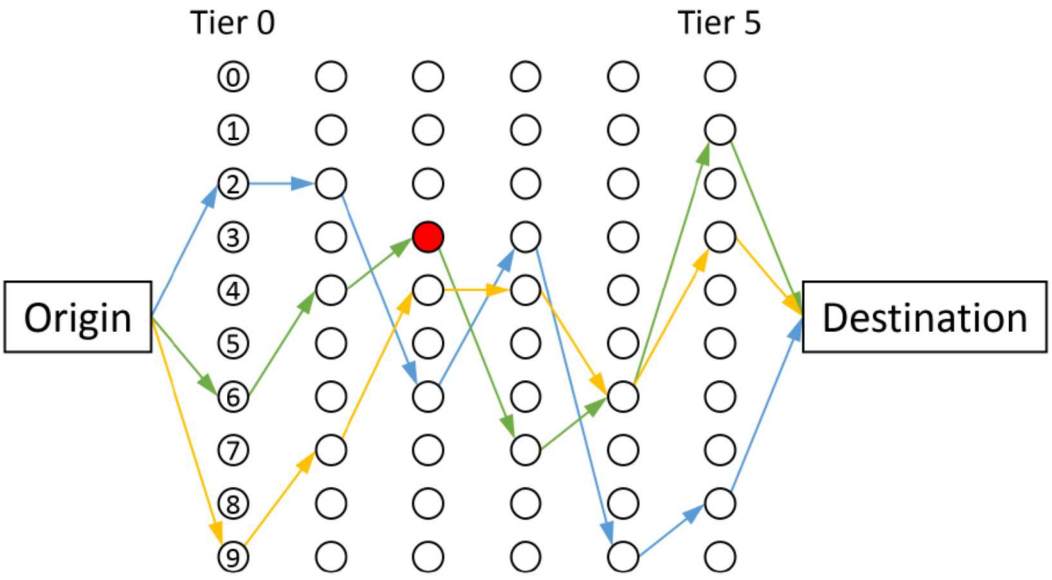


**Figure 18.** Diagram of message routing model. Nodes in each tier are numbered 0 through 9. Three example routes are shown by arrows. A node subverted by an attacker is shown in red; in this instance, `attack_tier = 2` and `attack_node = 3`. Of the routes shown, only the green route suffers message corruption in this instance due to passing through the subverted node.

A given message could also be transmitted along more than one route in parallel; this is how we instantiate a diverse voting system. Our simple approach for mitigating the possible corruption due to a subverted node is to take a majority vote of the contents of the message as received at Destination from the multiple routes. Thus, such a multiply routed message is considered as being successfully attacked if at least half of the routes used pass through a subverted node.

Our analysis assumes an attack model in which only a single node (in a single tier) can be subverted. This is motivated by the idea that, given differences in the implementation or provenance of each individual node, simultaneously subverting more than one node is substantially more difficult and less likely than subverting just one. For example, if the attack involves identifying a specific message or environmental condition that triggers a latent vulnerability in a node, differently implemented (and well-tested) nodes are unlikely to have a trigger in common. Or, if the attack involves a deliberately inserted, rarely manifested flaw in a node implementation, sourcing the nodes from independent suppliers makes it unlikely that more than one of them would exhibit the rare flaw simultaneously.

Ultimately, any application of diversity is exposed to the potential for "unknown unknown" common-mode vulnerabilities. When the entire system cannot be formally

proved, we must resort to some argument that diversity has been applied at the relevant semantic levels to mitigate vulnerabilities of concern. Our analysis here targets a specific increment of design: We quantify the effect of diversity in message routing, given the requisite diversity in the underlying nodes. A more comprehensive diversity approach would apply these concepts iteratively through multiple semantic levels.

We start with a model intended to measure the benefit of diverse voting in the simplest possible terms. We grant the attacker advance knowledge of our choice of routing, and ask whether the attacker can compromise the design by subverting any one node. Clearly, if we rely on any single route, the attacker can accomplish this by subverting any one of the nodes this route passes through. For example, if we used the green route in Figure 4.1, the attacker could succeed by subverting the node shown in red.

On the other hand, if we use redundant routing, the attacker can succeed against the voting system only if there is a single node through which at least half of the routes pass. There are many ways to choose three diverse routes, for example, such that no two have a node in common, thereby ensuring that the attacker cannot succeed in our model. Note that the blue, green, and gold routes in Figure 4.1 illustrate a case that does not satisfy this condition (both the green and gold routes use node 6 in tier 4). The number of nonoverlapping triple routes can be computed as follows: At each of the 6 tiers, there are 10 choices for the first route, 9 choices (avoiding the node already used) for the second route, and 8 choices for the third route. Thus the total number is $(10 \text{ x } 9 \text{ x } 8)^6 \approx 1.4 \text{ x } 10^{17}$. This model was initially developed by a prior LDRD project [16].

If the attacker knows the routing, the probability of attacker success, *P(s)*, for a single route is 1 and for three routes is

$$P(s) = 1 - \left(1 - n\left[\binom{3}{2}\left(\frac{1}{n}\right)^2\left(1 - \frac{1}{n}\right) + \left(\frac{1}{n}\right)^3\right]\right)^m \quad (3)$$

when there are $m$ tiers and $n$ nodes per tier. If routes are unknown to the attacker then *P(s)* for one route is $1/n$ and for three routes is

$$P(s) = \binom{3}{2}\left(\frac{1}{n}\right)^2\left(1 - \frac{1}{n}\right) + \left(\frac{1}{n}\right)^3 \quad (4)$$

We plot these probabilities for various values of the parameters in Figure 19.

In digital circuits we can implement the routing model by finding a disjoint decomposition of the circuit, creating diverse versions of each of the resulting components, and then choosing a subset of these to implement the function. This implementation is illustrated in Figure 20.

To evaluate the area and performance overhead of this implementation we applied it to a collection of ISCAS benchmarks in an Altera Cyclone IV EP4CE55F23C6 FPGA. We used the same set of benchmarks as in Section 3.1. We measure area by the number of logic elements required to implement the circuit and performance overhead

by the change in maximum operating frequency ($f_{max}$). To implement the routing model we first attempt to partition the benchmark circuit into 1024 disjoint components. To do this we first choose a random node in the benchmark and then trace its transitive fanout cone for 5 levels of logic. If the fanout at any point the transitive fanout cone exceeds 200 gates then we randomly choose a new initial node. We want to study the overhead

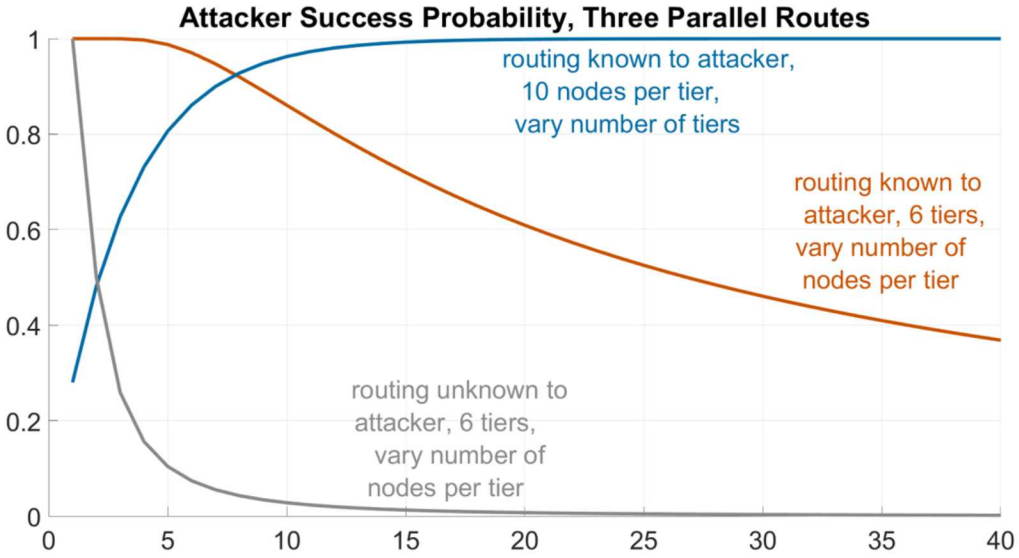


**Figure 19.** Attacker probability of success in the routing model for various numbers of tiers and nodes per tier, and with routing either known or unknown to the attacker.

as a function of the percentage of circuit components for which redundant implementations are produced. To do this we generate diversely redundant implementations of 0-100% of the components in steps of 10%. The diversely redundant implementations are generated by randomly selecting a component with fewer than 8 primary inputs and then generating four diversified implementations of it. Each of these four implementations is created by randomly choosing amongst the gate replacement, gate addition, and dynamic output inversion techniques. We then introduce one 4:1 MUX for each primary output from the component so that we can dynamically choose which implementation to employ.

In Figure 21 and Figure 22 we present the area overhead, as measured by the number of logic elements required to implement the circuit in an Altera Cyclone IV EP4CE55F23C6 FPGA, and performance overhead, as measured by the change in

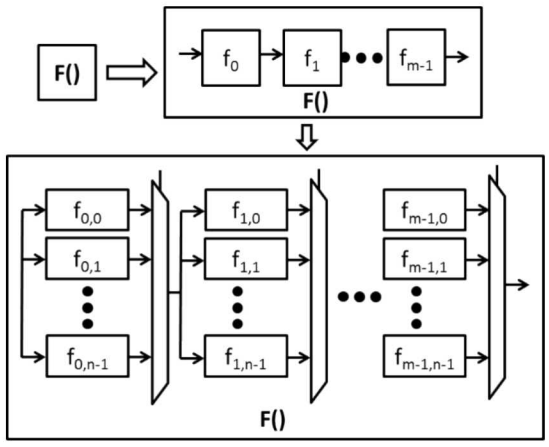


**Figure 20**. One realization of the routing model in hardware

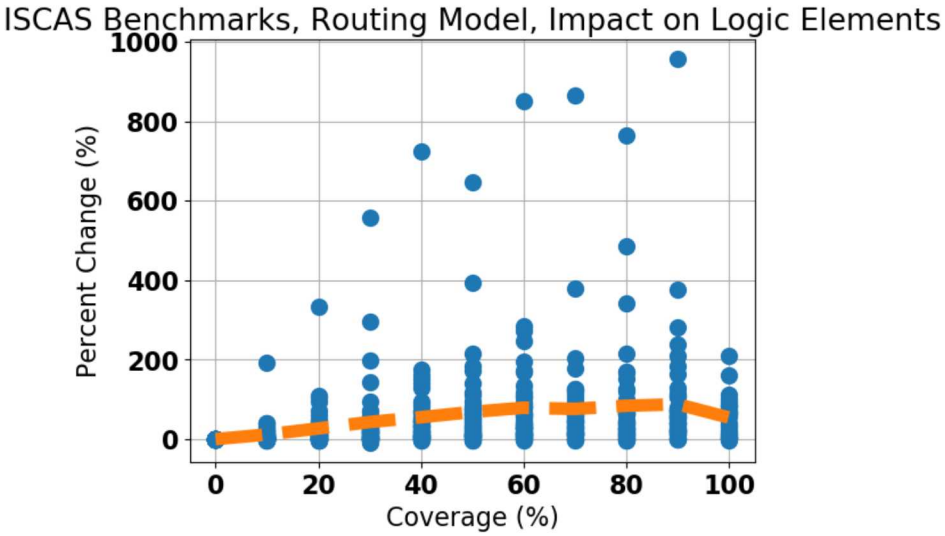


**Figure 21.** Area overhead for the routing model structure as measured by increase in the number of logic elements required to implement the circuit in a Cyclone IV EP4CE55F23C6 FPGA.

maximum operating frequency ($f_{max}$) when targeting the same FPGA, after applying the routing model to the same set of ISCAS benchmarks as used in Section 3.1. As expected, we see that the area overhead tends to increase with coverage, with the average area overhead reaching about 100% at 100% coverage. There is wide variance in the results, with most circuits showing overheads between 0 and 200% for coverage ranging from 30-100%. A few circuits have overheads in excess of 300% for overheads of 20% or more. We also find that $f_{max}$ decreases with increasing coverage, although the loss in performance tends to level out after coverage exceeds 50%. The average reduction in $f_{max}$ is about 25% when coverage exceeds 50%. We observe wide variation in the results for all coverage values, with some circuits showing an increase in $f_{max}$ even at 100% coverage and others showing reductions of 40% or more at only 10% coverage.

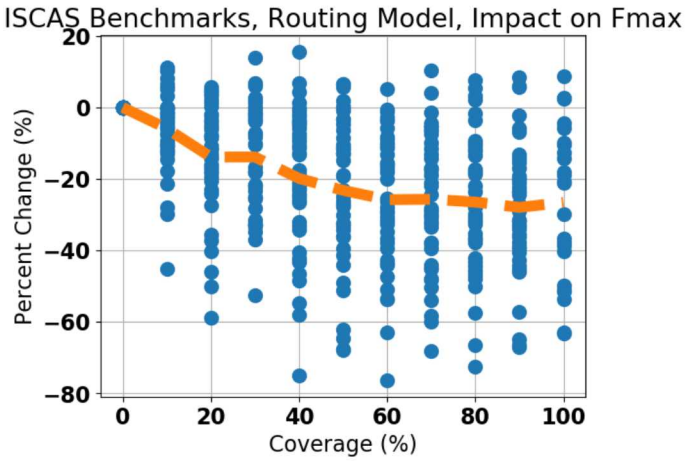


**Figure 22.** Performance overhead for the routing model structure as measured by increase in the number of logic elements required to implement the circuit in a Cyclone IV EP4CE55F23C6 FPGA.

### *4.1.3. Effect of Diverse Voting*

We consider a general template that places three random implementations in a voting system whose output is the majority of the individual outputs. If there are two or more incorrect outputs and there is no majority, we treat this as an incorrect behavior. This represents a security analogue of the reliability technique of triple modular redundancy, and follows a well-known formula: if the probability of incorrect output from any one implementation is $q$, then the probability of incorrect output from the voting system is $3q^2 - 2q^3$.

#### 4.1.3.1. Voter of Compositions

For a "voter of compositions", we use three random implementations of the entire component-based system. Thus, we take $q$ as the probability from Eq. (1), and the new probability of compromise is

$$p = 3(1 - (1 - 2^{-k})^m)^2 - 2(1 - (1 - 2^{-k})^m)^3 \quad (5)$$

We have constructed a concrete formal model of a voting system of this kind, and computed the probability of compromise directly using the probabilistic model checker PRISM [9]. The result $p = 0.00111$ for $k = 8$ and $m = 5$ agrees numerically with Eq. (5). We plot Eq. (5) for various key lengths $k$ and number of components $m$ in Figure 23.

A potential hardware implementation of the voter of compositions is presented in Figure 24. We begin with a circuit that implements some function $F()$ and decompose it into $m$ disjoint components $f_0$, $f_1$ ... $f_{m-1}$. We can then generate diverse implementations of each of these $m$ components to build diverse implementations of the desired function $F()$. In Figure 24 we show three such implementations of $F()$. These implementations are combined with a majority voter to implement the diversely redundant voter of compositions architecture. In this implementation each of the three variants of $F()$

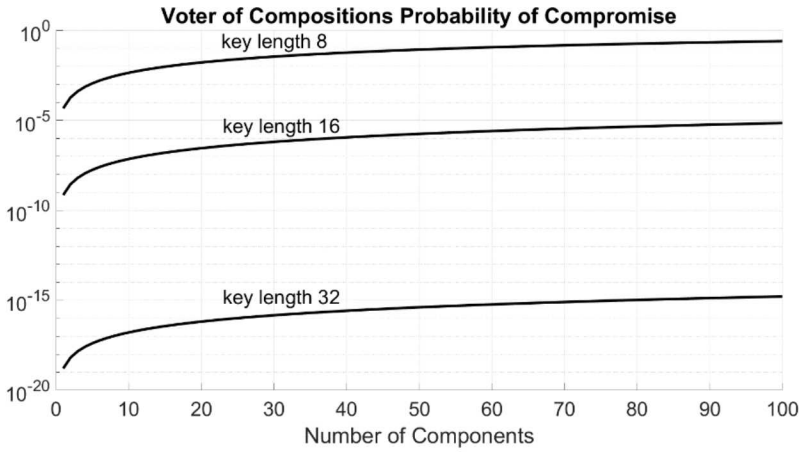


**Figure 23.** Probability of compromise for a voter of compositions system with varying numbers of components and for different key lengths.

utilizes the same decomposition into components. An alternative architecture is to allow the component-wise decomposition to vary between the implementations. This alternative architecture permits more diversity between the components and individual implementations of *F()* by allowing the component boundaries to vary.

In Figure 25 and Figure 26 we present the area overhead, as measured by the number of logic elements required to implement the circuit in an Altera Cyclone IV EP4CE55F23C6 FPGA, and performance overhead, as measured by the change in maximum operating frequency ($f_{max}$) when targeting the same FPGA, for applying the voter of compositions to the same set of ISCAS benchmarks as used in Section 3.1. For these results, we consider the alternative implementation for which the component definitions vary between the three implementations of *F()*. We evaluate the overhead as a function of the percentage of the circuit "covered" by the diversification, where coverage is measured as the fraction of components $f_0, f_1 \ldots f_{m-1}$ for which we produce diverse implementations. We consider coverage in steps of 10% from 0% to 100% of the components. To generate the diverse implementations we first attempt to partition the circuit into 1024 disjoint components. To accomplish this, we choose a random node within the circuit and trace its transitive fanout for five levels of logic. If at any point the number of gates in the transitive fanout cone exceeds 200 gates then we select a new initial node. Then, for 0-100% of the components, in steps of 10%, we randomly choose a component with fewer than 8 primary inputs. Next we diversify the implementation of this component with randomly choosing amongst gate replacement, gate addition, and dynamic output inversion. We repeat this process for the desired fraction of components from the decomposition, and then proceed to the next copy of *F()*. The outputs of the three variants of *F()* are combined by introducing one majority-3 voter for each primary output of *F()*.

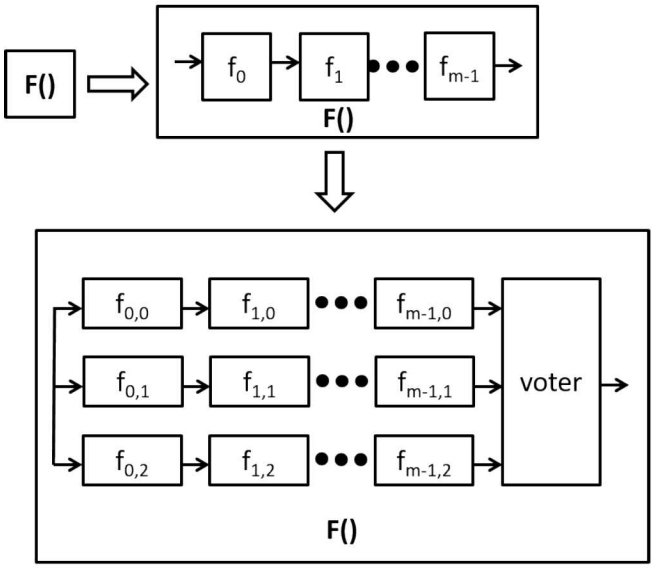


**Figure 24.** Hardware architecture for a voter of compositions

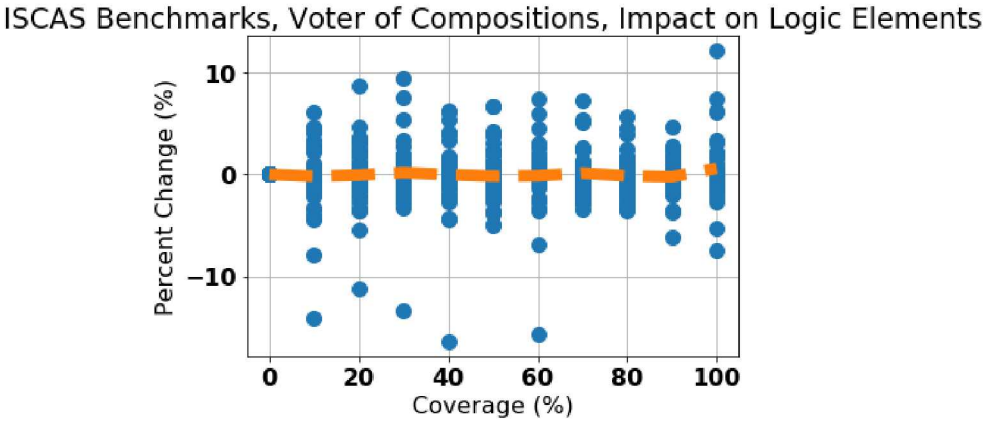


**Figure 25.** Area overhead for the "voter of compositions" structure as measured by increase in the number of logic elements required to implement the circuit in a Cyclone IV EP4CE55F23C6 FPGA.

For these results we constrain the synthesis tool to preserve all of the latches in the composition structure to prevent it from collapsing the three implementations of *F()* into a single implementation. However, this constraint does not prevent the synthesis tool from finding, and optimizing, redundancies in the diverse implementations of individual components. Due to these optimizations we find that, on average, the voter of compositions structure does not cause the number of logic elements to increase. However, there is wide variance between benchmarks, with overheads ranging from about -20% to about 12%.

The impact of the voter of compositions structure on fmax is also in dependent on coverage, with there being no change on average. As with the area overhead, there is considerable variance in fmax, with most circuits having anywhere from a 20% reduction in fmax to a 20% increase. This suggests that diversifying individual

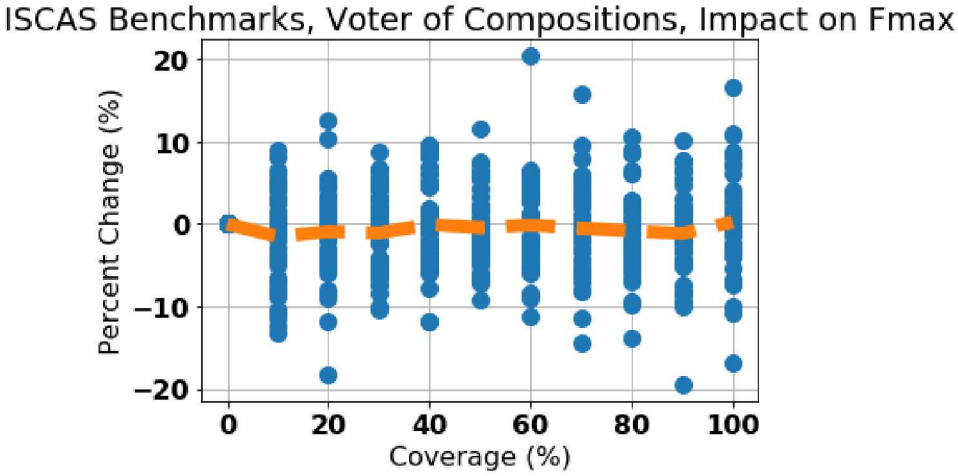


**Figure 26.** Operating frequency overhead for the "voter of compositions" structure as measured by decrease in the maximum operating frequency of the circuit in a Cyclone IV EP4CE55F23C6 FPGA

components does not appreciably impact critical paths within the circuits or overly burden placement and routing for these benchmarks.

#### 4.1.3.2. Composition of Voters

For a "composition of voters", we instead create a system in which each component is individually diversified by voting three variants. More generally, a "composition with some voters" is a system in which some components may be threefold voters while others may be single random variants as before. Let $d$ be the fraction of components that are diversified as voters. Then $dm$ components are diversified and $(1 - d)m$ are not. Applying the same basic voting formula, the probability of incorrect behavior for a diversified component is $3f^2 - 2f^3 = 3x2^{-2k} – 2x2^{-3k}$. Thus, the probability that at least one component produces incorrect behavior is

$$p = 1 - (1 - 3 \times 2^{-2k} + 2 \times 2^{-3k})^{dm}(1 – 2^{-k})^{(1-d)m} \quad (6)$$

The probability computed from a corresponding PRISM model agrees numerically with Eq. (6) for $k = 8$, m = 5, and $d \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$. When $d = 0$, Eq. (6) reduces to Eq. (1) as it should. We plot Eq. (6) for various values of $k$, $m$, and $d$ in Figure 27. Particularly when $k$ is large, we see a strong benefit from diversifying all components, but much less benefit when even one component is left undiversified, as it becomes the weakest link.

We can even consider a higher-overhead "voter of compositions of voters" or "voter of compositions with some voters", in which the system with individually diversified components is itself instantiated randomly three times in a higher-level voting system. The probability of compromise is $3p^2 - 2p^3$ where $p$ is given by Eq. (6).

A hardware architecture for implementing a composition of voters is shown in Figure 28. As with the voter of compositions, we first decompose function $F()$ into $m$ disjoint components $f_0, f_1 \ldots f_{m-1}$. For this decomposition we attempt to partition the circuit into 1024 components by iteratively selecting a random node within the design and tracing

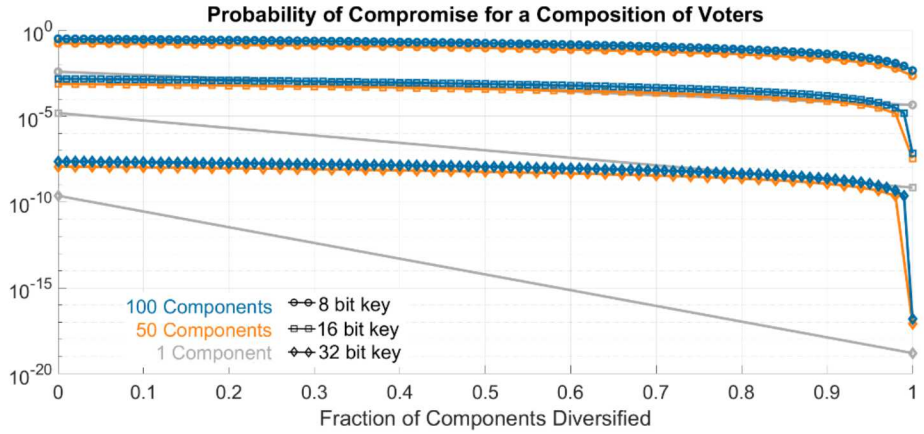


**Figure 27.** Probability of compromise in the composition of voters structure for various numbers of components, key lengths, and fractions of diversified components

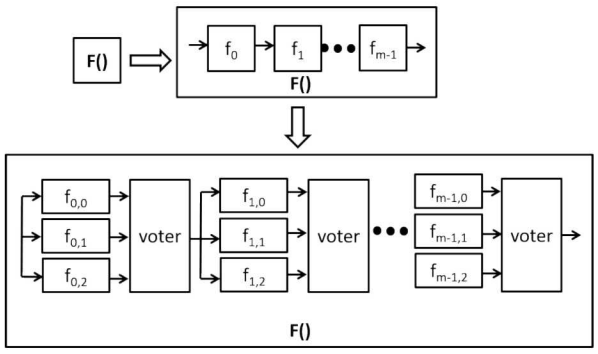


**Figure 28.** Hardware architecture for a "composition of voters"

its transitive fanout cone for five levels of logic. If the resulting component has fewer than 200 gates then it becomes a component in our decomposition, otherwise we randomly select a new starting node and repeat the process. Recall that in the voter of compositions we could choose either to use the same decomposition for each redundant copy of *F()*, or we could use different decompositions for the various copies. In the composition of voters structure we use the same decomposition for each of the redundant copies of the circuit so that we can vote on the outputs of the redundant copies of the components.

We again consider covering from 0-100% of the components by randomly selecting a fraction of the components from the decomposition $f_0, f_1 \ldots f_{m-1}$ to diversify and vote. For the diversification, for 0-100% of the components in steps of 10% we first choose a random component with fewer than 8 inputs to diversify. Then, we generate three diverse implementations of this component by randomly choosing between gate replacement, gate addition, and dynamic output inversion for each of the diverse implementations. The outputs from these three variants are combined with a majority voter by introducing one majority-3 voter for each primary output of the component.

We found the area and performance overheads for a collection of the ISCAS benchmarks as a function of the fraction of diversified components. For these experiments we used the same ISCAS benchmarks as in Section 3.1. Results from these experiments appear in Figure 29 and Figure 30. As before, we constrain the synthesis tool to preserve all of the latches within the design to ensure that the diverse implementations of the circuit are not collapsed into a single implementation. We find that the area overhead from the composition of voters is on average more than that from the voter of compositions, with an average overhead of only 65% even at 100% coverage. This likely results from the necessity of using the same circuit decomposition for each of the diverse implementations providing greater opportunities for the synthesis tool to identify and optimize redundancies within the design. We still observe a large variance in the area overhead, with overheads typically ranging from 0-200% for this set of benchmarks. For one of the benchmarks the overhead exceeds 800% for

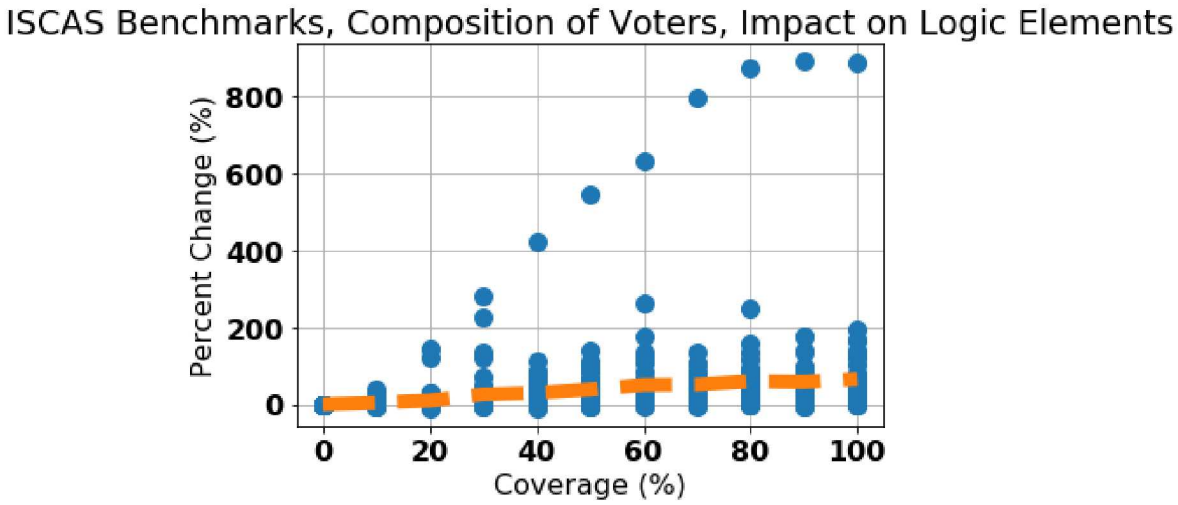


**Figure 29**. Area overhead for the " composition of voters" structure as measured by increase in the number of logic elements required to implement the circuit in a Cyclone IV EP4CE55F23C6 FPGA.

component coverages of 70-100%. We find that the area overhead increases steadily with increasing coverage.

The performance overhead from the composition of voters structure also tends to increase with coverage. Average reductions in fmax are about 0-5% for coverage from 0-20%, and about 10-20% for coverage from 20-100%. The variance in results is again large, with fmax increases of almost 20% observed for some circuits, and decreases of 60-70% for others.

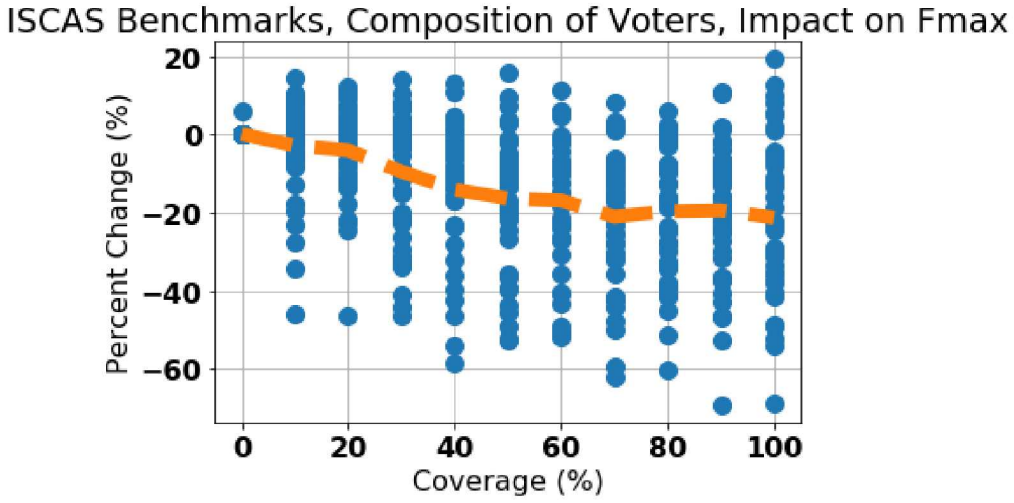


**Figure 30.** Operating frequency overhead for the "composition of voters" structure as measured by decrease in the maximum operating frequency of the circuit in a Cyclone IV EP4CE55F23C6 FPGA

### 4.1.4. *Effect of Moving Target*

All the results above apply to the probability of success for an attacker trying a single input. If the attacker can make multiple tries, the probability of success increases. In this case, if the component implementations are random but static, then once a system vulnerability is found, it can be exploited repeatedly with certainty. On the other hand, if the component implementations are re-randomized between attacker tries – a "moving target" approach - then each attempt to compromise the system is independent of previous attempts and equally difficult. This is particularly helpful to security if the attacker's goals require compromising the system more than once, e.g., for exploration, testing, and deployment stages, and returning to the system repeatedly over time to gain further benefits from the exploit.

In the moving target approach, if the attacker tries $N$ inputs, each with independent probability $p$ of compromising the system, then the number of successes follows the binomial distribution $B(N, p)$. Because in a well-designed system $p$ is very small, and correspondingly the attacker likely needs a large number $N$ of tries, the binomial distribution can be approximated by a Poisson distribution with the parameter $\lambda = Np$ (the mean number of successes) as $N \rightarrow \infty$ and $p \rightarrow 0$. The probability of at least $\xi$ successes according to the Poisson distribution is

$$P_{\geq}(\xi) = 1 - \sum_{i=0}^{\xi-1} \frac{\lambda^i}{i! e^{\lambda}} = 1 - Q(\xi,\lambda) \tag{7}$$

where $Q$ is the regularized gamma function [10]. If $\xi \gg \lambda$ , then this probability is small and is dominated by exactly $\xi$ successes:

$$P_{\geq}(\xi) \approx \frac{\lambda^{\xi}}{\xi! e^{\lambda}} \tag{8}$$

It has been noted that moving target does not have a strong effect on the difficulty of compromising the system *once*, compared to static random implementations [11]. For example, if the probability $p = 1/M$ is due to a single vulnerability in a large input space of size $M$, then in the static case, an exhaustive search will find the vulnerability with

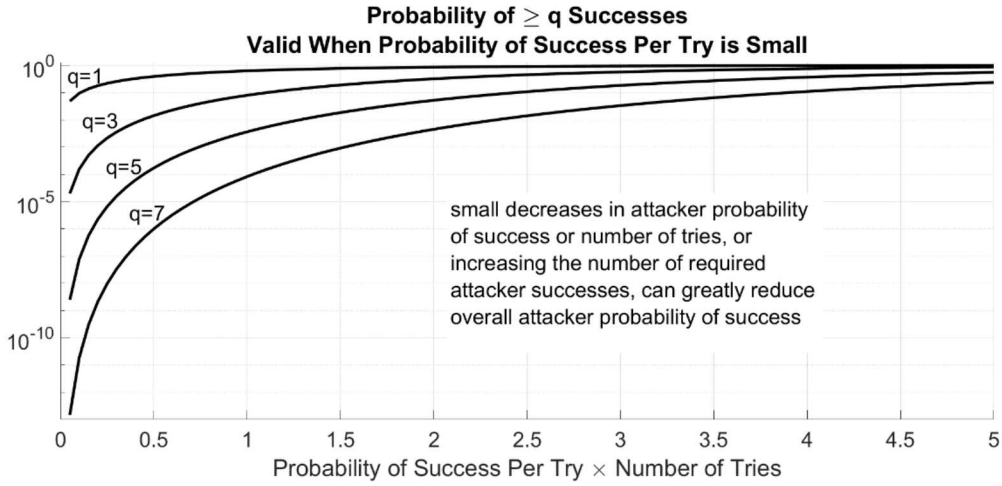


**Figure 31.** Plots of Eq. (7)

certainty within $N = M$ tries. The moving target result (Eq. 7) for the same number of tries ($\lambda = Np = 1$) indicates that the probability of at least one success ($\xi = 1$) is $1 - 1/e \approx 0.63$, which is still high. The advantage of moving target becomes apparent when requiring multiple successes ($\xi = 2, 3, \ldots$), since in the static case subsequent compromises are immediate once the first is achieved, whereas the moving target probability (Eq. 7) decreases rapidly with $\xi$, especially when $\lambda \leq 1$. We plot Eq. (7) in Figure 31, where the horizontal axis represents the parameter $\lambda$.

We introduce dynamic, moving target aspects to our "voter of compositions" and "composition of voters" structures by creating additional diverse implementations of the circuit components and then using multiplexors to select specific implementations to use during circuit operation. Although we do not specify how the select inputs to these multiplexors are generated, there are many possibilities. The select inputs can be chosen, and varied, individually or in groups. They can be defined once when the circuit is fabricated, or its bitstream generated, or they can be changed over time. For example, they can be updated each time the circuit powers on or on some schedule during circuit operation. If they are varied during circuit operation then it may be necessary to define the components along latch boundaries to ensure proper operation of the circuit. The select inputs can be provided by a user, or calculated as some function of the primary inputs and internal signals of the circuit.

First, we discuss the dynamic voter of compositions structure depicted in Figure 32. As with the regular voter of compositions, we can either use the same decomposition of function $F()$ for each of the distinct implementations of $F()$, as shown in Figure 32, or we can choose different decompositions for different variants. In either case, for each variant of $F()$ we first decompose the circuit into $m$ disjoint components $f_0, f_1 \ldots f_{m-1}$. For each copy of $F()$ we then generate a diverse collection of variants of each of the $m$ components. The outputs of these variants are input to a multiplexor that chooses which of them to deploy. In Figure 32 we illustrate $n$ variants for each of the components, although it is not necessary that the same number of variants be generated for each of the components, or that the same number of variants be used from one implementation of $F()$ to another. In either case, we use this approach to

generate a several distinct implementations of function *F(),* and the outputs of these implementations are combined with a majority voter.

In these experiments we use the same collection of ISCAS benchmarks as in Section 3.1. For the results shown in Figure 33 and Figure 34 we use different decompositions of *F()* for each of the three variants. We consider only 4:1 multiplexors in these experiments. For each of the three copies of the circuit we first attempt to partition *F()*

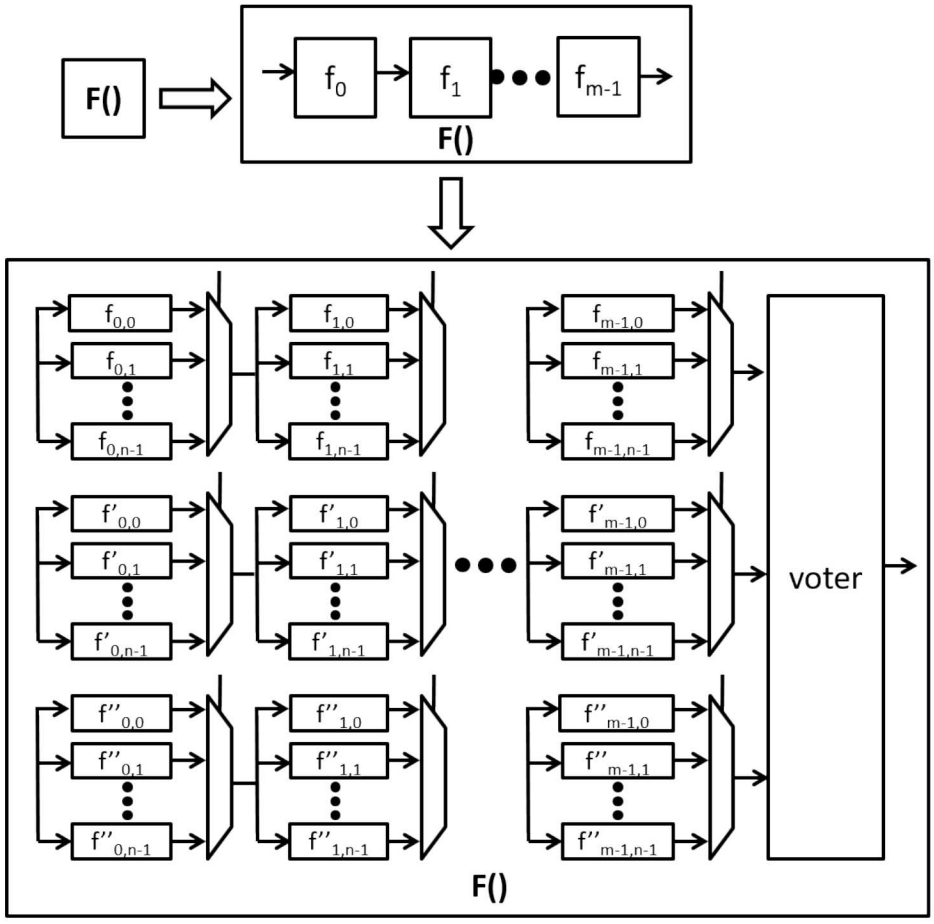


**Figure 32.** Hardware architecture for a dynamic "voter of compositions"

into 1024 disjoint components. To do this, we choose a random node in the design and trace its transitive fanout for five levels of logic. If at any point the fanout exceeds 200 gates, then we reject this node and randomly select a new initial node for the component. As before, we consider the impact of applying the dynamic voter of compositions architecture to between 0% and 100% of the resulting components in steps of 10%. To generate the components variants themselves we randomly choose a component with fewer than eight primary inputs. We then generate four diversified implementations of that component. The diverse variants are created by randomly selecting amongst the gate replacement, gate addition, and dynamic output inversion diversification approaches. We then add a 4:1 MUX to select between these four

diverse variants. This process is repeated until we have diversified the desired fraction of the components. We then repeat this process to generate the remaining two variants of *F()*, and finally combine the results of the three variants with a majority voter. For this, we create one majority-3 voter for each primary output of *F()*.

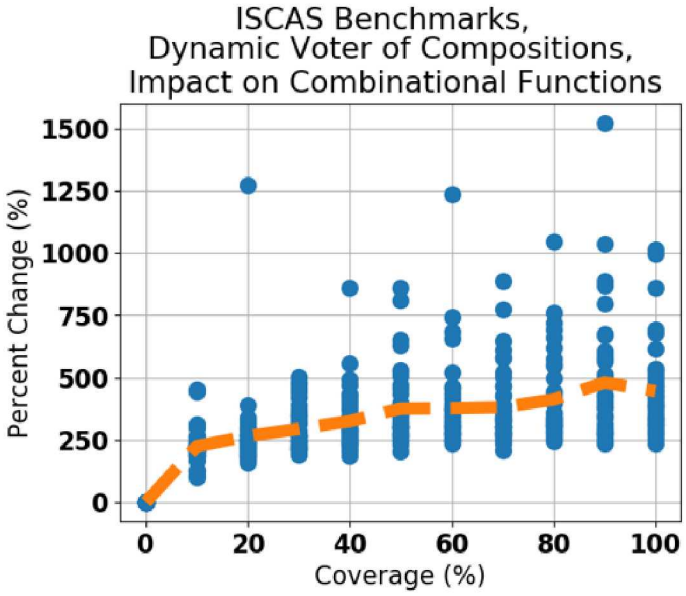


**Figure 33.** Area overhead for the dynamic "voter of compositions" structure as measured by increase in the number of logic elements required to implement the circuit in a Cyclone IV EP4CE55F23C6 FPGA.

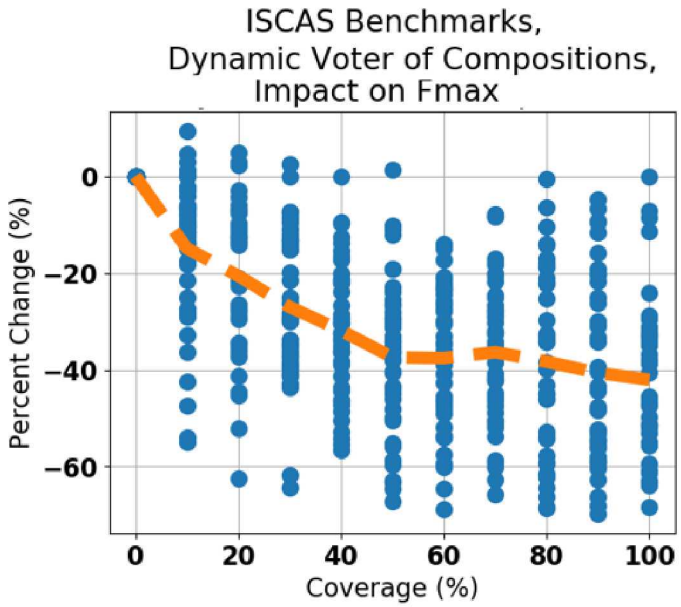


**Figure 34.** Operating frequency overhead for the dynamic "voter of compositions" structure as measured by as measured by decrease in the maximum operating frequency of the circuit in a Cyclone IV EP4CE55F23C6 FPGA

The area overhead for the dynamic voter of compositions architectures is presented in Figure 33. We see that the area overhead increases steadily with circuit coverage. The overhead in this architecture is also much larger than in the static case, increasing to almost 100% at 80% coverage. There is considerable variance across benchmarks, with overheads typically ranging from 0-200% at 100% coverage. At 100% coverage we expect approximately 1200% increase in circuit area (four redundant implementations of each component in each of 3 redundant implementations of the circuit), so these results are less than expected. We attribute this reduction to optimizations performed by the synthesis tool, and to an inability to generate redundant copies of all of the components due to our restriction that transitive fanout cones have fewer than 200 gates. This restriction can prevent our tool from achieving 100% coverage.

In Figure 34 we see that reductions in fmax also increase more consistently with increasing circuit coverage than in the static case. Here, we find that fmax initially decreases steadily, reaching nearly a 40% reduction at 50% coverage. After this we see relatively modest decreases in fmax, with the average hovering around 40% for coverage from 50-100%. There is wide variation across circuits, with fmax reductions ranging from 0-60% across all coverage levels, and extending to 0-80% when coverage exceeds 50%.

Now, we discuss the dynamic composition of voters structure shown in Figure 35. As with the static composition of voters, we begin by attempting to partition *F()* into disjoint components $f_0$, $f_1$ ... $f_{m-1}$. We generate several distinct implementations of each of these *m* components. Then, the output from a subset of these implementations

is selected to be input to a majority voter. While Figure 35 illustrates *n* diverse implementations of each of the components, in practice the number of distinct implementations can vary from one component to the next.

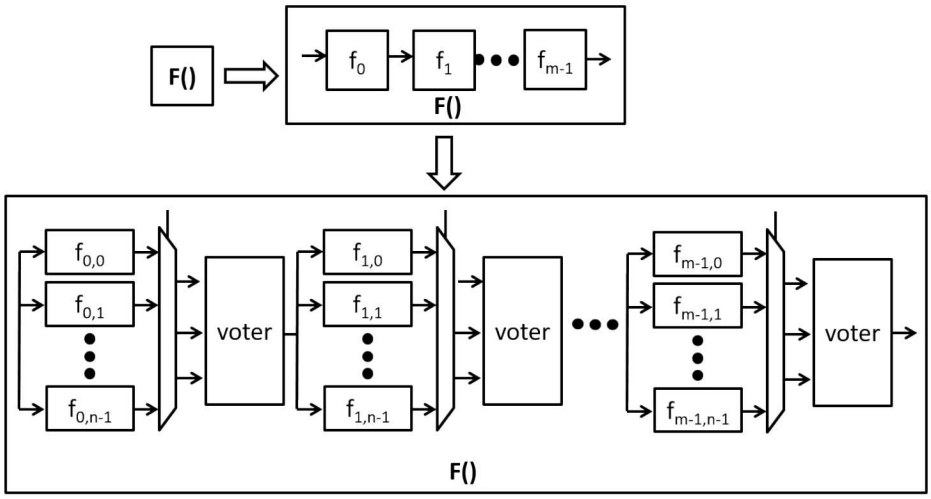


**Figure 35.** Hardware architecture for a dynamic "composition of voters"

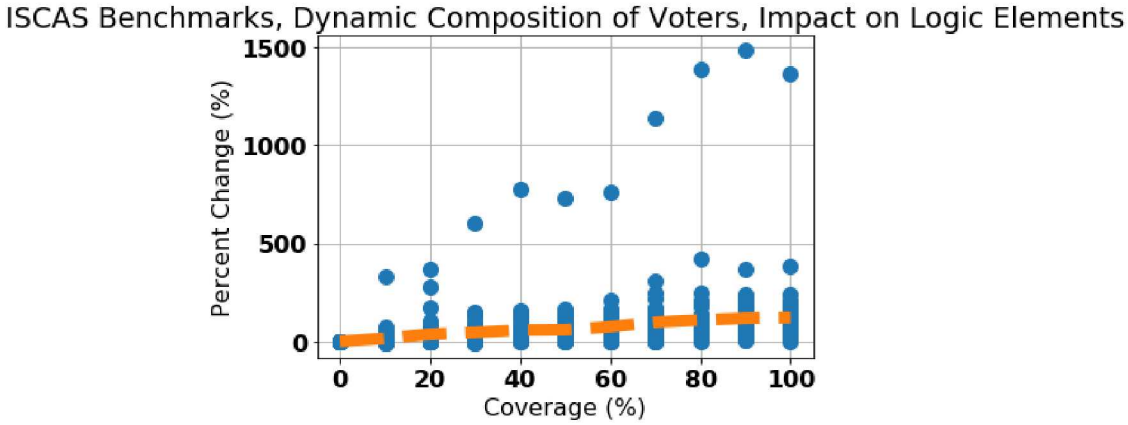


**Figure 36.** Area overhead for the dynamic "composition of voters" structure as measured by increase in the number of logic elements required to implement the circuit in a Cyclone IV EP4CE55F23C6 FPGA.

For our experiments we first attempt to partition the circuit into 1024 components. For this we randomly select a node in the circuit and trace its transitive fanout for 5 levels of logic. If at any time we have more than 200 gates in the fanout then we reject this initial node and select a new one to be the root of the logic cone. As with the other architectures, we consider the impact of applying the dynamic composition of voters architecture to between 0% and 100% of the resulting components, incrementing in steps of 10%. To generate the component variants themselves we randomly choose a component with fewer than eight primary inputs and generate four diversified implementations of that component. The diverse variants are created by randomly

selecting amongst the gate replacement, gate addition, and dynamic output inversion diversification approaches. We then introduce three 4:1 MUXs and one majority-3 voter for each output bit of the logic cone. This MUX structure allows any combination of the four implementations of the component, including repetitions of a single implementation, to be input to the majority voter.

We found the area and performance overheads for the same collection of the ISCAS benchmarks used in Section 3.1 as a function of the fraction of diversified components. Results from these experiments appear in Figure 36 and Figure 37. As in the previous implementations, we constrain the synthesis tool to preserve all of the latches within the design to ensure that optimization does not collapse the diverse implementations of the components into a single implementation. We find that the area overhead from the dynamic composition of voters is on average more than that from the static composition of voters, which is to be expected. The average overhead is about 120% at 100% coverage, with variation from 0-500% across circuits. This is less than we would expect, and likely results from our requirement to use the same circuit decomposition for each of the diverse implementations, which provides the synthesis tool with opportunities to identify and optimize redundancies in the design. While the overhead increases steadily with coverage, we see large variance across benchmarks at all coverage levels. Area overhead typically varies from 0% to about 200% for 10-50% coverage, with maximum overhead increasing to about 500% for coverage ranging from 60-100%.

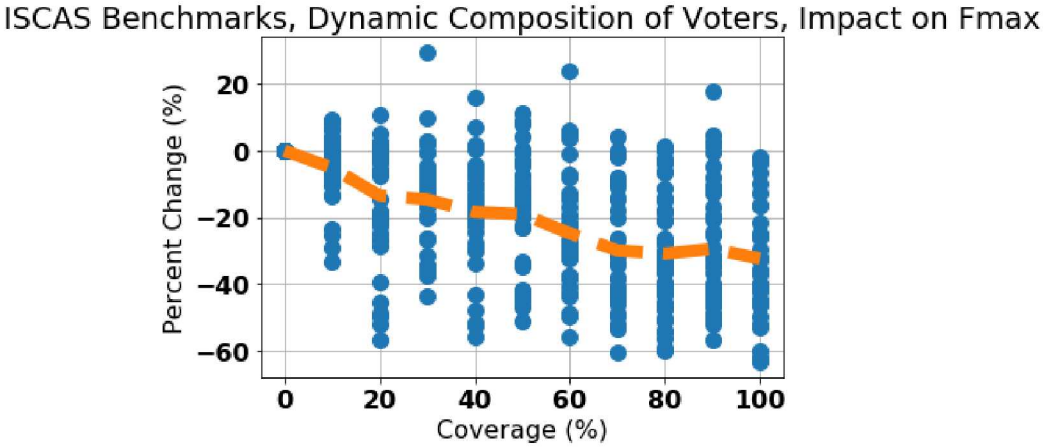


**Figure 37.** Operating frequency overhead for the dynamic "composition of voters" structure as measured by as measured by decrease in the maximum operating frequency of the circuit in a Cyclone IV EP4CE55F23C6 FPGA

The timing overhead from the composition of voters structure also increases with coverage. Average reductions in $f_{max}$ are always less than 40%, but with wide variance. At all coverage levels we find that some circuits have reductions in $f_{max}$ approaching 60%, while others have increased $f_{max}$. No circuit experiences more than a 70% reduction in $f_{max}$ at any level of coverage.

Both area and timing overheads are larger in the dynamic architecture than in the static architecture, which is expected due to increasing the number of diverse

implementations from three to four and from the incorporation of multiplexors in the dynamic composition of voters.

## 4.2. Hardware Considerations

The models we have presented assume that the diverse implementations of the circuit components have independent vulnerabilities. There are several ways that this independence may be achieved in practice:

1. If each of the diversified components is restricted to preserve the intended functional behavior of the component, then vulnerabilities that exist at a semantic level other than the functional level can be impacted by mitigations that preserve the input-to-output behavior of the components but that diversify the internal behavior. Examples of such vulnerabilities are trojans that exist in side channel analog behavior.

2. The logical behavior of the components can be changed, subject to the constraint that the modifications do not prevent the circuit from performing its intended function.

When combined with the hardware architectures presented in this section, the diversification methods described in Section 3 are sufficient to address item 1. Other simple modifications to circuit implementations could also be employed. For example, a chain consisting of an even number of inverters can be replaced by a wire to change the timing and power consumption characteristics of a circuit without modifying its logical behavior.

For the second approach, one possibility for achieving these modifications is to implement the components as collections of approximate circuits [15]. In this approach a component's functionality is implemented by three (or more) circuits, each of which approximates the intended behavior of the component, and for which the majority vote of the approximate circuit implements the intended behavior. In such an implementation each of the approximate circuits implements different functions, and so their vulnerabilities may also be different. Generating several distinct approximate circuit based implementations of a component is also possible, allowing the approach to be used in the "voter of compositions" architectures. A second possibility is to form components consisting only or mostly of trojan nets, dangling nodes, or other circuit structures that are unnecessary for proper functioning of the circuit as measured by a comprehensive suite of simulations. Since these structures do not impact the intended functionality of the circuit they can be modified freely without impacting the desired functionality. However, under the assumption that trojan nets are only a small fraction of the all of the nets in the design, there will be very few components that can be diversified in this way.

Finally, we reiterate that the circuit variants considered in this section have the same logical behavior as the original circuit from the primary inputs to the primary outputs, but differ in the way that this behavior is implemented. This implies that the variants also have differing analog characteristics, such as timing and power consumption, and that their fault behavior will also be distinct. This provides the possibility of changing

vulnerabilities or trojan behavior related to the specific implementation of the circuit, although it cannot impact the logical function of a trojan at the output of the circuit since logical behavior is preserved.

## 4.3. Conclusions and Recommendations

In Table 3 we compare the average area and $f_{max}$ costs for implementing the various diversity architectures, and in Figure 38 we show the attacker's expected probability of success as a function of the number of components in the design over a range of parameters for the various architectures.

**Table 3.** Average area and performance overheads for various diversity architectures at 100% coverage

| | % change area (average) | % change $f_{max}$ (average) |
|---|---|---|
| **routing model** | 100% | -25% |
| **static voter of compositions** | 0% | 0% |
| **static composition of voters** | 65% | -20% |
| **dynamic voter of compositions** | 100% | -40% |
| **dynamic composition of voters** | 120% | -30% |

The routing model generally offers less protection than the other architectures, but at comparable cost to the static voter of compositions and static composition of voters.

Comparing the static voter of compositions and static composition of voters, we find that the composition of voters has on average much less overhead than the voter of compositions, however, we also note that the variances are large and that the average results from the voter of compositions are 0% because the results are distributed approximately equally between increases and decreases in both fmax and logic elements. Even so, the maximum overheads for the voter of compositions are less than the averages for the composition of voters. However, assuming a perfect composition in which the system functions correctly when each of the components functions correctly, the composition of voters provides more protection against attacks than the voter of compositions. Consequently, the choice between these static voting architectures will depend on whether low overheads or attack resilience are more desirable. Additionally, due to the symmetric variance in the voter of compositions, we recommend finding the overheads for a specific circuit prior to choosing between the architectures.

When we consider the dynamic moving target architectures we find that the dynamic voter of compositions has somewhat less area overhead than the dynamic composition of voters, but somewhat more fmax overhead. Since the attack resilience provided by the dynamic architectures is related to the protection offered by the static architectures, as described in Section 4.1.4, and the static composition of voters usually provides

better attack resilience than the static voter of compositions, we find that the composition of voters will usually be the preferred dynamic architecture.

Recall from Section 4.1.4 that the most dramatic decreases in attacker probability of success from moving target architectures result from forcing the attacker to successfully attack the system multiple times, with a more modest 37% reduction in attacker probability of success when only a single successful attack is required. If we cannot force the attacker to require multiple successes then the increased costs of the dynamic voter of compositions over the static voter of compositions will usually not be worthwhile. As such, we usually prefer the static voter of compositions to the dynamic voter of compositions. On the other hand, if the performance reduction of the dynamic composition of voters is acceptable, then the 37% reduction in attacker probability of success when compared to the static composition of voters is appealing. If the attacker can be forced into requiring multiple successful attacks, the dynamic composition of voters provides the strongest protections for about twice the area overhead and only modestly larger fmax overhead than the static composition of voters. For these reasons, we recommend the dynamic composition of voters if the impact on area is acceptable, and the static composition of voters when the area impact from the dynamic composition of voters is unacceptable. When little area or fmax overhead is tolerable, then the static voter of compositions is preferred.

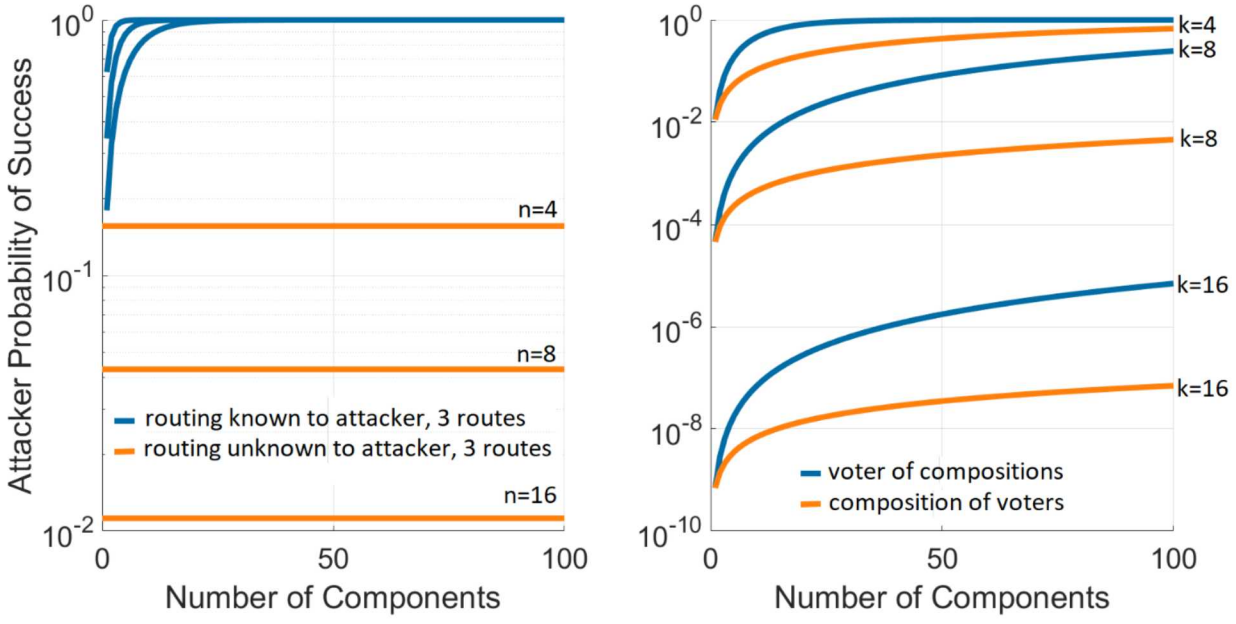


**Figure 38**. Attacker probability of success as a function of the number of components in various diversity architectures

This page left blank

# 5. HEALING VOTERS

In this section we analyze different majority voting structures and propose a new majority voter structure, which we call a *healing voter*, that has better characteristics than standard voter structures under many circumstances. There are several ways to build a majority voter that determines what the output from a collection of diverse variants should be. Consider the case of three diverse variants. If two or more of these variants produce the same output, then the majority should be selected as the output. More interesting is the case in which the outputs from the three variants are all different, which implies that two or more of them are incorrect. The simple majority voter, which compares the full output vectors from the variants, can detect this scenario and choose to fail safe, rather than outputting an incorrect value, because there is low confidence in the output. The bitwise majority voter, which compares the output vectors bit by bit, cannot detect this scenario. However, it always produces an output, and as long as the output variants have errors in nonoverlapping bits will produce the correct output even when all of the variants differ. The simple majority voter prefers integrity of the output, while the bitwise majority voter favors availability. There are advantages to each of these. In particular, a failsafe output would be preferred over the bitwise majority voters output when there are overlapping errors in the variant's output, but the simple majority voter may be too conservative and choose the failsafe option too often. We propose a third option, the healing voter, which using the bitwise majority output if it exactly matches the output of any of the variants, and otherwise fails safe. This voter produces the correct output whenever two of the variants have nonoverlapping errors, and fails safe most of the time when there are overlapping errors. In this section we provide an analysis of these voting approaches, and then validate our analysis by comparing the performance of the various voter structures on the empirical data generated in Section 6.2.

## 5.1. Analysis of Voting Systems Under an Error Model with Correlations Among Output Bits

### *5.1.1. Notation and Assumptions*

The number of output bits is $b$. For each variant, at least on average, the fraction of possible inputs that are "hard" (potentially yield errors) is $h$. We assume that for each possible input and a randomly chosen variant, with probability $1 - h$ there are definitely no errors in the variant's output, and otherwise, i.e., when the input is hard, each output bit may be incorrect with independent probability $r$. For $h < 1$, this introduces correlations among the output bits. The overall probability that a particular output bit of a variant is incorrect is $hr$. In a reliability scenario, $h$ can be interpreted as the probability of an intermittent event such as a particle strike within a circuit that has the potential to corrupt any of the output bits, thereby making it "hard" to produce the correct output. Given an input that is hard with respect to the variant(s) concerned:

- The probability that a variant's output is incorrect ($w$ for "wrong") is $w \equiv 1 - (1 - r)^b$.

- The probability that two variants' outputs both have a common incorrect bit (*d* for "doubly wrong") is $d \equiv 1 - (1 - r^2)^b$.
- The probability that two variants' outputs are different (*u* for "unequal") is $u \equiv 1 - (1 - 2r + 2r^2)^b$.
- The probability that one variant's output has an incorrect bit not present in a second variant's output (*a* for "additional error") is $a \equiv 1 - (1 - r + r^2)^b$.

### *5.1.2. Simple Majority Voter*

**Table 4. Simple Majority Voter Output Possibilities**

| Correct Variants | Description | Number of output possibilities | Voter result |
|---|---|---|---|
| 3 | All correct | 1 | Correct |
| 2 | Two correct | $3(2^b - 1)$ | Correct |
| 1 | Two incorrect, no majority | $3(2^b - 1)(2^b - 2)$ | Don't care / fail-safe |
| 1 | Two identical incorrect | $3(2^b - 1)$ | Incorrect |
| 0 | All incorrect, no majority | $(2^b - 1)(2^b - 2)(2^b - 3)$ | Don't care / fail-safe |
| 0 | All incorrect, two identical | $3(2^b - 1)(2^b - 2)$ | Incorrect |
| 0 | All identical incorrect | $2^b - 1$ | Incorrect |
| | Total | $2^{3b}$ | |

**Table 5. Simple Majority Voter Output Probabilities**

| Description | Probability |
|---|---|
| All correct | $(1 - hw)^3$ |
| Two correct | $3hw(1 - hw)^2$ |
| Two incorrect, no majority | $3h^2(1 - hw)(-2w + 2w^2 + u)$ |
| Two identical incorrect | $3h^2(1 - hw)(2w - w^2 - u)$ |
| All incorrect, no majority | $h^3(-2 + 3w(2 - 4w + 2w^2 + u) + 2(1 - 3r + 3r^2)^b)$ |
| All incorrect, two identical | $h^3(3 + 3w(-3 + 5w - 2w^2 - u) - 3(1 - 3r + 3r^2)^b)$ |
| All identical incorrect | $h^3(-(1 - w)^3 + (1 - 3r + 3r^2)^b)$ |
| Total | 1 |

**Table 6. Simple Majority Voter Results and Probabilities**

| Voter result | Probability |
|---|---|
| Correct | $1 - 3h^2w^2 + 2h^3w^3$ |
| Don't care / fail-safe | $3h^2(-2w + 2w^2 + u) + 2h^3(-1 + 3w - 3w^2 + (1 - 3r + 3r^2)^b)$ |
| Incorrect | $3h^2(2w - w^2 - u) + 2h^3((1 - w)^3 - (1 - 3r + 3r^2)^b)$ |
| Total | 1 |

When $h = 1$ (all bits are independent and have error probability $r$), the simple majority voter corresponds to a "voter of compositions" with $b$ components, each of which produces one of the output bits. The previously derived "probability of compromise" for such a system matches the probability of an other-than-correct result from above, $3w^2 - 2w^3$, with $r = 2^{-k}$ in terms of the previously used "key length" $k$.

### *5.1.3. Bitwise Majority Voter*

**Table 7. Bitwise Majority Voter Output Possibilities**

| Correct Variants | Description | Number of output possibilities | Voter result |
|---|---|---|---|
| 3 | All correct | 1 | Correct |
| 2 | Two correct | $3(2^b - 1)$ | Correct |
| 1 | Two incorrect, all bits okay | $3(3^b - 2^{b+1} + 1$ | Correct |
| 1 | Two incorrect, some bit bad | $3(2^{2b} - 3^b)$ | Incorrect |
| 0 | All incorrect, all bits okay | $2^{2b} - 3(3^b - 2^b) - 1$ | Correct |
| 0 | All incorrect, some bit bad | $2^{3b} - 2^{2b+2} + 3^{b+1}$ | Incorrect |
| | Total | $2^{3b}$ | |

When $h = 1$ (all bits are independent and have error probability $r$), the bitwise majority voter corresponds to a fully diversified "composition of voters" with $b$ components, each of which produces one of the output bits. The previously derived "probability of compromise" for such a system matches the probability of an other-than-correct result from above, $1 - (1 - 3r^2 + 2r^3)^b$, with $r = 2^{-k}$ in terms of the previously used "key length" $k$.

**Table 8. Bitwise Majority Voter Output Probabilities**

| Description | Probability |
|---|---|
| All correct | $(1 - hw)^3$ |
| Two correct | $3hw(1 - hw)^2$ |
| Two incorrect, all bits okay | $3h^2(1 - hw)(w^2 - d)$ |
| Two incorrect, some bit bad | $3h^2(1 - hw)d$ |
| All incorrect, all bits okay | $h^3(-1 + w^3 + 3(1 - w)d + (1 - 3r^2 + 2r^3)^b)$ |
| All incorrect, some bit bad | $h^3(1 - 3(1 - w)d - (1 - 3r^2 + 2r^3)^b)$ |
| Total | 1 |

**Table 9. Simple Majority Voter Results and Probabilities**

| Voter result | Probability |
|---|---|
| Correct | $1 – 3h^2(1 - h)d – h^3(1 – (1 – 3r^2 + 2r^3)^b)$ |
| Don't care / fail-safe | 0 |
| Incorrect | $3h^2(1 - h)d + h^3(1 – (1 – 3r^2 + 2r^3)^b)$ |
| Total | 1 |

### *5.1.4. Healing Voter*

Table 10 - Table 12 present results for the healing voter.

**Table 10. Healing Voter Output Possibilities**

| Correct Variants | Description | Number of output possibilities | Voter result |
|---|---|---|---|
| 3 | All correct | 1 | Correct |
| 2 | Two correct | $3(2^b - 1)$ | Correct |
| 1 | Two incorrect, all bits okay | $3(3^b – 2^{b+1} + 1)$ | Correct |
| 1 | Two incorrect, errors partly overlapping | $3(2^{2b} – 3(3^b – 2^b) - 1)$ | Don't care / fail-safe |
| 0-1 | At least two incorrect, errors conspiring | $(2^b - 1)(3(3^b - 2^b) + 1)$ | Incorrect |
| 0 | All incorrect, errors scattered | $(2^b - 3)(2^{2b} – 3(3^b – 2^b) - 1)$ | Don't care / fail-safe |
| | Total | $2^{3b}$ | |

**Table 11. Healing Voter Output Probabilities**

| Description | Probability |
|---|---|
| All correct | $(1 - hw)^3$ |
| Two correct | $3hw(1 - hw)^2$ |
| Two incorrect, all bits okay | $3h^2(1 - hw)(w^2 - d)$ |
| Two incorrect, errors partly overlapping | $3h^2(1 - hw)(d + 2a – w^2 – u)$ |
| At least two incorrect, errors conspiring | $3h^2(w^2 + u – 2a) + h^3(-1 – w^2(3 - w) + 3(1 – w)d + 3a + (1 – 3r + 3r^2)^b)$ |
| All incorrect, errors scattered | $h^3(1 + 3w(w – w^2 – u) – 3(1 – w)d – 3(1 – 2w)a – (1 – 3r + 3r^2)^b)$ |
| Total | 1 |

**Table 12. Healing Voter Results and Probabilities**

| Voter result | Probability |
|---|---|
| Correct | $1 – 3h^2(1 – hw)d – h^3w^3$ |
| Don't care / fail-safe | $3h^2(d + 2a – w^2 – u) + h^3(1 + 3w^2 – 3d – 3a – (1 – 3r + 3r^2)^b)$ |
| Incorrect | $3h^2(w^2 + u – 2a) + h^3(-1 – w^2(3 – w) + 3(1 – w)d + 3a + (1 – 3r + 3r^2)^b)$ |
| Total | 1 |

### *5.1.5. Tradeoffs*

First, let us assume that only the simple majority and bitwise majority voting systems are available, and consider the choice between them. Whenever the simple majority voter gives a correct result, so does the bitwise majority voter. This is because, if at least two of the three variant outputs are correct, then every bit has a majority correct. Whenever the simple majority voter gives an incorrect result, so does the bitwise majority voter. This is because, if at least two of the three variant outputs are identical and incorrect, then some bit has a majority incorrect. The effect of using the simple majority voter is that the fail-safe is output in some cases, where the bitwise majority voter may be correct or incorrect. Whether this fail-safe is beneficial depends on how likely it is for the bitwise majority voter to be incorrect in these cases, and how costly the fail-safe is.

We model the "error cost" in the sense of a penalty incurred by the system owner for each response that is other-than-correct. We normalize the cost of an incorrect system output to 1, and denote the cost of a failsafe output by *s*. (The cost of a correct output is 0.) A small value of *s* prioritizes integrity whereas a large value prioritizes availability.

The interesting range is $0 < s < 1$: *If* $s \leq 0$, then we should create a trivial system that always outputs the failsafe because it is as good as a correct output; and if $s \geq 1$, we should use the bitwise majority voter and never output the failsafe because it is as bad as an incorrect output. The expectation value of the cost for a given voting system is

$$sPr(failsafe) + Pr(incorrect) \quad (9)$$

The bitwise majority voter is thus preferred (has lower expected cost) if *s* exceeds a breakeven value, namely

$$s > \frac{Pr_{bitwise}(incorrect) - Pr_{simple}(incorrect)}{Pr_{simple}(failsafe)} \quad (10)$$

with the probabilities read from above; note that $Pr_{bitwise}(failsafe) = 0$.

We consider an example where the probability of error in each bit is fixed at $hr = 1/256$, while *h* varies from 1/256 (i.e., a variant has a 1/256 chance of having *all* its output bits flipped) to 1 (i.e., each output bit of a variant has an independent 1/256 chance of being flipped). The breakeven value of *s* is plotted in Figure 39 for several values of *b*. For $b = 1$, there is no difference between the two voting systems. For large values of *h*, the bitwise majority voter is favored because it can correct errors in multiple variants as long as they are in nonoverlapping bits, which is more likely when the bits are independent or nearly so. For small values of *h*, the simple majority voter is favored because, when a variant output has errors, they are likely to be pervasive rather than sparse; thus two erroneous variants are likely to have errors in overlapping bits and the bitwise result is likely to be incorrect, making the fail-safe advantageous. Similarly, increasing the number of output bits favors the simple majority voter because it is more likely that some bit is incorrect in more than one variant.

Next, we incorporate the healing voter into the tradeoff. By analogy with Eq. (10), the bitwise majority voter is preferred over the healing voter if

$$s > \frac{Pr_{bitwise}(incorrect) - Pr_{healing}(incorrect)}{Pr_{healing}(failsafe)} \quad (11)$$

For $b \leq 2$, Eq. (11) is indeterminate because the bitwise majority voter and the healing voter are identical because the bitwise majority result of three 1-bit or 2-bit outputs always matches at least one output. For $b > 2$, the condition in Eq. (11) is stricter than in Eq. (10): The bitwise majority voter is preferred over the healing voter in only a subset of the cases in which it is preferred over the simple majority voter. That is, the

healing voter provides a useful fail-safe over a greater part of the parameter space than the simple majority voter does.

However, the simple majority voter may still be preferred in some cases. The simple majority voter is preferred over the healing voter if

$$s < \frac{Pr_{healing}(incorrect) - Pr_{simple}(incorrect)}{Pr_{simple}(failsafe) - Pr_{healing}(failsafe)} \qquad (12)$$

The condition in Eq. (12) is not satisfied in most cases because the right-hand side is very close to zero if $b$ is large and $r \leq ½$, which corresponds in the $hr = 1/256$ example to $h \geq 1/128$. It is reasonable to expect that $r \leq ½$ because, when errors occur, the output should at worst behave like random coin flips, i.e., it seems unlikely, though possible in principle, that a variant would systematically tend to output the negation of the correct output.

The breakeven values of $s$ for the tradeoffs among all three voting systems are plotted in Figure 39.

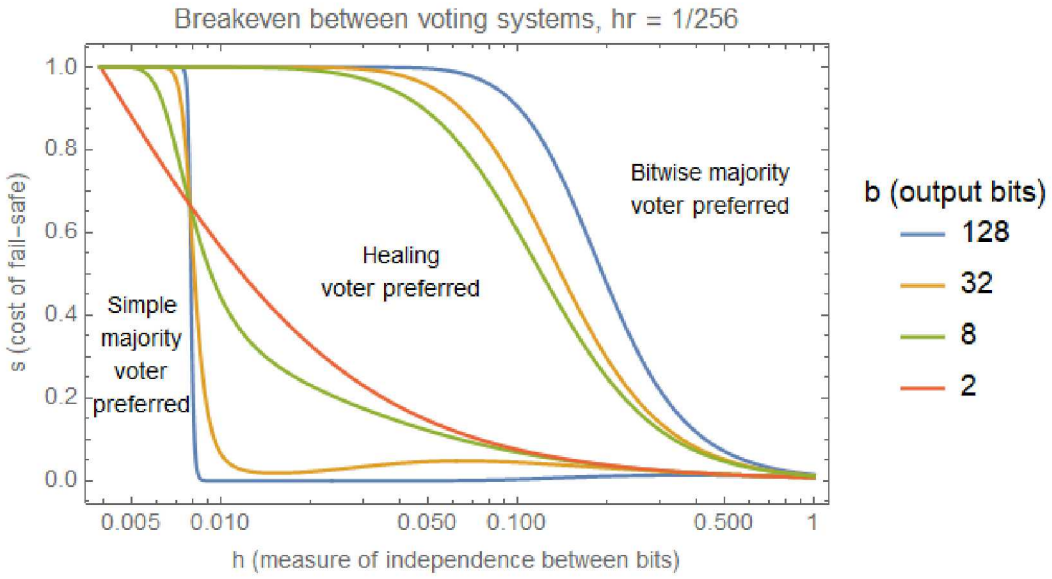


**Figure 39.** Comparison of simple majority, bitwise majority, and healing voters as a function of the independence between output bits

## 5.2. Empirical Results

Now, we evaluate simple majority, bitwise, and healing voters using the selectively randomized AES benchmark circuits generated by the selective randomization process described in Section 6.2. These selectively randomized circuits typically fail to implement the desired encryption functionality on a small fraction of inputs. To mitigate these failures, we propose using majority voting on a diverse set of randomized circuits. Here, we evaluate the performance of three types of voters on empirical data generated by simulating each of three randomized copies of each of the

benchmarks on 100,000 random (key, plaintext) pairs derived from NIST's AES Monte Carlo test vectors [7].

In Table 13 and Table 14 we show the performance of bitwise majority voters, simple majority voters, and healing voters on empirically collected data from the trojanized AES-128 circuits that we diversified in Section 6.2. Table 13 compares the performance for data from circuits generated by using 2,000 known-answer test vectors during generation of the circuit variants, and Table 14 shows results from variants generated using 10,000 known-answer tests during the diversification process. The tables show the number of incorrect simulation results when the collection of diversified circuits, with their outputs combined by the specified voting scheme, is used to encrypt 100,000 known-answer tests. We see that the simple majority voter and healing voter have similar performance, and that this performance is better than that of the bitwise majority voter. The total number of failsafe and incorrect outputs from the simple majority voter is always as least as many as produced by the healing voter, and is sometimes more. This is due to a reduction in the number of failsafe outputs from the healing voter, but comes at the cost of slightly more incorrect outputs.

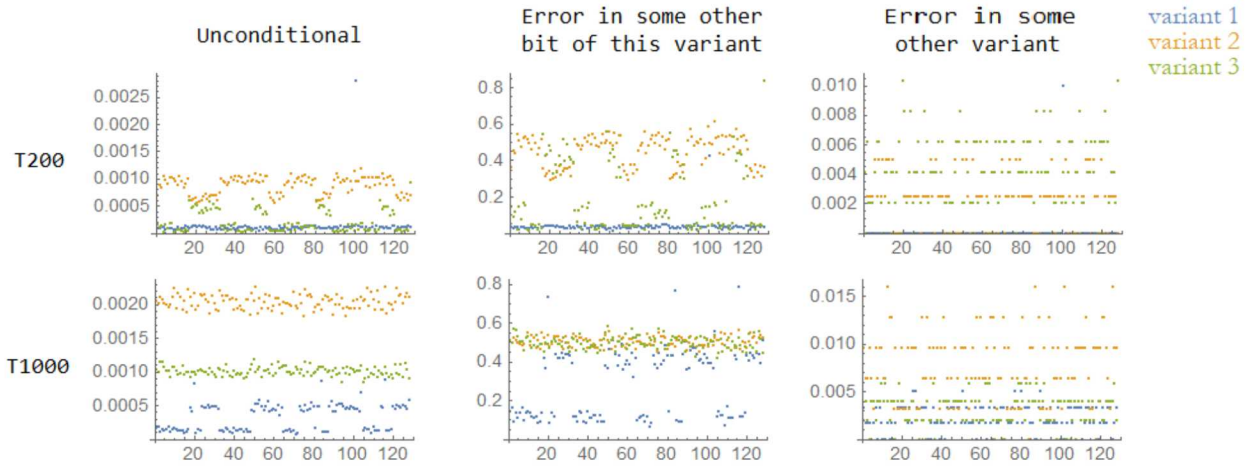


**Figure 40.** Bit error distributions for variants of trojanized AES benchmark circuits

In Figure 40 we study the bit error distributions across the three variants of the trojanized AES circuits. These results are, again, from circuits generated while using 2,000 known-answer test simulation vectors during the diversification process. The left column shows raw error rates in each bit, while the much higher rates in the middle column show that given an error in some output bit of a variant, errors in other output bits of that variant are quite likely. This high correlation is consistent with the expected chaotic error propagation in AES. The right column shows, to a lesser extent, that there is correlation across variants, that is, the same inputs that trigger errors in one variant are more likely to do so in the others as well. This violates the idealized assumption that the variant sampling spreads out the vulnerabilities uniformly over the input space. Yet, we find that majority voting is effective nevertheless.

**Table 13.** Performance of majority voter implementations on selectively randomized AES benchmarks circuits generated using 2,000 training vectors

| Circuit | Bitwise Majority Voter | | Simple Majority Voter | | Healing Voter | |
|---|---|---|---|---|---|---|
| | **Failsafe** | **Incorrect** | **Failsafe** | **Incorrect** | **Failsafe** | **Incorrect** |
| **T100** | 0 | 21 | 21 | 0 | 21 | 0 |
| **T200** | 0 | 3 | 5 | 0 | 2 | 1 |
| **T300** | 0 | 2 | 2 | 0 | 2 | 0 |
| **T400** | 0 | 10 | 10 | 0 | 10 | 0 |
| **T500** | 0 | 4 | 4 | 0 | 4 | 0 |
| **T600** | 0 | 6 | 7 | 0 | 6 | 0 |
| **Circuit** | **Bitwise Majority Voter** | | **Simple Majority Voter** | | **Healing Voter** | |
| **T700** | 0 | 1 | 1 | 0 | 1 | 0 |
| **T800** | 0 | 7 | 7 | 1 | 5 | 2 |
| **T900** | 0 | 5 | 4 | 1 | 4 | 1 |
| **T1000** | 0 | 5 | 6 | 0 | 5 | 0 |
| **T1100** | 0 | 74 | 12 | 62 | 11 | 63 |
| **T1200** | 0 | 67 | 49 | 18 | 48 | 19 |
| **T1300** | 0 | 54 | 38 | 18 | 36 | 18 |
| **T1400** | 0 | 6 | 6 | 0 | 6 | 0 |

| | | | | | | |
|---|---|---|---|---|---|---|
| **T1500** | 0 | 11 | 11 | 0 | 11 | 0 |
| **T1600** | 0 | 13 | 14 | 0 | 13 | 0 |
| **T1700** | 0 | 1 | 1 | 0 | 1 | 0 |
| **T1800** | 0 | 12 | 12 | 0 | 12 | 0 |
| **T1900** | 0 | 66 | 66 | 0 | 66 | 0 |
| **T2000** | 0 | 22 | 3 | 19 | 3 | 19 |
| **T2100** | 0 | 6 | 4 | 2 | 3 | 3 |

**Table 14.** Performance of majority voter implementations on selectively randomized AES benchmarks circuits generated using 10,000 training vectors

| **Circuit** | **Bitwise Majority Voter** | | **Simple Majority Voter** | | **Healing Voter** | |
|---|---|---|---|---|---|---|
| | **Failsafe** | **Incorrect** | **Failsafe** | **Incorrect** | **Failsafe** | **Incorrect** |
| **T100** | 0 | 0 | 0 | 0 | 0 | 0 |
| **T200** | 0 | 0 | 0 | 0 | 0 | 0 |
| **T300** | 0 | 0 | 0 | 0 | 0 | 0 |
| **T400** | 0 | 2 | 2 | 0 | 2 | 0 |
| **T500** | 0 | 1 | 1 | 0 | 1 | 0 |
| **T600** | 0 | 0 | 1 | 0 | 0 | 0 |
| **T700** | 0 | 0 | 0 | 0 | 0 | 0 |
| **Circuit** | **Bitwise Majority Voter** | | **Simple Majority Voter** | | **Healing Voter** | |
| **T800** | 0 | 35 | 21 | 14 | 21 | 14 |
| **T900** | 0 | 7 | 7 | 0 | 7 | 0 |
| **T1000** | 0 | 0 | 0 | 0 | 0 | 0 |
| **T1100** | 0 | 0 | 0 | 0 | 0 | 0 |
| **T1200** | 0 | 13 | 13 | 0 | 13 | 0 |
| **T1300** | 0 | 1 | 0 | 1 | 0 | 1 |
| **T1400** | 0 | 0 | 0 | 0 | 0 | 0 |
| **T1500** | 0 | 0 | 0 | 0 | 0 | 0 |

| | | | | | | |
|---|---|---|---|---|---|---|
| **T1600** | 0 | 0 | 0 | 0 | 0 | 0 |
| **T1700** | 0 | 0 | 0 | 0 | 0 | 0 |
| **T1800** | 0 | 0 | 0 | 0 | 0 | 0 |
| **T1900** | 0 | 28 | 8 | 20 | 8 | 20 |
| **T2000** | 0 | 5 | 2 | 3 | 2 | 3 |
| **T2100** | 0 | 1 | 1 | 0 | 1 | 0 |

# 6. TARGETED CIRCUIT MODIFICATIONS

Due to the theoretical result that there is no program that can decide whether any other program will eventually halt we know that the properties of a program, or circuit, cannot be known in advance of running it [5]. In particular, while we can design a circuit to meet some functional specification and, assuming that this functionality is testable, generate a simulation or test suite to obtain at least probabilistic confidence that the circuit implements the intended functionality; we cannot test a circuit for unintended functionality due to the combinatorially large state space.

Circuits are specified to produce certain outputs given certain inputs. The specification itself is typically of low complexity and is, by definition, what the designer wants the circuit to do. The particular circuit implementation that the designer creates is a single implementation for a provably infinite number of potential designs. More concretely, if we have a simulation suite completely defining the intended input to output behavior for a circuit, then any circuit that passes this simulation suite is correct. If the simulation does not completely specify the input to output behavior, then all circuits that pass the simulation are in some sense equivalent. In this view, the specific circuit created by the designer is only one of many equally valid implementations.

We take advantage of this realization to selectively modify gate level circuit netlists to remove or modify circuitry to create new implementations that still meet the specification. Our intention is to selectively remove or modify hardware Trojans that may exist in the original netlist. To target nets for deletion or modification, we use a neural network classifier to identify suspect nodes within a netlist. The neural network is trained on labeled data from a set of trojanized RS-232 benchmark circuits that is openly available [6]. We test the approach by applying it to trojanized AES-128 benchmarks available from the same source. The neural network generates a list of suspected trojan nets. One by one, we then either remove the suspect net from the circuit or modify its implementation. After this change, we evaluate the circuit by simulating its performance against the AES known answer test vectors and a subset of the AES Monte Carlo test vectors provided by NIST [7]. If the circuit passes each of these tests then the modification is retained. If it fails any of them the modification is removed and we continue to the next suspect node. After exhausting the list of suspect nodes we evaluate the circuit against a set of the Monte Carlo test vectors that were not used during the modification phase. If the circuit passes these tests then we have some confidence that the new implementation is also correct. We then evaluate the behavior of the trojan functionality to determine if it has been eliminated or disrupted.

We will present results showing the effectiveness of targeted deletion and randomization of nets at selectively disrupting trojan functionality while maintain the intended functionality of the AES core. We also present results showing the computational efficiency gains of the neural network targeting approach when compared to random sampling of nets for deletion or randomization.

There is behavior in circuits that we do not need to preserve. For instance, a large IP core may have configurable functionality. After a circuit designer choose a particular configuration for the core the portions of the IP core not needed for the selected configuration become extraneous and can be modified. More generally, any portion of

a circuit that does not directly influence the outputs of the circuit, where outputs are defined as POs and inputs to memory elements. These can be targeted for randomization. In some cases, we may not be able to identify these directly. Instead, we can find targets for randomization, and then test what happens when we randomize those. For finding targets, we can use metrics on the circuit, machine learning based approaches to identify suspect nets, fan-in analysis to identify any portions of a circuit that do not eventually lead to a PO or memory input, netlist reverse engineering RTL re-writing approaches that identify common trojan structures, and genetic programming approaches that seek to generate simpler, trojan-free variants of a circuit.

Additionally, we can also consider the influence of side channels and side channel trojans. We can think of a side channel as behavior that falls outside of some domain of interest. For example, the power side channel exists separately from the logical behavior of a circuit, which is our domain of interest. When we consider randomization and diverse voting, we realize that diverse voting, which operates at a logical level, may not have any meaningful impact on the side channel: if we make three diverse copies of the circuit containing the side channel the power from them is additive and, if the circuit behavior at the output of the voter is not changed then the side channel still exists. Indeed, even if the output of the voter is changed by diversity, the side channel may still exist. Instead, a "moving target" approach that varies the power side channel over time might be more appropriate.

## 6.1. Use Cases for Targeted Randomization

Why would we randomize the implementation of suspect nets rather than deleting these nets? Deleting nets does not result in a diversity of implementations of the circuit; it produces a single new implementation with fewer nets. Since this approach does not produce diverse implementations, it cannot take advantage of the impacts of diversity and moving target (dynamic) diversity on attackers that we have quantified in our models. There are also situations in which not all aspects of an intellectual property block are needed for a design. For example, the block may have some functionality enabled or disabled by licensing or software programming, or may have outputs that are not needed for a particular design. In these cases the unused logic can be randomized to render it, and any trojans that may be within it, benign.

The approaches presented in this section change the functional behavior of the input circuit. They are most appropriate for mitigating trojans that are introduced at the RTL or netlist level. Such trojans may exist in untrusted third party IP (3PIP), could be inserted by untrusted design tools, or may be introduced by some other means.

The composition structures discussed in Section 4 are more appropriate for targeting trojans introduced after a design is complete, for example, during fabrication. This is because they add uncertainty to an attacker relating to the final structure of the circuit and by permitting runtime modification of the circuit. The approaches described in this section and Section 4 are complementary and can be used in concert. In such implementations we recommend that the modifications in this section be made first,

and that the resulting circuit is then implemented with one of the composition structures.

## 6.2. Trojan Targeting with Machine Learning

An overview of an approach for using machine learning to target trojan nodes for randomization or deletion is shown in Figure 41. We assume that we have access to a netlist for the circuit and to a comprehensive suite of simulation vectors that evaluate the netlists ability to implement the desired functionality. We also assume that we have a classifier, illustrated in Figure 41 as a neural network, that has been trained to differentiate between trojan and benign nets in a netlist. Given these inputs, we first convert the netlist to a graph based representation, as shown in Figure 43. To do this, we create a vertex for every primary input, primary output, gate output and latch output in the design. We then connect two vertices with an edge if one of the vertices appears at a gate or latch input and the other appears at the output. We consider both directed and undirected graphs. For directed graphs, directionality is from gate or latch inputs to gate or latch outputs. Next, we calculate a set of graph metrics for each vertex in the graph for the classifier to use as features. These features are calculated with the python library Networkx [49]:

- Clustering coefficient (undirected graph)
- Degree centrality (directed and undirected graphs)
- In degree centrality (directed graph)
- Out degree centrality (directed graph)
- Average neighbor degree (directed and undirected graphs)
- Number of triangles including this node (undirected graph)
- Square clustering coefficient (directed and undirected graphs)
- PageRank (directed and undirected graphs)
- In degree (directed graph)
- Out degree (directed graph)
- Degree (directed graph)

Additionally, we also use the degree-2 polynomial features of these, which are generated by multiplying each feature with each of the other features, including itself[†].

Next, we use the neural network to classify each of the nodes as either benign or suspect. We collect the suspect nodes into a list to target for randomization or deletion. We then work through this list iteratively. Beginning with the first suspect net, we either remove it from the circuit or randomize its implementation. For randomization, we find the transitive fanin cone for the net and then randomly choose to apply gate

[†] Development of the neural network was performed under a different project

replacement, gate addition, or dynamic output version to each gate in the transitive fanin cone. After this randomization is complete we simulate the behavior of the circuit with a subset of the simulation vectors. If the expected result is obtained for all of these simulation vectors then the randomization is kept in the circuit. If any of the simulations fail, then we revert the randomization. We then repeat this process for each suspect net. At the end of this process we have a circuit that passes a subset of the available simulation vectors, and, if it was possible to modify or delete any of the suspect nodes, has a different implementation than the input netlist. We then simulate this new netlist on the full suite of simulation vectors to ensure that it still implements the intended functionality. Note that we could use the full suite of simulation vectors during generation of the diversified circuit, but that this would increase computational effort and would not allow us to determine whether the modifications are likely to generalize to input conditions that are not covered by the simulation suite. Finally, note that the process of identified a list of suspect circuit elements, iteratively modifying the implementation of those suspect elements, and keeping only those modifications that preserve intended behavior can be used with any approach for identifying suspect circuit elements. We consider neural networks as the targeting approach in the current work. Additional targeting approaches are described in Section 6.3 and Section 6.5.

To evaluate this approach we used scikit-learn [50] to train a neural network on labeled data from a subset of the trojanized RS232 and wishbone benchmarks available from trust-hub [26]. For these experiments we consider only the trojanized benchmarks provided at the RTL level (RS232-T100 through RS232-T901, wb_conmax-T200). The approach described in this section operate on gate level netlists, not RTL descriptions, so we synthesize the benchmark circuits and map them to a simple gate library consisting of an inverter and two-input AND, OR, XOR, NAND, NOR, XNOR gates and a latch, prior to our experiments. We do not consider the benchmarks provided at the netlist level because we do not have access to the gate library used to map those netlists. We convert these netlists to graphs and extract feature vectors for each node in each of the circuits. We then label these as benign or trojan using our knowledge of the benchmarks. Finally, we modify the resulting training data by using a dynamic weighting approach to ensure that we have approximately equal amounts of benign and trojan nets in the training set. For this we first remove any duplicated rows in the training data. Then, assuming that there are $N_{min}$ samples in the minority class and $N_{maj}$ samples in the majority class, we repeat each sample from the minority class $round(N_{maj} / N_{min})$ times. We use the resulting data to train a neural network.

After training the neural network we test our approach on some of the trojanized AES benchmarks from trust-hub. As before, we consider only those benchmarks provided at the RTL level (AES-T100 through AES-T2100). Since our method operates on netlists, we first synthesize and map these circuits to our simple gate library. As with the neural network training, we do not consider the trojanized AES benchmarks provided at the netlist level because we do not have access to the gate library used to map those netlists. After synthesizing and mapping the trojanized AES benchmark to a netlist we convert the netlist to a graph and extract the graph metrics, and calculate the

interaction and polynomial features. Then we use the neural network to identify a list of suspected trojan nodes. Then we iteratively remove or randomize each of these nodes, running a simulation consisting of either 2,000 or 10,000 known-answer tests from NIST after

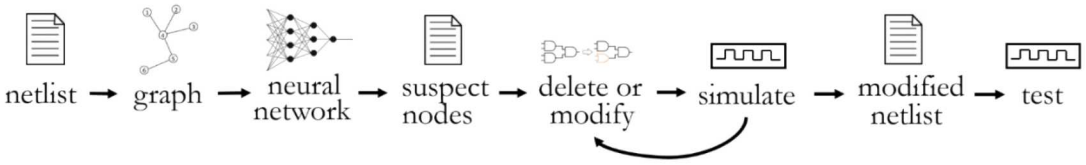


**Figure 41**. Our overall approach to identify and selectively randomize trojan nets

each randomization [7]. These test vectors were randomly sampled from the Monte Carlo encryption tests in the NIST suite. If any of these tests fail then the randomization is reverted. Otherwise, the randomization is left in the circuit. In either case, we then proceed to the next node. This process continues until we have attempted to remove or randomize each of the nodes in the list of suspect nets. Note that in our experiments we consider node removal and node randomization separately. That is, we performed one set of experiments in which we removed suspect trojan nodes, and an additional set of experiments in which we randomized them. After evaluating the list of suspect nodes we then simulate the resulting circuit on 100,000 known-answer test vectors from NIST. This simulation suite consists of all 100,000 Monte Carlo decryption test vectors in the NIST suite. Since our circuit is an encryptor, we first converted these decryption tests to the corresponding encryption tests. All of these tests are different than those used during generation of the diversified circuits. This simulation gives us additional evidence about the ability of the diversified circuit to implement the desired AES functionality. Finally, since we know the trigger conditions for the benchmarks, if there are any, and the behavior of the payload, we then simulate the trojan behavior of the diversified circuit to see if the trojan has been impacted.

### *6.2.1. Efficiency of Neural Network Targeting Compared to Random Selection*

Rather than using a neural network to identify suspect nets, we could randomly select nets for modification or deletion. To compare these options we define the *efficiency* of the neural network approach as the number of suspicious nodes identified by the neural network divided by the number of nodes we would have to randomly select to identify the same number of trojan nodes as the neural network. This comparison is shown in Figure 43 for the case where we allow primary inputs (PIs) and primary outputs (POs) to be trojan nodes, as well as the case were we assume that PIs and POs are not trojanized. We find that the neural network targeting approach requires evaluation of only about 7-12% as many nodes as a random targeting technique, which is a considerable reduction of computational effort. For many of the benchmarks the neural network approach requires targeting 1% or fewer of the nodes as would be required with a random targeting approach.

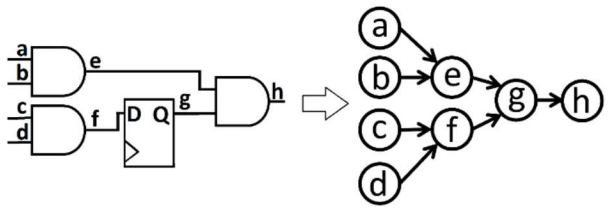


**Figure 42.** We convert circuit netlists to graphs by creating a vertex for each net in the netlist, with edges connecting vertices *i* and *j* if *j* is the output of a gate and *i* is an input to the same gate

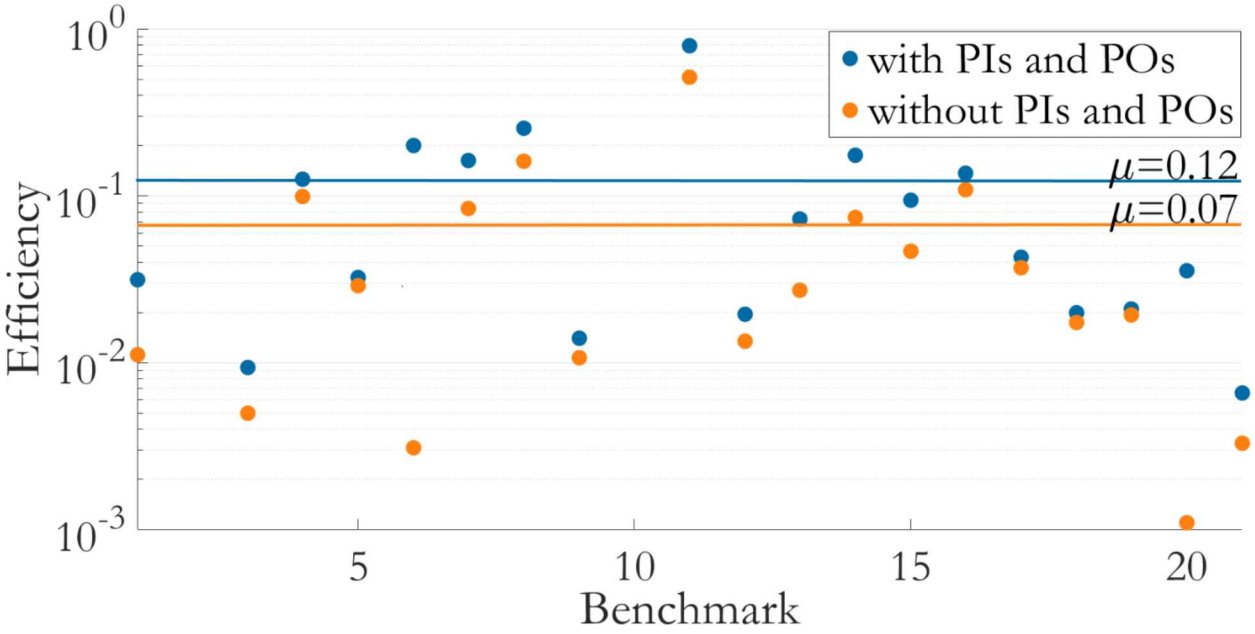


**Figure 43**. We define the efficiency of the neural network targeting as the number of suspicious nodes identified by the neural network divided by the number of nodes we have to randomly select to identify the same number of trojan nodes as the neural network

### *6.2.2. Results*

After identifying suspicious nodes, we either delete them or randomize them. When using our selective node deletion approach, 17 of the 21 trojans were disabled. In two of the remaining four cases, T200 and T1000, the node deletion approach was not able to remove any nets from the design. In the other two circuits, T1100 and T1200, some nodes are deleted, but the deletions fail to impact the trojan. In T1100 the trigger is removed from the circuit, but the payload remains present, active, and unmodified. In T1200 most of the trigger logic has been removed, but as with T1100 the payload remains active and modified. In all cases, the diversified circuits pass all of the simulations in the post-diversification test suite.

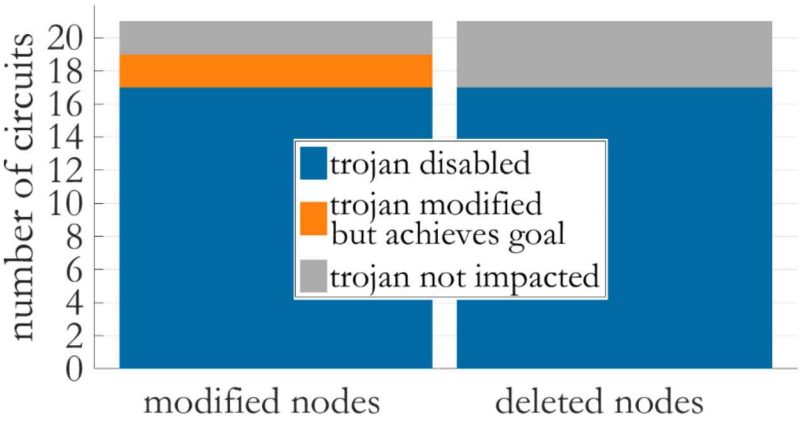


**Figure 44.** Selective deletion and selective randomization of circuit nodes both disable 17/21 trojans. Selective randomization impacts an additional 2/4 remaining trojans, while both approaches failed to impact 2/4 trojans.

Selectively randomizing nodes disables 17 of the 21 trojans. Of the remaining trojans two are modified, but still achieve their goal. These specific trojans, T300 and T1800, are designed to drain battery power from an embedded device by continuously rotating a shift register. We find the diversified circuits have modified this payload, but that the shift registers still update their values. Due to this, we assess that the trojan will still be able to achieve its goal of prematurely draining the battery. In the final two circuits, T200 and T1000, the trojans are not impacted. The diversified circuits usually fail on a small percentage of the simulations in the post-diversification test suite. The number of failures is a function of the number of simulation vectors used during the modification step. When we use 2,000 simulation vectors during the generation of the diversified circuits the average diversified circuit fails 330 of the 100,000 post-diversification test vectors, while circuits generated with 10,000 simulation vectors fail only 98 of the post-diversification simulations. This is a reduction of 70%.

We also find that majority voting of diverse modified copies is useful. To create a diverse, redundant collection of circuits we repeat the entire diversification process, from training of the neural networks through identification and diversification of suspect nodes, three times. We then combine the outputs from these three circuits with a simple majority voter. When using 2,000 simulation vectors during the diversification process this results in an average of 19 simulation failures in the post-diversification test suite, a reduction of 94% from the single circuit case. When using majority voting and 10,000 simulation vectors during the diversification step we have an average of 4 failures in the post-diversification simulations, a reduction of 95% from the single circuit case. Using 10,000 simulation vectors during diversification and majority voting results in a 77% reduction in post-diversification errors when compared to using majority voting and 2,000 simulation vectors during the diversification step. These results are summarized in Figure 45.

### 6.2.3. *Discussion*

Using machine learning to target suspecting trojan nodes for randomization or deletion appears to be a useful approach for disrupting or eliminating trojan triggers and payloads

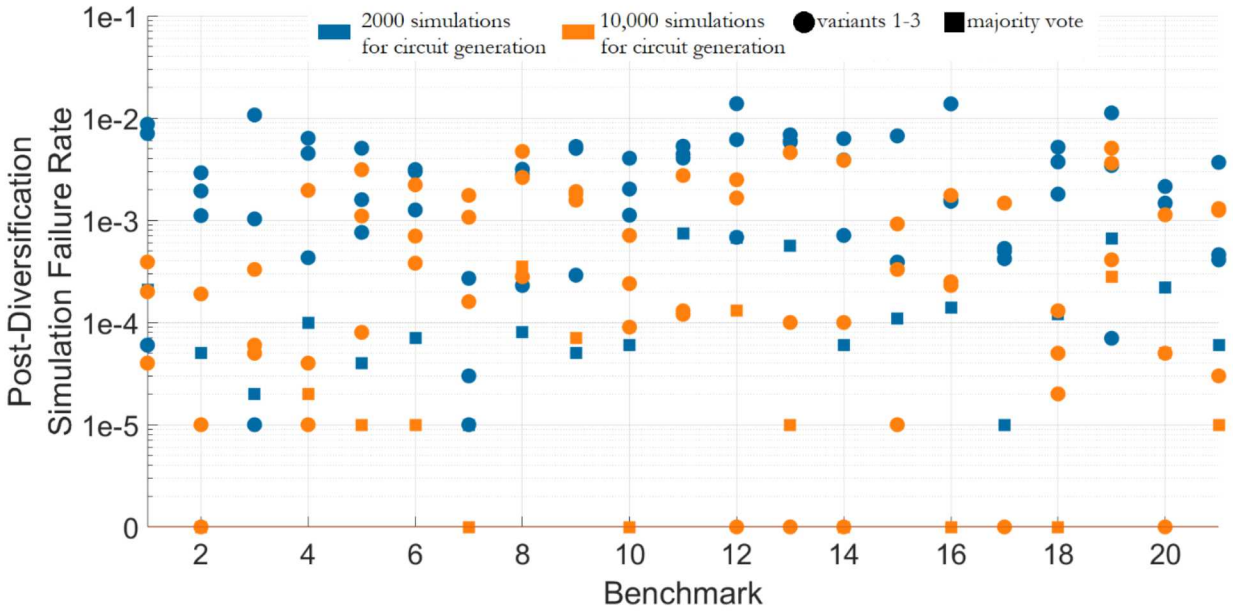


**Figure 45.** Comparison of the number of simulation failures from encrypting 100,000 random AES-128 (plaintext, key) pairs when using 2,000 and 10,000 simulation vectors during the circuit modification step and with and without majority voting.

from circuit netlists. Our initial experiments are promising, showing that the technique is able to eliminate or disable trojans in 81% of the test cases. Future work should further explore this technique by applying it to a broader set of benchmarks. It would also be useful to explore refining the neural network targeting approach to be more selective at targeting trojan nodes. Our current prototype implementation does a good job of identifying trojan nodes, but it has a high false positive rate. These false positives result in wasted computational effort for randomizing benign nodes, simulating the resulting circuit, and ultimately reverting the randomization. Improving the false positive rate would make the approach more attractive for use in practice, and would allow it to be applied to larger circuits. It would also allow us to increase the number of simulation vectors used during the circuit modification phase, which has a dramatic improvement on the correctness of the diversified implementations, particularly when they are used in majority voting architectures.

## 6.3. Trojan Targeting through Identification of Common Trojan Structures

Rather than attempting to identify suspicious nets with a circuit for targeted randomization, we can instead identify structures within a circuit that commonly appear in trojans. While these structures are likely to appear within benign portions of the circuit as well, we can avoid disrupting the intended functionality of the circuit by iteratively modifying a structure of interest, simulating the resulting circuit, and then

choosing to retain the modification if the circuit still functions as intended or to revert the modification if the desired circuit functionality has been lost. We then advance to the next structure of interest and repeat this process until all of the identified structures have been randomized. For these experiments, we focus our efforts on two common types of trojan trigger mechanisms: time bombs and comparisons to rare values.

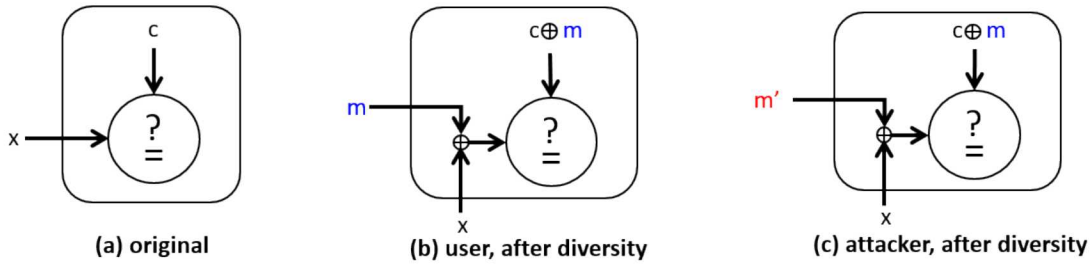


**Figure 46.** Masking circuit structures, such as comparators, with a key prevents the circuit from functioning correctly for users without knowledge of the correct key

However, we note that additional common trojan structures, such as shift registers, could also be targeted.

### *6.3.1. Register Transfer Level*

Many trojans are triggered by a comparison circuit. Typically, these triggers involve comparing an input or internal signals of the circuit to some rare, predetermined value that is known to the trojan designer. The attacker can then supply the necessary inputs to the circuit to satisfy the trigger condition and activate the payload. Time bomb trojans are a variation of this and are triggered after a predetermined number of clock or execution cycles. For example, a Time bomb in an encryption circuit may be triggered after a specified number of encryptions have been performed.

An alternative approach is to identify the structure of interest and then randomize some portion of it without providing a recovery path. In the comparator example, this might involve randomizing the constant-valued input to the comparator, but not adding an unmasking input. In many cases, this will break the intended functionality of the circuit. However, by identifying all of the comparators in the design, we can iteratively randomize each of them and then run the circuit's simulation suite. If these simulations pass, then it is safe to maintain the randomization. If they fail, then the randomization can be reverted prior to moving on to the next circuit.

#### 6.3.1.1. Approach

To implement this approach we prototyped software for modifying RTL descriptions of circuits written in Verilog. For this, we initially parse the Verilog files and convert them to a syntax tree using an open source tool [53]. We then scan the syntax tree to find comparators.

We implemented two approaches for modifying comparators. The first approach simply replaces one of the compared values with a random value. The second approach implements the technique illustrated in Figure 46 by masking one of the

comparator inputs and adding a user-input unmasking value to the second input. The unmasking value is propagated to the toplevel, where it becomes a primary input to the circuit.

#### 6.3.1.2. Results

This approach will impact any trojan triggered with a comparator, including 18 of the 21 trojanized AES-128 benchmarks available from [26]. In particular, it would disable benchmarks T400-T2100. The remaining benchmarks, T100-T300, do not have triggers. These trojans also do not include comparators, and so they cannot be impacted by this specific approach.

### *6.3.2. Netlist Level*

In this section we develop a technique to disable hardware trojans in gate-level netlists through automated modification of common trojan structures. In particular, we introduce diversity to suspect structures to alter trojan trigger conditions, rendering the trojan unusable by the attacker. In this discussion we focus our attention on identifying and modifying comparators, which are often found in trojan triggers, as a proof of concept. Similar approaches can be used to target additional structures of interest. The netlist reverse engineering approach used in this work was motivated by [46].

#### 6.3.2.1. Approach

We begin by synthesizing the circuit to our reduced gate library, which consists of an inverter, buffer, latch, and 2-input AND, OR, XOR, NAND, NOR, and XNOR gates. We then convert the resulting netlist to a graph. We operate on this graph representation of the netlist to target and disable trojans in the design. There are five processing steps required for the targeting and disablement:

1. graph reduction
2. functional simulation
3. bit slice enumeration
4. suspect slice identification
5. diversification and verification.

These are described in the following subsections.

#### 6.3.2.2. Graph Reduction

Prior to operating on the graph we first ensure that it is a directed acyclic graph (DAG). We ensure this by first converting the original graph to an S-Graph. An S-Graph of graph *G* is a graph *G'* which has a number of vertices equal to the number of flip-flops in *G*. In G' an edge exists from vertex $V_1$ to vertex $V_2$ if there exists a combinational path from flip-flop 1 to flip-flop 2 in *G*. Figure 48 shows the graph from Figure 47 after reducing it to an S-Graph.

Next, we detect and remove cycles in the graph by applying a depth-first search at every node [47]. The complexity of this operation is *O(V*(V+E)),* where *V* and *E* are vertices and edges in a graph, respectively. It may be possible to use heuristics to improve the performance of this operation.

For all removed edges, we also remove the corresponding source flip-flop. The remaining noncyclic flip-flops are ignored, so all of the fan-in vertices from the source flip-flop are connected to fan-out vertices. We ignoring noncyclic flip-flops, rather than greedily removing all flip-flops, to account for trigger structures specifically designed

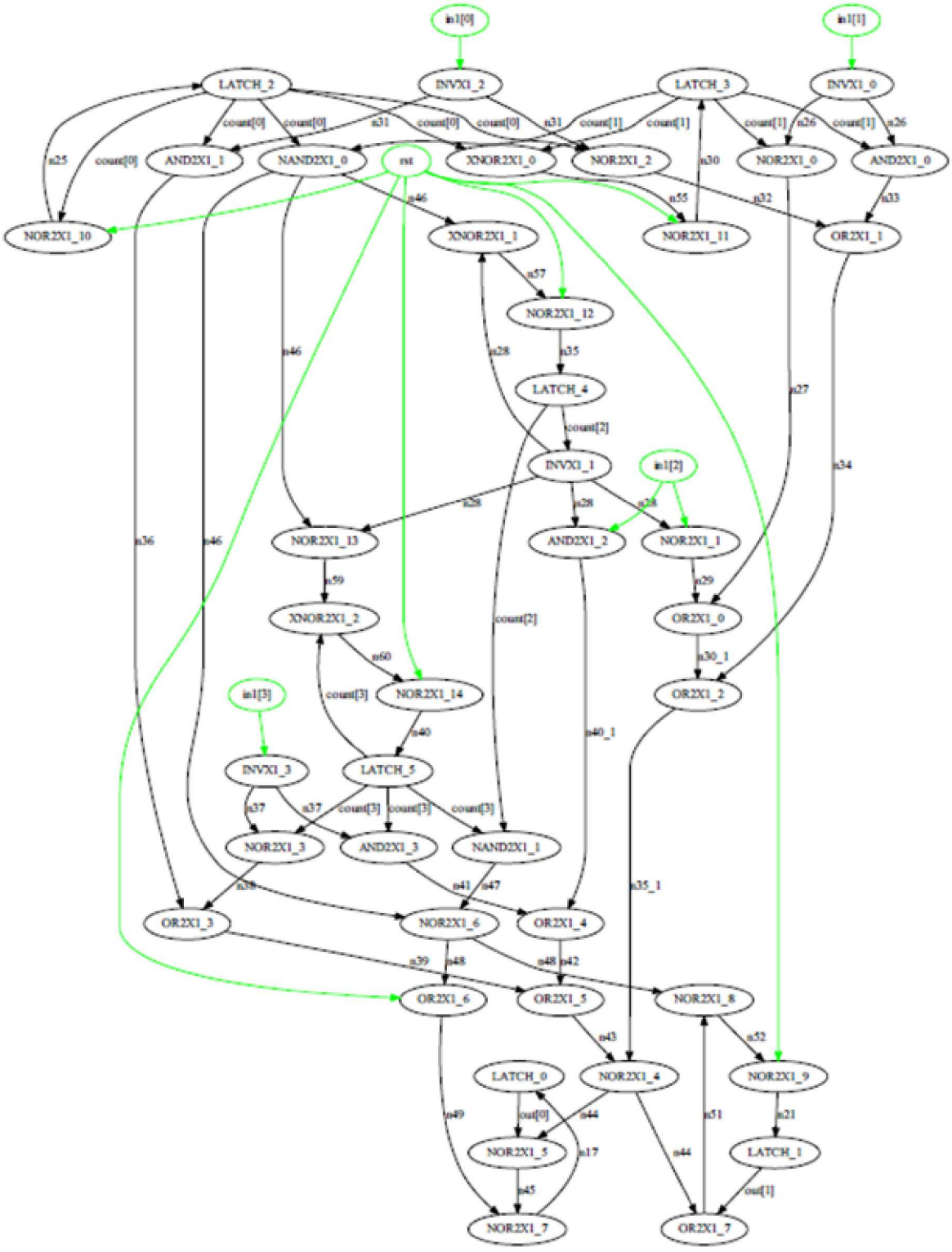


**Figure 47.** Graph representation of sample 4-bit counter with comparator and 2-bit output.

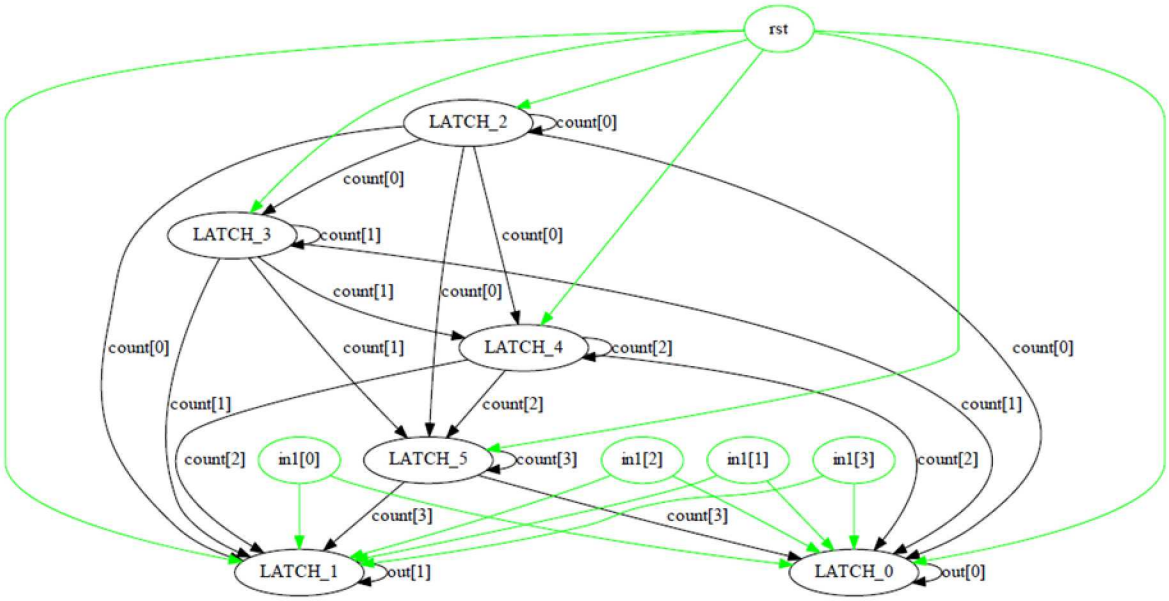


**Figure 48.** S-Graph representation of sample 4-bit counter with comparator.

to avoid detection [48]. Additionally, to simplify the bit slice enumeration step we also remove all inverters under the assumption that the synthesis tool will tend to incorporate inverters into gates, for example, by selecting an NAND gate rather than an AND gate and an inverter. This option can be disabled to increase the rigor of the analysis. Figure 49 shows the graph from Figure 47 in reduced DAG, no inverters form.

#### 6.3.2.3. Functional Simulation

After converting the circuit to a graph representation, as shown in Figure 47, the circuit is simulated using 10,000 randomly generated input vectors. Note that fewer, or more, input vectors could be used for this simulation. From these simulations we estimate the probability of each node in the circuit having logic value 1 as the fraction of inputs for which the node obtains value 1. The resulting signal probabilities $s_{p0}$, $s_{p1} \in (0,1)$ are used to identify suspect nodes. To do this we choose a rare signal threshold, $\theta$. Any node with a signal probability $s_{p0} < \theta$ or $s_{p1} < \theta$ is considered rare. We then consider only rare nodes in our search for potential trojan structures, greatly reducing the search space.

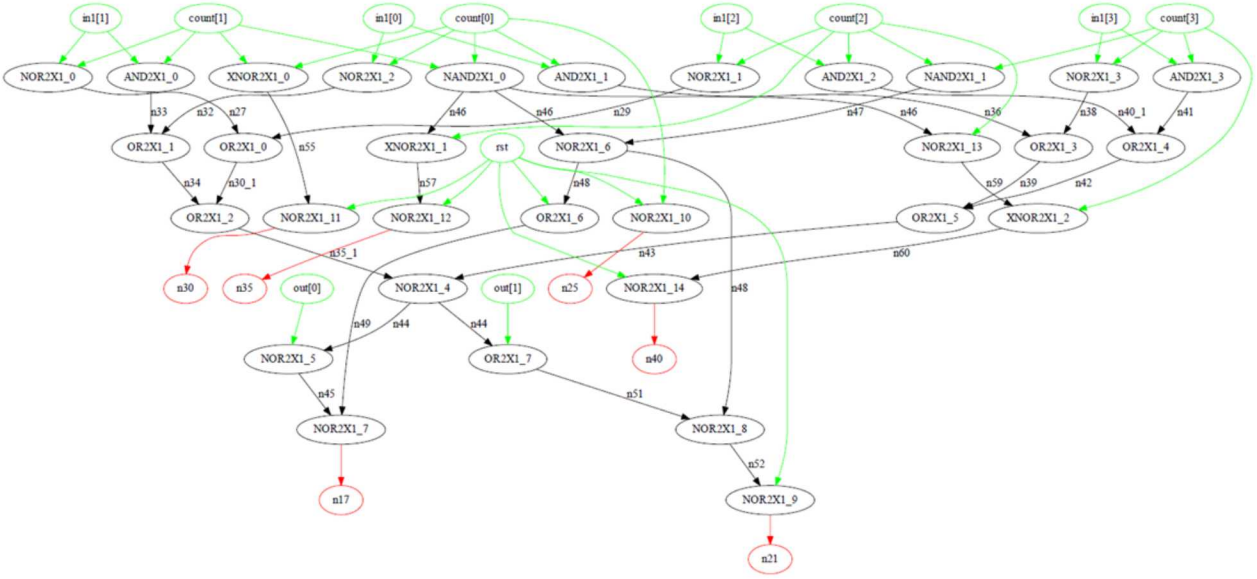


**Figure 49.** Final reduced graph of sample 4-bit counter with comparator.

#### 6.3.2.4. Bit Slice Enumeration

From the list of suspect nodes, we then enumerate all $k$-bit slices where a $k$-bit slice is defined as a set of nodes in the fan-in of a subgraph $H$ of $G$ such that $H$ has no more than $k$ inputs. For this all $k$-bit slices, $k \in [i,j]$ are recursively enumerated for each suspect node. We note that it is possible for a $k$-bit slice to have an "unclean" fan-in, with inputs unrelated to the suspect function. Such scenarios are possible due to synthesizer optimizations, which can prevent correct identification of suspect structures. Typical solutions to this problem involve providing the $k$-bit slice to a Quantified Boolean Formula (QBF) solver. However, because we enumerate the bit slices for all suspect nodes it is likely that a "clean" version is also included in the enumerations. For example, consider the circuit shown in Figure 50. If we have a rarity threshold of $\theta = 0.05$ then both clean and unclean versions of the comparator will be considered.

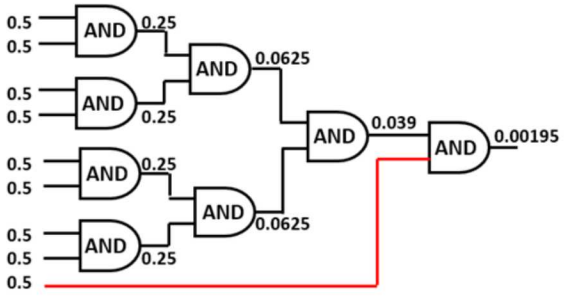


**Figure 50**. Sample circuit with toggle rate for each wire. The red nets is unrelated to the comparator structure of interest. If it is included in a bit slice, then that slice will be unclean.

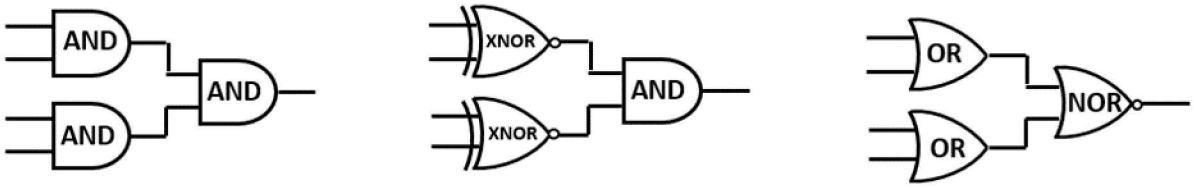


**Figure 51.** Template "==" comparator structures.

#### 6.3.2.5. Suspect Slice Identification

To identify suspect slices, we must be able to match structures within the circuit of interest to known suspect structures. To do this we extract the reduced order binary decision diagram (BDD), a canonical representation of the Boolean formula, for each slice. Working at the BDD level is important for properly considering all possible structures of interest at the netlist level. The extracted BDDs are compared to the BDDs of template suspect structures with the same input order. In this work we target comparators, which are often found in trojan triggers The template comparators that we consider are shown in Figure 51. If a slice's BDD matches the BDD of one of the template comparator structures then that slice is marked as suspicious.

#### 6.3.2.6. Diversification and Verification

After identifying suspect structures, we next diversify the implementations of those structures. The goal of this diversification is to disrupt the trigger structure so that the attacker is unable to activate the Trojan using the original trigger. This effectively eliminates the attacker's ability to make use of the trojan, rendering it benign. One approach is random diversification, such as arbitrarily adding, removing, or changing gates within the structure of interest as in Section 6.2. However, such a technique may only change the rarity of activation in the trigger condition. For example, consider Figure 52. If AND3 is converted to an OR gate, the Trojan activation probability goes from 1/16 to 7/16, with the original trigger condition still valid. This may be problematic during verification that the modification preserves the intended functionality of the circuit, as a functional trojan effect will activate, potentially change the output of the circuit, and ultimately fail the verification step. The modification would then be reverted, which would restore the trojan functionality. This was less problematic in Section 6.2 since in that approach we are able to target arbitrary portions of the circuit and can modify trojan payloads in addition to triggers. However, when targeting comparators with the current approach we are typically only modifying trojan triggers and so we must be more precise in our diversifications. Consequently, a more fine-tuned diversification technique is needed. We choose to diversify the suspect structure by changing the type of comparator and adding inverters to the inputs. This preserves rarity of the trigger condition, but randomizes the triggering value, making the trigger unusable to the attacker.

Slices marked as suspect may be benign parts of the original circuit. To ensure that we do not change these, which could disrupt the intended function of the circuit, we perform a verification step after each diversification to determine the validity of circuit modification. For every diversification applied, a comprehensive test bench is run to

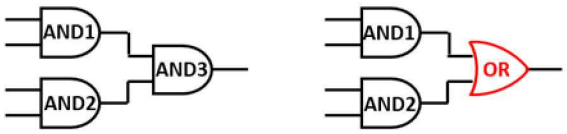


**Figure 52.** Example diversification.

ensure functionality is maintained. If at any point the test bench fails, the diversification under test is reverted.

#### 6.3.2.7. Results

To evaluate our approach, we ran the comparator identification on the RS232-T400, RS232-T700, RS232-T800, and AES-T500 from Trust-Hub [26]. Table 15 shows the identification results. We are able to successfully identify all Trojan structures in the designs, along with several benign structures. For AES, while there are fewer than 91 comparators in the design, several identified comparators make up the structure of larger comparators (128-bit).

We tested the diversification process on AES-T500. For this, all 91 of the identified comparator structures are diversified as described in Section 6.3.2.6. After each diversification we simulated the resulting circuit using some of the NIST known-answer AES tests to ensure that the diversification maintains the intended AES functionality. In this case, all 91 diversifications are kept in the final circuit, as they each pass the NIST simulation suite. Next, we determined whether the diversification impacted the trojan by running a second test bench with the correct trigger sequence for the Trojan to identify which of the diversifications modified the Trojan trigger. We found that 24 of the 91 diversifications mitigate the Trojan by disrupting all or part of the 5-stage trigger. The remaining diversifications have no effect on the Trojan trigger, but are kept in the circuit since they do not change the desired AES functionality.

Future work includes expanding the identification templates to include more known common elements of trojan circuits such as combined ‘<’ and ‘>’ comparators, shift registers, ring oscillators, linear-feedback shift registers (LFSR) and nonlinear feedback shift registers (NLFSR). Other features, such as connectivity analysis, could be incorporated to aid in identifying additional trojan nodes.

In enumerating the bit slices, we notice that oftentimes several suspect comparators are part of a larger comparator structure. Due to this we can reduce the number of suspect comparators by applying propagation analysis to merge similar neighboring structures. We can apply Roth’s D-calculus for symbolic simulation on the design in this situation. However, deciding to merge suspect comparators reintroduces the problem of “unclean” slices, which will require the use of a QBF solver.

Verification of component diversification involves validating functional correctness with diversification in place. However, as discussed earlier, there is the potential for trojans to bypass our method if the diversification activates the trojan and the trojan causes the circuit behavior to deviate from the intended behavior of the design. If the trojan is activated during the test bench, either from the diversification decreasing

**Table 15.** Trojan structure identification

| Benchmark | No. Gates | $\theta$ | Bit slice Bounds [*i,j*] | No. Suspect Structures | Fraction of suspect structures that are part of a trojan |
|---|---|---|---|---|---|
| **RS232-T400** | 322 | 0.1 | 4,6 | 20 | 1/20 |
| **RS232-T700** | 363 | 0.1 | 4,6 | 26 | 3/26 |
| **RS232-T800** | 304 | 0.1 | 4,6 | 20 | 1/20 |
| **AES-T500** | 298323 | 0.001 | 4,6 | 91 | 24/91 |

activation rarity of the trojan or due to a lucky test pattern, the test bench will fail and the diversification will be reverted. One possibility for mitigating this is to apply multiple different diversifications to the same structure, and then use a majority voter to combine the outputs from the various diversifications.

## 6.4. Trojan Targeting with Genetic Programming

We begin by postulating that an uncompromised circuit is by some measure "simpler" than a compromised circuit because the compromised circuit implements both the intended circuit functionality and the trojan functionality. If we accept this postulate then, if we can identify an appropriate measure of simplicity, it may be possible to automatically generate circuit variants that are simpler than the input circuit and in the process eliminate the trojan functionality. We propose genetic programming as one technique for generating these circuit variants.

### *6.4.1. Approach*

We performed an initial exploration of the use of using genetic programming for circuit modification. The objective was to identify a scheme that uses genetic programming to generate diverse implementations of select Boolean functions. Genetic programming addresses the problem of having computers learn to program themselves by providing a framework to search the space of possible computer programs for a program that solves a given problem most efficiently. To leverage this for circuit modification, we instead want to use genetic programming to create diverse implementations of a Boolean function.

Genetic programming is a highly parallel mathematical algorithm that transforms a set (population) of individual mathematical objects (hierarchical tree structures), each with an associated fitness value, into a new population using operators patterned after the Darwinian principle of reproduction and survival of the fittest, and after naturally occurring genetic operations such as reproduction and recombination. In genetic programming, populations of many individuals are genetically bred using the Darwinian principle of survival and reproduction of the fittest along with a genetic recombination (crossover) operation appropriate for combining Boolean functions.

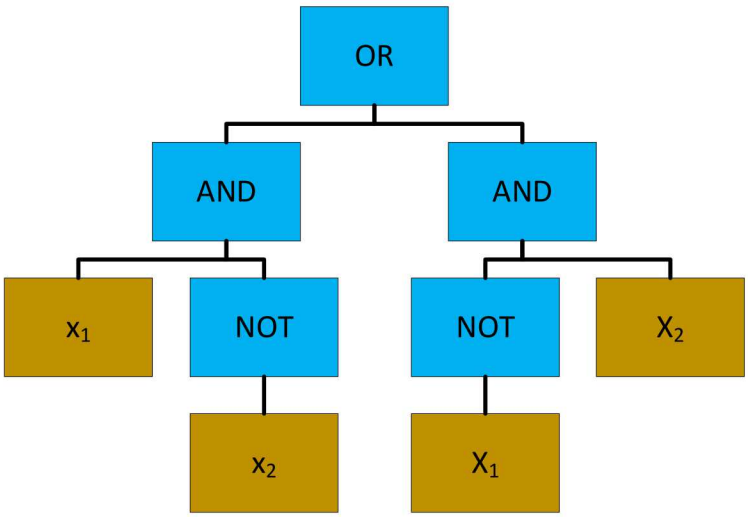


**Figure 53.** A tree representing the XOR of inputs $x_1$ and $x_2$

Genetic programming creates variants by executing the following steps:

1. Generate an initial population of random compositions of the functions and terminals of the problem
2. Interactively perform the following sub steps until the termination criterion has been satisfied:
    a. Execute each Boolean function in the population and assign it a fitness value according to how well it solves the problem
    b. Create a new population of Boolean functions by applying the following two primary operations. The operations are applied to Boolean functions in the population with a probability based on fitness:
        i. Copy existing functions to the new population
        ii. Create new Boolean functions by genetically recombining randomly chosen parts of two existing functions

To explore this concept, we studied the use of genetic programming for evolving the 'XOR' function. This can be represented by the tree structure T=[x1,x2], with functions F=[OR,AND,NOT], as shown in Figure 53.

### *6.4.2. Results*

We used the opensource MATLAB toolbox GPLab [54] for our experiments. Additional useful references are found in [55-58]. There are three steps needed for evolving the function:

1. SET VARS – This module initializes the parameters
2. GEN POP – Generates the initial population and calculates its fitness
    a. Three initialization methods exist:
        i. Full
        ii. Grow
        iii. Ramp Half and Half

b. By default, the fitness is the sum of the absolute difference between the obtained and expected results

3. GENERATION – Generates a new population by applying genetic operators (tree crossover, tree mutation)
    a. Parents are selected from the pool through one of the following four sampling methods:
        i. Roulette
        ii. Stochastic Universal Sampling
        iii. Tournament
        iv. Lexicographic Parsimony Pressure Tournament
    b. Three methods to calculate the expected number of offspring are:
        i. Absolute
        ii. Rank85
        iii. Rank89
    c. Repeats itself until the stopping conditions are met or maximum generation is reached

An example for 100 generations with a population of ten for the XOR gate is presented in Figure 54 through Figure 57.

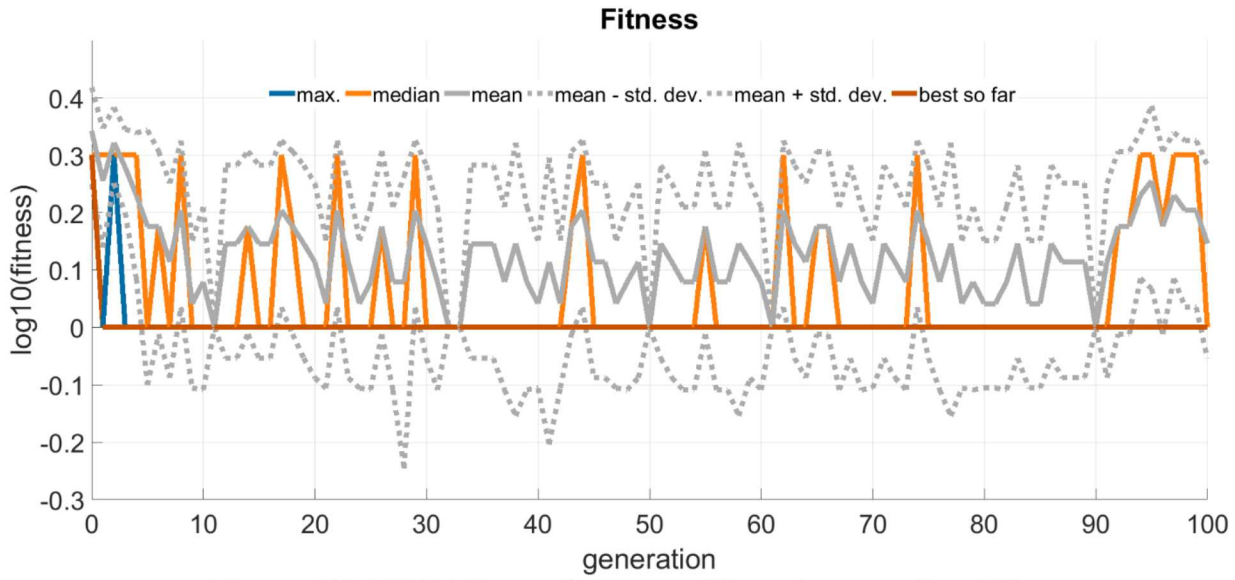


**Figure 54**: XOR Fitness (stop condition at generation 87)

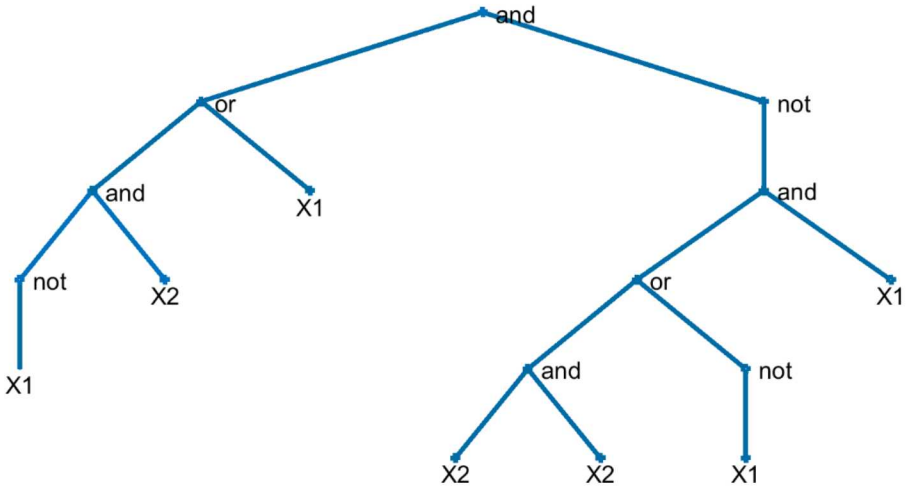


**Figure 55**: Tree Representation

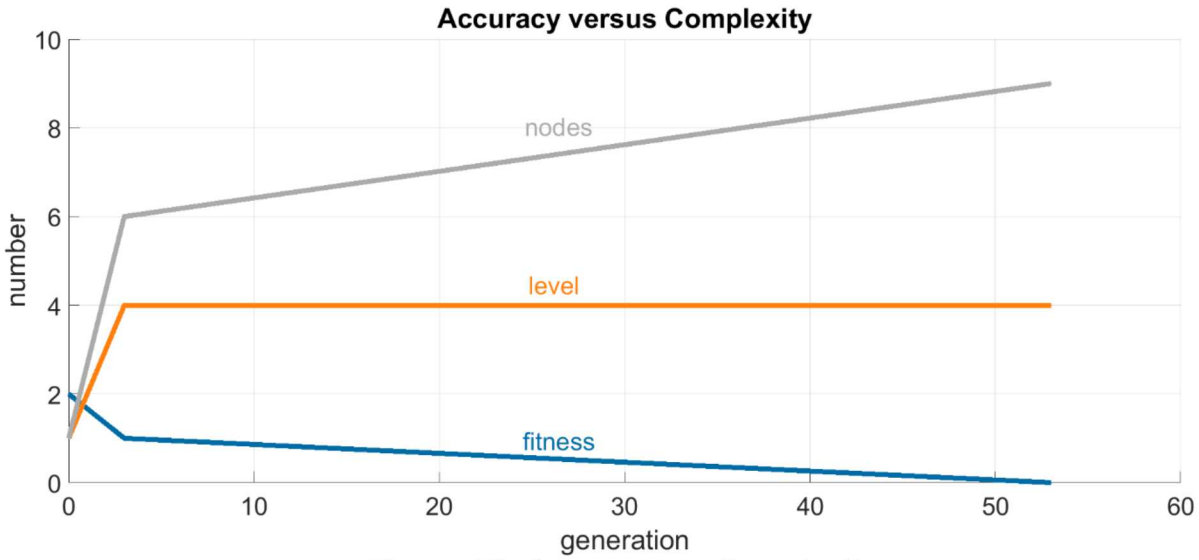


**Figure 56:** Accuracy vs. Complexity

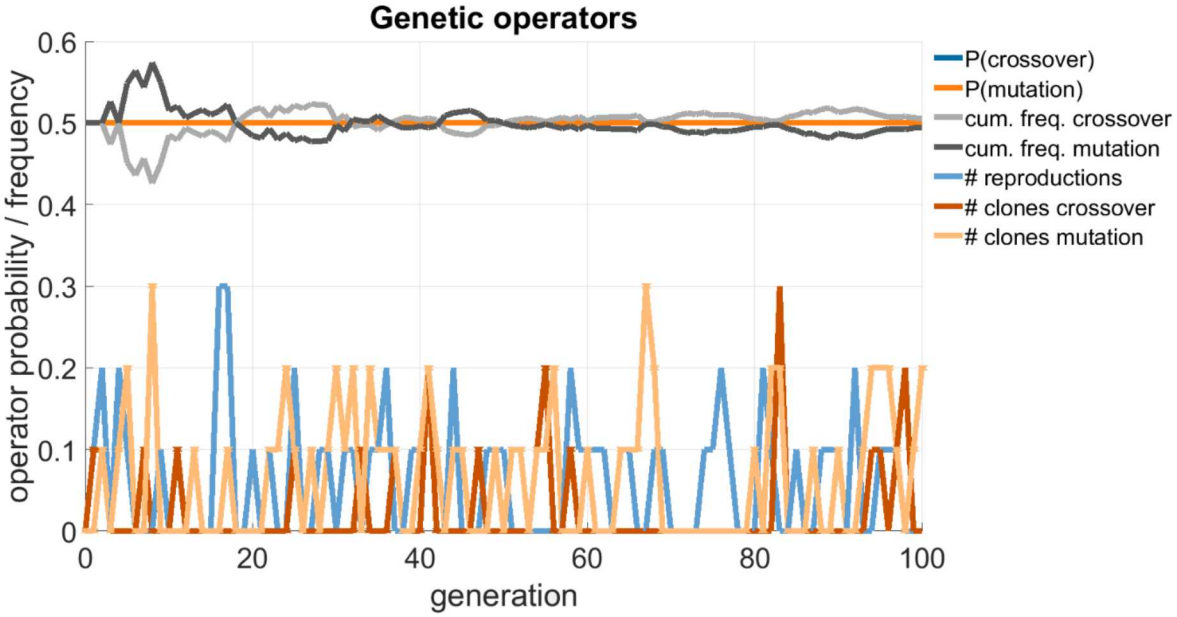


**Figure 57:** Genetic Operators

Overall, this approach seems promising, although stopping conditions will likely be a challenge for more complex Boolean functions. We also have not explored how to combine the generated trees to implement more complex functions. However, we believe that genetic programming could be a powerful tool for diversity, in that each Boolean function can have unique stopping conditions if the desired output is correct. With this, each function could have its own unique tree representation.

## 6.5. Targeting Dangling Nodes

When we modify dangling nodes we are changing aspects of the circuit that do not impact the logical function of the circuit. We say that these aspects exist at a semantic level that is different from the functional or logical level that we are concerned about during functional verification and testing. In particular, trojan payloads consisting of dangling nodes typically modulate some analog behavior of the design, such as power consumption, in a predictable (to the attacker) fashion. By randomizing the logic associated with these nodes we can also randomize the analog characteristics (*unintentional semantics* of the design) that are of interest to the attacker, while preserving the functional behavior (the *intentional semantics*) of the design that are of interest to us. This demonstrates that identifying and diversifying circuit behavior at different semantic levels can mitigate trojans that exist at one semantic level while preserving desired behavior at other semantic levels.

We present an approach for identifying nodes in a digital circuit that may belong to a leakage Trojan that is intended to transmit important information, such as the secret key for a cryptography algorithm, through a side channel. By randomizing the logic associated with these cones we can modify the behavior of these Trojans, reducing or nullifying the attacker's knowledge of the Trojan's functionality and consequently reducing its utility to the attacker. The same approach can also change the triggering mechanism for the Trojan, making it difficult for an attacker to activate and control the Trojan's behavior.

### *6.5.1. Approach*

A generalized view of a digital circuit is presented in Figure 58. There, we see that a circuit C has PIs, which are those signals that have no predecessors, and POs, which are those signals that have no successors. The circuit performs some function on the data presented at the PIs to transform it into the data outputs on the POs. We define dangling nodes to be nodes that have no POs in their transitive fanout cones. Since there is no path from the dangling nodes to the POs, then we can modify the logic functions implemented by the dangling nodes without impacting the intended functionality of C from the PIs to the POs. Amongst other possibilities, dangling nodes can result from poor coding, from abandoned circuitry within a design, or from circuits designed to leak information through side channels. Of these, we are particularly concerned with leakage Trojans that use power, timing, electromagnetic, or other types of side channels to leak important information, such as secret keys, from a device.

A diagram of one such leakage Trojan is shown in Figure 59. There, we see that the intended circuit functionality is an AES encryption. One of the PIs is the plaintext to be encrypted. The hardware Trojan consists of a trigger, which in this case is a comparison between the plaintext and a known constant. The attacker can provide the known plaintext at any convenient time, which will then trigger the leakage circuit. In this example, the leakage circuit consists of a shift register loaded with the AES secret key.

After the shift register is enabled it shifts right by one bit every 128 clock cycles. The least significant bit of the shift register is attached to a logic circuit that consumes power as a function of the least significant bit. By monitoring this power consumption the attacker can learn the value of the secret key.

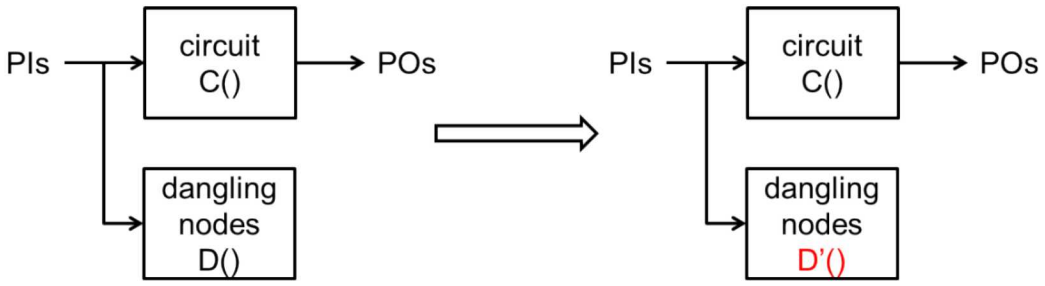


**Figure 58.** Dangling nodes are gates that do not appear on any path between the circuit's PIs and POs.

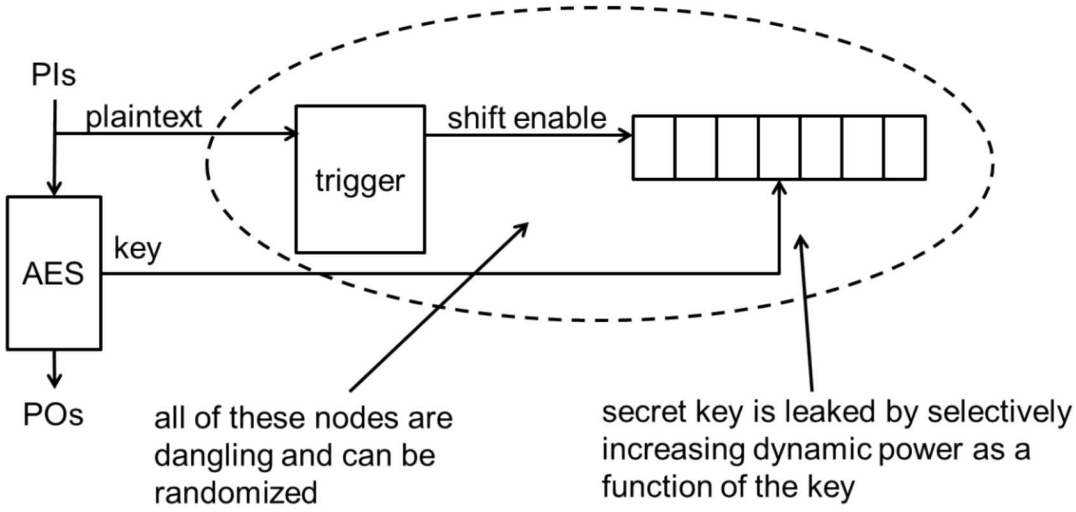


**Figure 59.** Example leakage Trojan in which the Trojan trigger and Trojan circuit consist entirely of dangling nodes

However, the Trojan trigger and the leakage circuit itself consist entirely of dangling nodes. There are no paths from any portion of these circuits to any of the POs in the host AES circuit. Consequently, we can identify these dangling nodes and randomize them to change the behavior of the Trojan and its trigger.

To identify the dangling nodes, we first collect a list of all of the nodes in the circuit. Then, we find the transitive fan-in cones of each of the POs. We permit these cones to be of any depth, allow them to cross latch boundaries, and follow them to the PIs. Along the way, we remove any node that appears in one of these fan-in cones from our complete list of nodes in the circuit. At the conclusion of this procedure we have a list of dangling nodes. Next, we can randomize the fan-out cones of these nodes. In our experiments we have considered three randomization approaches:

1. Gate addition [17]

2. Gate replacement [17]
3. Dynamic output inversion [18]

In gate addition we add one or more logic gates to a logic cone. These gates are then wired to existing gates within the logic cone. This process is illustrated in Figure 2. Gate replacement is a related technique in which we select one or more of the logic gates within a cone and replace them with different logic gates. This is depicted in Figure 5. In dynamic output inversion we selectively invert one or more of the outputs from the logic cone as a function of some control signals. In Figure 8 these control signals are the inputs to the logic cone. To see the impact of randomizing dangling nodes on leakage Trojans we consider an example. Figure 60 presents a simulation of an AES circuit that has been Trojanized according to the diagram in Figure 59. At the beginning of the simulation the trigger condition is satisfied. This activates the Trojan and afterwards a shift register holding the secret AES key shifts right by one bit every 128 clock cycles. The attacker can provide the trigger condition at will, and afterwards has a power side

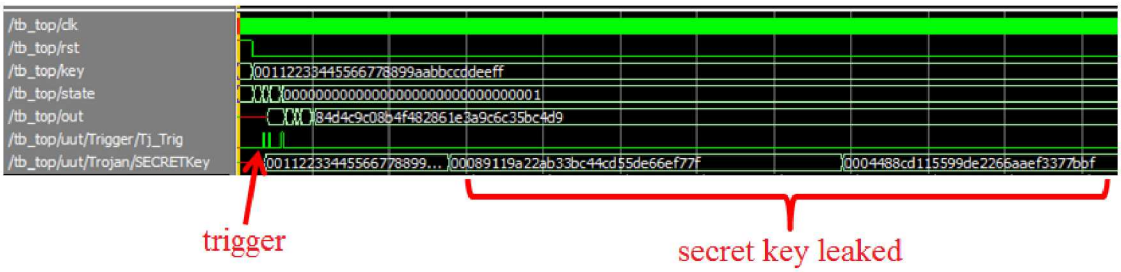


**Figure 60**. Simulation of a Trojanized AES circuit

channel that is a well-defined function of the secret key. Now consider Figure 61, which a simulation of the same circuit after randomizing the logic cones of the dangling nodes. We first notice that the trigger condition has changed, and so the Trojan is never triggered. This means that the attacker has lost the ability to selectively activate the Trojan with a known input plaintext. More importantly, the behavior of the leakage circuit has also changed. Now, rather than shifting by one bit every 128 cycles the shift register updates on every cycle. Furthermore, it does this whether or not the trigger condition has occurred. So, not only has the attacker lost the ability to control the Trojan but, after applying what the attacker thinks is the trigger condition the leakage circuit behaves the same as it did prior to the application of the trigger condition and this behavior is different from that expected by the attacker. Additionally, the shift register no longer holds the secret key, and rather than shifting by one bit on each update it instead changes between a small set of fixed values. Randomizing the logic cones of dangling nodes has broken the attacker's assumptions about how to trigger the Trojan and about the behavior of the Trojan after it is triggered. Finally, notice by comparing the 'out' signals in Figure 60 and Figure 61 that the intended behavior of the AES circuit has not changed.

### 6.5.2. *Results*

To study the overhead associated with this hardware Trojan mitigation we applied the technique to 21 different AES hardware Trojan benchmarks [19, 20]. Some of these benchmarks contain leakage Trojans, while others contain Trojans that modify the circuit behavior at the POs. All of the benchmarks, however, contain dangling nodes. This indicates that some of the dangling nodes exist within the AES implementation

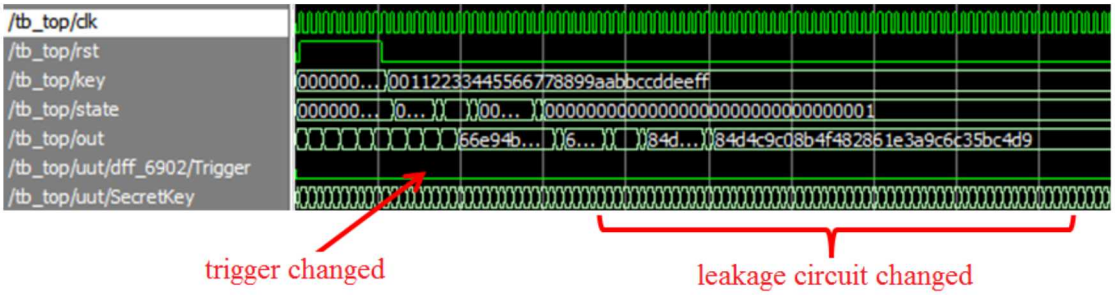


**Figure 61**. Simulation of a Trojanized AES circuit after randomizing the fan-out cones of dangling nodes

itself. For each of the benchmarks we identified all of the dangling nodes, and then randomized the logic cones of a percentage of them. Then, we synthesized and placed and routed the resulting circuits and compared their area and maximum operating frequency to that of the trojanized, but not randomized circuit. The results are presented as a function of the percentage of dangling nodes with randomized logic cones in Figure 62 and Figure 63. Neither the area nor the operating frequency is strongly impacted by the percentage of dangling nodes selected for randomization. The area overhead is nominal, with an average of about 0.1% area increase. The maximum operating frequency also increases by an average of about 2%, although there is more variability in this metric. Overall, the approach has only a modest impact on area and operating frequency. While we find that randomizing the logic cones of only 10% of the nodes is sufficient to disrupt the Trojan functionality, since the overhead is not a strong function of how many of the dangling nodes' logic cones is randomized it may be advisable to randomize the logic cones of all of the dangling nodes in the circuit.

Finally, we note that in many cases dangling nodes can be considered a portion of the "unintentional semantics" of a circuit in that they are not required for the circuit to perform its intended function. This is a demonstration of how modifying those unintentional semantics can impact undesirable portions of a circuit, such as hardware Trojans, while leaving the functionality of the intended circuit behavior, or the intentional semantics, intact. Other examples of incidental semantics include unspecified portions of a digital circuit, such as those that arise when a state machine is incompletely specified or when a logic function has some "don't care" conditions. Randomizing those unspecified portions of a circuit may also be an effective approach for disrupting some types of hardware Trojans, although in practice it would be

difficult to target Trojans in this way since, if a Trojan is implemented in unspecified portions of a design then those portions are no longer unspecified and cannot easily be targeted for randomization.

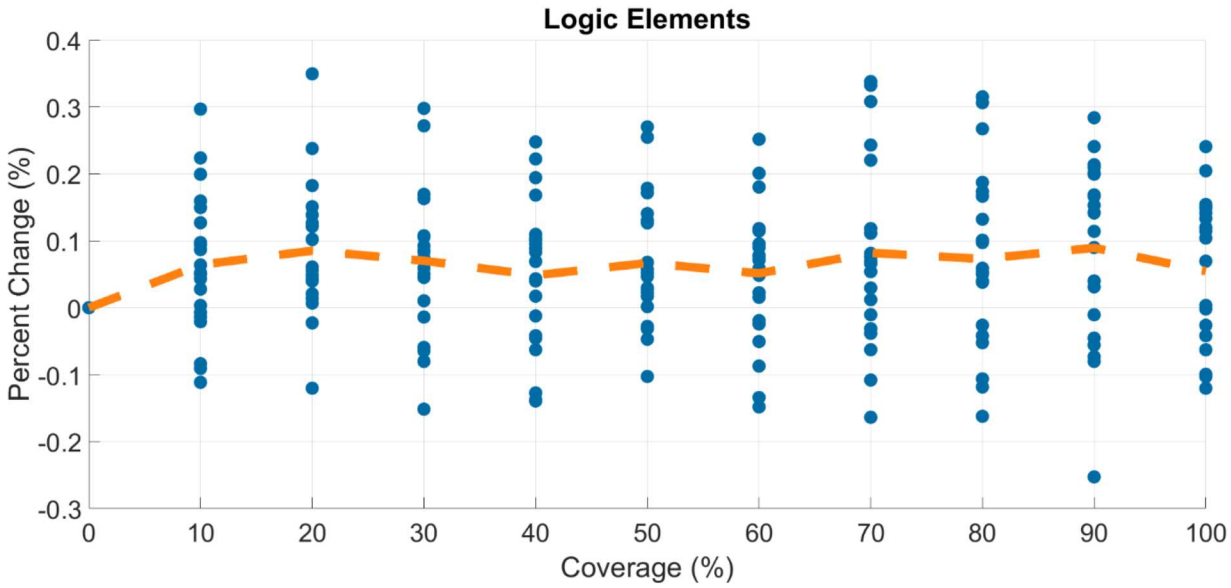


**Figure 62.** Area overhead from randomizing the logic cones of various percentages of the dangling nodes in a collection of AES Trojan benchmark circuits. Some of the Trojans are of the leakage type, while others are not.

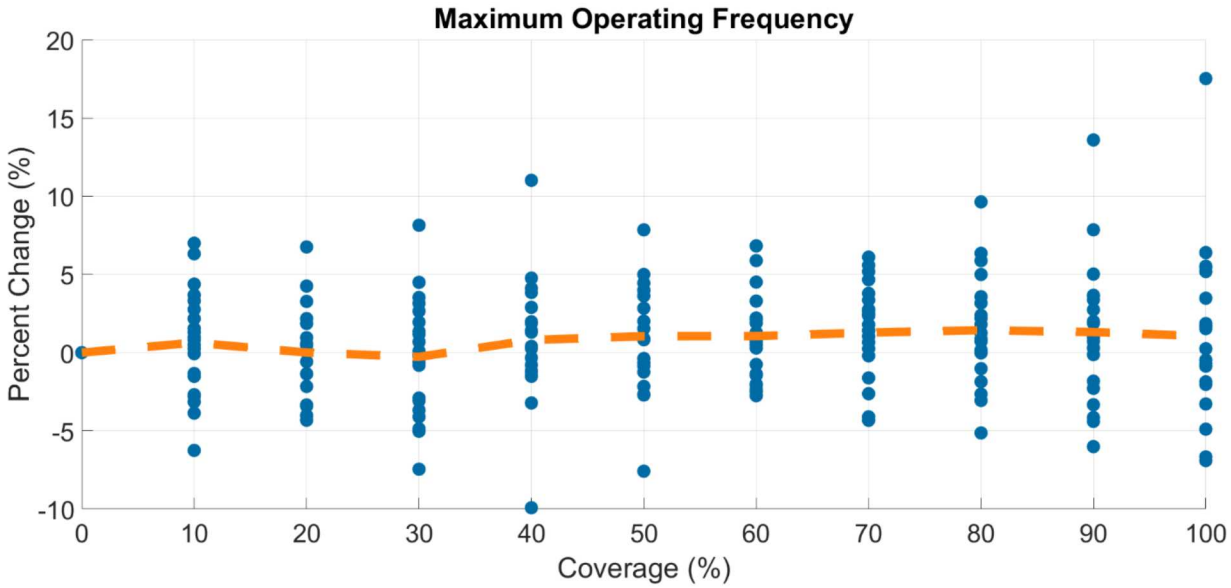


**Figure 63.** Operating frequency overhead from randomizing the logic cones of various percentages of the dangling nodes in a collection of AES Trojan benchmark circuits. Some of the Trojans are of the leakage type, while others are not.

This page left blank

# 7. FORMAL METHODS

While the complexity of many circuit designs prevents effective formal analysis of their properties, it is possible to apply formal methods to portions of circuits or to higher-level constructs, such as the diversity frameworks discussed and analyzed in Section 4. Here, we briefly describe applying the NuSMV model checker [59] to prove properties of a majority voter, a comparative redundancy circuit [60], and several variants of a comparator. These techniques are useful for formally verifying properties of security sensitive components, such as majority voters, or for analysis of select portions of designs. For example, if a structure of interest, such as a comparator, is located in a netlist then it can be targeted for formal verification. Recall that Section 6.3 introduced one approach for finding such structures.

We began by formally verifying the properties of a majority voter circuit. For this, we created a BLIF file containing a simple majority-3 voter mapped to our reduced gate library (NOT and two-input AND, OR, XOR, NAND, NOR, XNOR). We then converted this BLIF representation to an SMV model using ABC and some other open source tools [61,62]. For a majority voter with inputs $x_0$, $x_1$, $x_2$ and output q, NuSMV verified the property

```
x0==z0 AND x1==z0 OR x0==z0 AND x2==z0 OR x1==z0 AND x2==z0
```

indicating that the output always matches some two of the inputs.

Comparative redundancy, illustrated in Figure 64, is similar to simple majority voting except that it can potentially correct some multiple bit errors. Given inputs $x_0$, $x_1$, $x_2$ and output $z_0$, we set $z_0$ to $x_1$ if $x_0$ and $x_1$ are equal, and $z_0$ to $x_2$ otherwise. This circuit can potentially tolerate any single failure, in addition to the double failure of $x_0$ and $x_1$. In the single bit case the behavior is the same as a simple majority voter, because a simultaneous failure of both $x_0$ and $x_1$ will result in $x_0 = x_1$. However, in the multiple bit case if $x_0$ and $x_1$ fail differently the circuit can correct for this double error. As with the majority-3 voter, we created a BLIF file containing a comparative redundancy circuit

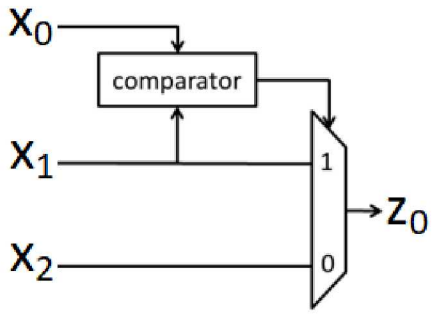


**Figure 64**. Comparative redundancy is an alternative to majority voting that can correct some double errors

mapped to our simple gate library and converted it to an SMV model. We then used NuSMV to verify the property

```
x0==z0 AND x1==z0 OR x0==z0 AND x2==z0 OR x1==z0 AND x2==z0
```

which is the same as the simple majority-3 voter in the single bit case. Note that we only performed the NuSMV analysis for the single-bit case in which the comparative redundancy circuit is logically equivalent to the simple majority-3 voter.

We also applied model checking to several comparator circuits. The first of these has two 128-bit inputs, named A and B, and three single bit outputs, AeqB, AgtB, and AltB, which are asserted if `A == B`, `A > B`, and `A < B`, respectively. We implemented this circuit in Verilog, and then converted it to BLIF format prior to performing the model checking. We used NuSMV to verify that the AeqB output is true if and only if `A == B`.

A second comparator circuit adds two additional 128-bit inputs, named key_in and key_const. In this circuit the comparison `A == B` is replaced with `(A XOR key_in) == (B XOR key_const)`. This models the RTL level circuit modification discussed in Section 6.3 and illustrated in Figure 46. We implemented this circuit in Verilog, and then converted it to BLIF format prior to performing the model checking. We then used NuSMV to prove that, given that `key_in == key_const`, then AeqB is true if and only if `A == B`. This property means that correct behavior of the AeqB output is maintained under the diversity strategy.

Next, we used NuSMV to prove properties of one of the trojan triggers from the trojanized AES benchmarks [26]. This particular trigger consists of a 128-bit comparison between input vector "state" and the 128-bit constant value x00112233445566778899AABBCCDDEEFF. If these two values are equal, then output "Tj_Trig" is true, and otherwise it is false. We converted the Verilog source for this circuit to BLIF format prior to the model checking. We then used NuSMV to find a counterexample to the assertion that Tj_Trig is always false. NuSMV found the counterexample `state == x00112233445566778899AABBCCDDEEFF`. In this case, NuSMV was able to find the single counterexample in a space of size $2^{128}$.

Finally, we modified the trojan trigger circuit by adding additional 128-bit inputs key_in and key_const, and modified the trigger condition to `(state ^ key_in) == (x00112233445566778899aabbccddeeff ^ key_const)`. This is the same modification that we previously applied to the 128-bit comparator circuit. We converted the Verilog source for this circuit to BLIF format prior to the model checking. We then used NuSMV to prove that, if

```
state == x00112233445566778899aabbccddeeff
```

then Tj_Trig is true if and only if `key_in == key_const`. This property means that the triggering input designed by the attacker can activate the Trojan if `key_in == key_const` but can never do so `if key_in != key_const`.

This page left blank

# 8. ADDITIONAL TROJAN PROTECTION CONCEPTS

In this section we briefly describe some additional approaches that may be useful for disrupting hardware trojans. While these techniques were identified during the course of the project, we prioritized further evaluation of the approaches described in the previous sections of these. Deeper exploration of these approaches is left to future work.

## 8.1. State Machine Tagging

Typically, control flow in state machines is not protected. This creates the possibility of faults causing state machines to transition between states in an unintended fashion, or, if there are fewer states than the total number that can be encoded by the state registers, for the machine to transition into an undefined state. Once in an undefined state the behavior of the circuit will also be undefined, and so state machines often include a "default" transition from any undefined state to some defined state, such as an initialization or reset state. State machines of this type are referred to as "safe" state machines. However, safe state machines only provide a recovery path to a known state from an undefined state. They do not prevent the state machine from improperly transitioning between two defined states. Such transitions can occur as a result of natural faults, or as a result of faults introduced by an attacker manipulating the voltage, clock, or temperature of a device. These faults can cause the state machine to incorrectly transition from one defined state to another defined state, from a defined state to an undefined state, or, in a variation of the first option, an attacker can introduce a new state with a transition to some important state and use the fault to transition to this newly created state [8]. In turn, this allows the attacker to break the control flow and transition into the important state without traversing the intended control path. These concerns are illustrated in Figure 65. In this example, there are three defined states, $S_0$, $S_1$, and $S_2$, with the intended control flow indicated by solid lines. When input signal d = 1 the machine advances through the states, and when d = 0 the machine remains in its current state. If we assume one-hot state encoding then there are three state registers. Since we can encode $2^3$=8 states with three registers, there are also 5 undefined states, which we illustrate as states $U_0$ to $U_4$. In a safe state machine implementation, any time the machine enters one of the state $U_0 \ldots U_4$ the machine will transition back to an initial state, say $S_0$. However, attackers can also manipulate the state machine and its transitions. We illustrate two possibilities with dashed lines. In the first manipulation the attacker introduces a new transition from state $S_0$ to state $S_2$ when d = 2. This allows the attacker to manipulate the intended control flow by applying a specific input to the state machine. A second modification is to add a transition from one of the undefined states, such as $U_0$, to one of the defined states, such as $S_2$. The attacker can then introduce a fault to cause the state machine to transition to $U_0$, and then the machine will enter state $S_2$.

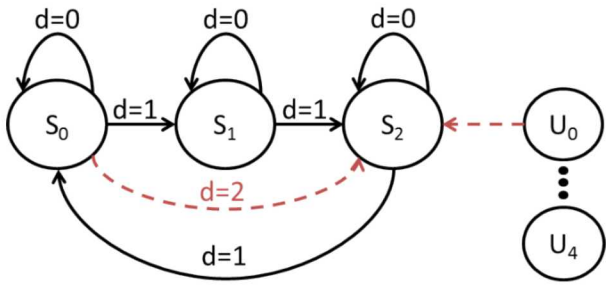


**Figure 65.** A state machine with three defined states and five undefined states

To address these vulnerabilities we can "tag" the transitions into states in some way so that the state machine knows what the previous state was. Then, if the previous state is not one of the expected states, the attacker transition is detected and it can be prevented by, for example, returning to the reset state. This is illustrated in Figure 66, where we have added additional transitions to state $S_0$ from states $S_1$ and $S_2$, and where we have updated the transition conditions from state $S_0$ to $S_1$, $S_1$ to $S_2$, and $S_2$ to $S_3$ to require that the previous state was an expected previous state. This protects against both of the unallowable transitions.

We can improve the protection by including checking other parameters in addition to the previous state. For example, we might also verify that the expected transition condition was satisfied. In the example of Figure 66 we would accomplish this by augmenting the transitions from $S_0$, $S_1$, and $S_2$ to $S_0$ to also return to $S_0$ if the value of the d signal on the clock cycle immediately preceding the transition was not 1. Implementation of these protections can be done by hand, automated as part of the synthesis process, or automatically added to netlists.

## 8.2. Decouple Side Channels from the Information of Interest

Several published trojan designs use side channels to leak secret information [22,51,52]. In the trojan depicted in Figure 67(a) some secret data, such as a cryptographic key, is loaded into a shift rotate register. The least significant bit (LSB) of this register is used

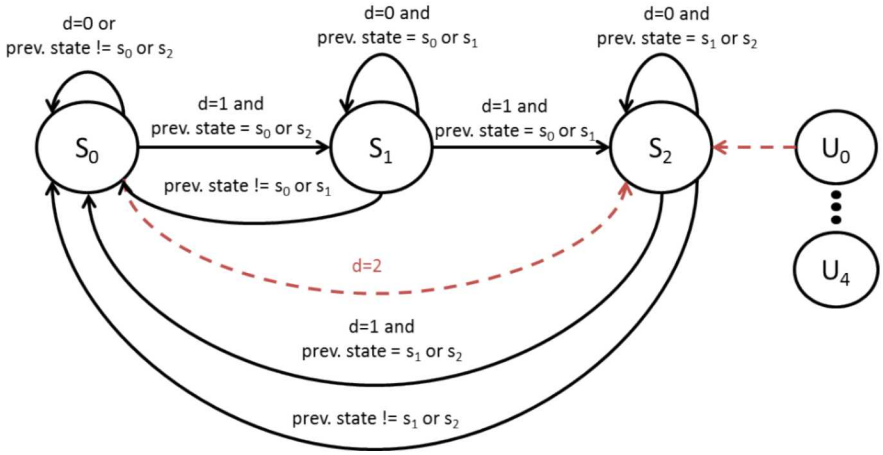


**Figure 66.** A state machine with enforced control flow

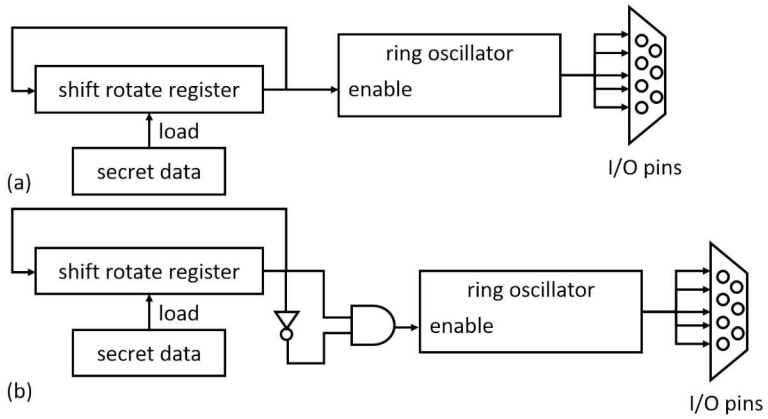


**Figure 67**. (a) A temperature side channel is created by rapidly charging and discharging the parasitic capacitances of an ICs I/O pins as a function of some secret data [51,52]. (b) The circuit can be modified to remove the dependence on the secret data

to enable a ring oscillator, which is then connected to I/O pins on the IC, which have large parasitic capacitances. When the LSB of the shift rotate register is a asserted the ring oscillator rapidly charges and discharges these capacitances, causing the IC's temperature to increase. When the least significant bit of the shift rotate register is deasserted these charge-discharge cycles stop, and the IC's temperature decreases. In Figure 68(a) a pseudo random number generator (PRNG) is used to encode the value of a secret data $k_0$, $k_1 \ldots k_{n-1}$ in the power consumption of a set of capacitors. An attacker than knows the initial value of the PRNG can measure the power consumption of the capacitors to acquire the encoded signal, and then use knowledge of the PRNG to decode the secret data. Other types of side channels can also be created. In all cases, the trojan will encode some secret data in an analog emission from the IC that the attacker can measure to retrieve the secret information. Decoupling the analog emission from the value of the secret data may disrupt the attacker's ability to extract the secret data from these emissions. The techniques described in Section 6 may be able to disrupt these types of trojans. For example, modifying the implementation of

the PRNG, which may prevent the attacker from knowing how the data is encoded, or by randomizing the ring oscillator, which may change the temperature gradients of the IC. Similarly, removing the shift rotate register, ring oscillator, or PRNG would eliminate the trojan side channels. The techniques described here are complimentary to those and seek to directly eliminate the side channel's dependence on the information of interest.

In Figure 67(b) we invert the LSB of the shift rotate register, and then enable the ring oscillator with a signal generated by the logical AND of the original and inverted shift rotate register LSBs. With the exception of transients the ring oscillator is always disabled, and the temperature dependence on the secret data has been eliminated. A logical OR could be used in place of the AND, but this would result in unnecessary power consumption and heating. Similarly, in Figure 68(b) rather than driving the capacitors with the XOR of the secret data and the PRNG output, we drive them with the logical AND of the output of the XOR and an inverted version of the XOR output. Other than transients this signal will always be low, so the power side channel will no longer leak the secret information. These examples show disablement of the side channels, although other implementations are possible. For example, in Figure 67 the ring oscillator and I/O pins could be duplicated, with the second ring oscillator enabled by the inverted LSB of the shift rotate register. The capacitors in Figure 68 could also be duplicated, with the newly added capacitances driven by the XOR of the PRNG outputs and inverted versions of the secret data. These implementations preserve the original circuit functionality, but modify the side channels by causing them to have the same behavior regardless of the value of the secret data. The side channels then become less useful to the attacker.

These approaches can preserve or change the functional behavior of the circuit, but will always modify its analog characteristics. If an implementation that modifies the functional behavior of the circuit, such as those illustrated in and , is chosen, then it is necessary to verify that the modifications do not impact the intended functionality of the circuit. Regardless of whether the functional behavior of the circuit is modified, if the analog characteristics of the circuit are important to the design, then these techniques must be used with care.

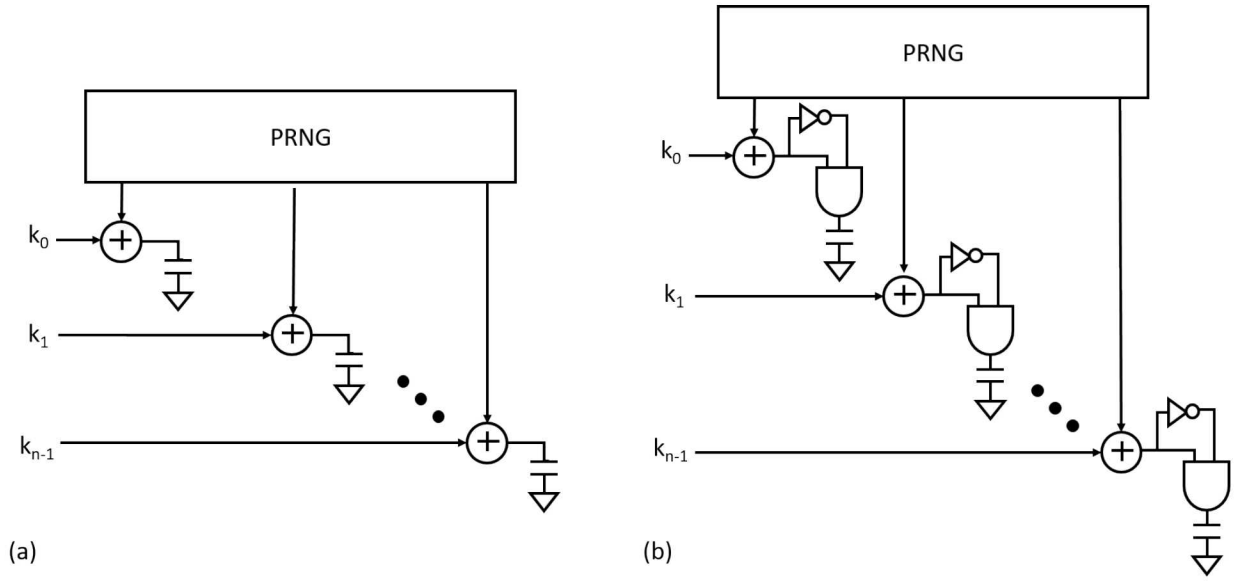


**Figure 68**. (a) A pseudo random number generator is used to encode secret data $k_0$, $k_1 \ldots k_{n-1}$ in the power consumption of a group of capacitors [22]. (b) The capacitor's power consumption can be decoupled from the secret data.

# 9. CONCLUSIONS

We have evaluated the theoretical effectiveness of diversity and moving target techniques, and have developed architectures for implementing these techniques in hardware. We have also identified and implemented techniques for automatically generating circuit variants with identical input-to-output behavior but diverse internal behavior. Diversity in the internal behavior results in diversity in analog characteristics, such as timing and power consumption, which can impact trojans that do not operate on the functional behavior of the circuit. For example, these approaches can impact trojans that create power or timing side channels to leak information. These approaches may also impact the ability of attackers to reverse engineer netlists, masks, or other post-design artifacts to successfully insert trojans.

We have demonstrated the effectiveness of several approaches for targeted modification of circuit designs to mitigate trojans in RTL and netlists. Such trojans may exist in third party IP, or they could be inserted by a malicious insider or design tools. These techniques address trojans inserted at different points in the lifecycle than those addressed by the diversity architectures. As such, the diversity architectures and targeted modifications approaches are complimentary and can be used together as part of a robust secure and trustworthy hardware development process. When used together, we recommend first applying the targeted modifications to increase

confidence in the netlist, and then implementing the resulting circuit with one of the diversity architectures.

## 9.1. Future Work

We prototyped several techniques for targeting trojans. Two of these target circuit structures that appear to be common in trojan triggers. In our prototypes, we targeted comparators, though we note that there are other structures that appear to be widely used in hardware trojans. Amongst these are shift registers, which can be used to build linear feedback shift registers and pseudo-random number generators that are then employed in trojan payloads to leak information from the circuit [22]. An example is illustrated in Figure 69(a). Another common structure is a wide AND (or wide NOR) gate that looks for a rare combination of input or internal signals to trigger the trojan, as shown in Figure 69(b). While it may not be feasible to selectively modify all of the wide AND (or wide NOR) gates within a design, if there are too many such gates then they can be prioritized by the probability of their outputs being asserted with lower probabilities being higher priority. Similarly, a common payload structure is to XOR some net in the circuit with the trigger so that the net is inverted when the trojan is activated, as depicted in Figure 69(c). We can selectively target XOR gates that have rare transitions on one of their inputs or those that are preceded by comparators, wide AND gates, or other trigger structures. A related payload structure, shown in Figure 69(d), is a multiplexor with a trojan payload as one of its inputs and the trigger as its select signal. Selectively targeting multiplexors with rare signal probabilities for at least one of the select inputs may permit targeting this type of payload. Additionally, both combinational and sequential variations of many trigger structures can be formulated, as illustrated in Figure 70. Somewhat more sophisticated trigger structures are shown in Figure 71 [48]. In this structure the trigger attempts to avoid detection by decomposing the rare trigger condition into a collection of combinational blocks, none of which has a control value that deviates significantly from the values seen in the benign portion of the host circuit. In Figure 71 this is accomplished by breaking the trigger into a cascade of combinational functions of 4 inputs.

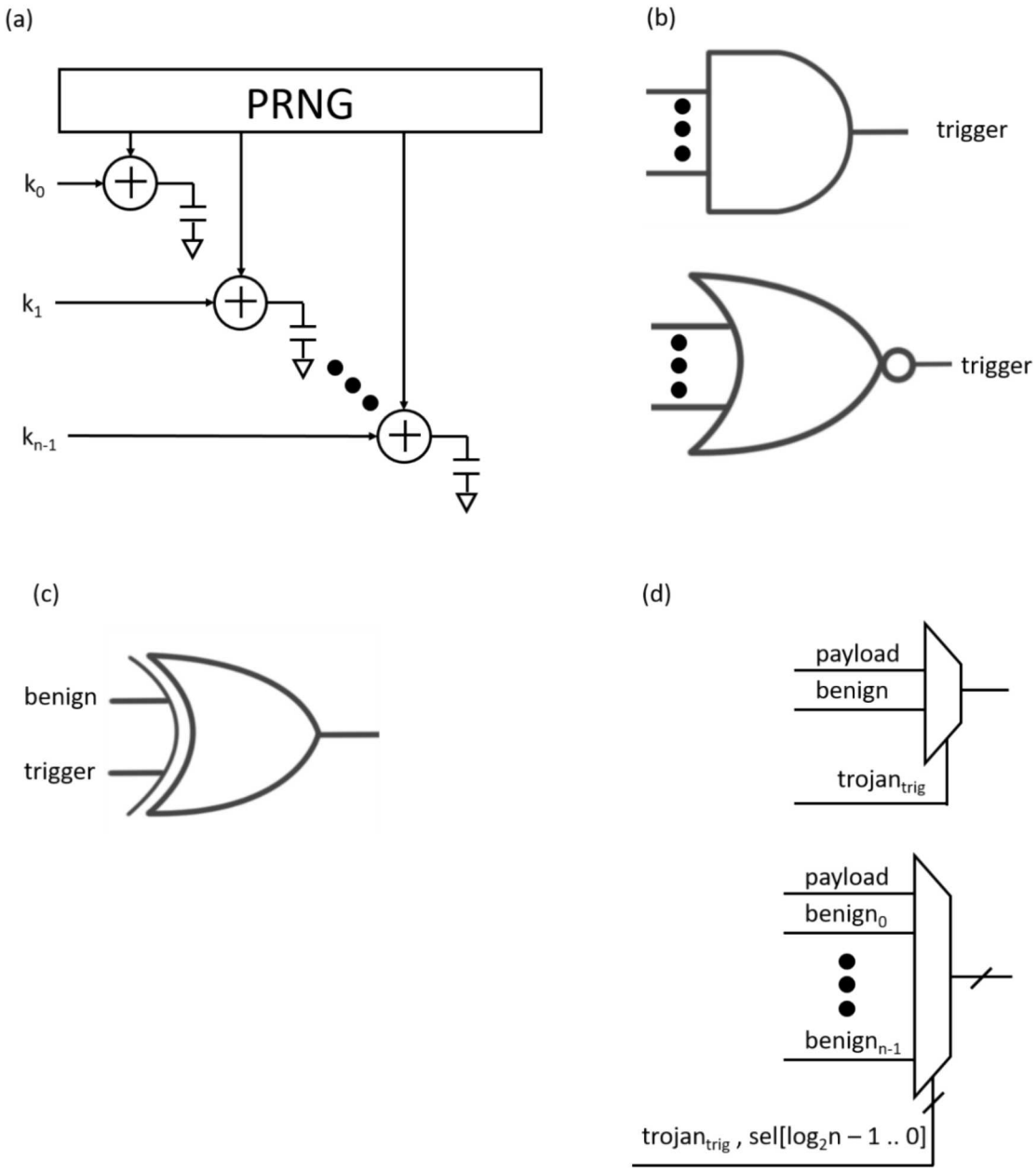


**Figure 69**. Additional trojan structures that can be targeted

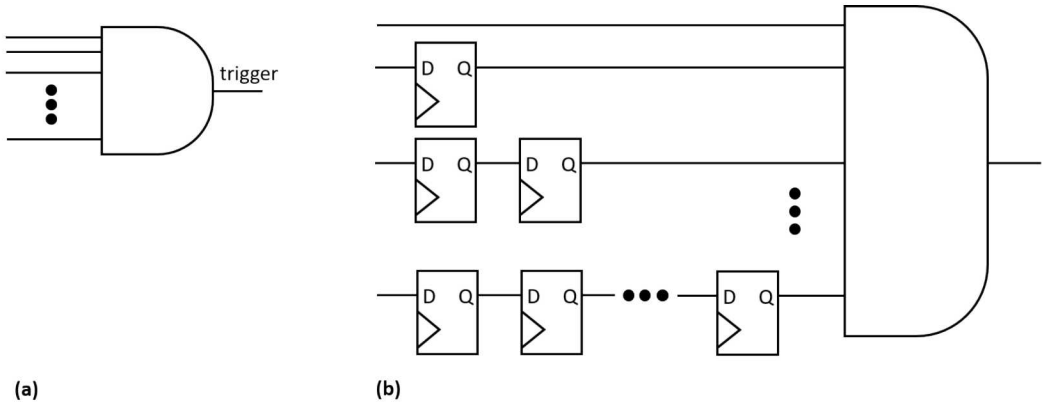


**Figure 70**. Combinational (a) and sequential (b) variations of a trigger structure

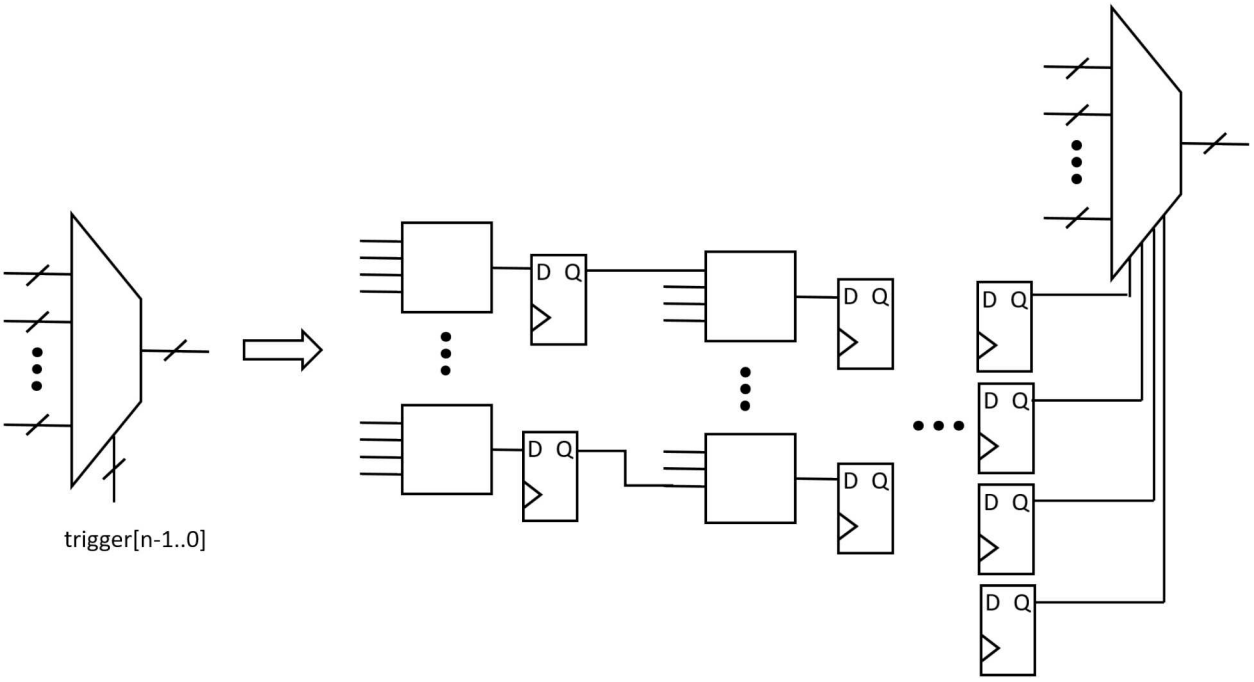


**Figure 71**. Rare trigger conditions can be decomposed into smaller chunks so that no individual combinational block has a control value so rare as to raise suspicion [48]. Here, an n-bit trigger is decomposed into a cascade of combinational blocks, each having four inputs. Eventually, a four bit trigger is produced.

This page left blank

## REFERENCES


1. Allan, Benjamin A., et al. "The Theory of Diversity and Redundancy in Information System Security: LDRD Final Report." (2010). SAND2010-7055.
2. Jean-Francois Monin. Understanding Formal Methods. Springer, 2003.
3. Robert C. Armstrong and Jackson R. Mayo. Leveraging complexity in software for cybersecurity. In Proc. 5th Cyber Security and Information Intelligence Research Workshop, 2009.
4. Daniel Williams., et al. Security through diversity. IEEE Security & Privacy, 7(1):26–33, 2009.
5. Turing, Alan Mathison. "On computable numbers, with an application to the Entscheidungsproblem." Proceedings of the London mathematical society 2.1 (1937): 230-265.
6. H. Salmani, M. Tehranipoor, and R. Karri, "On Design vulnerability analysis and trust benchmark development" IEEE Int. Conference on Computer Design (ICCD), 2013.
7. NIST, “Cryptographic Algorithm Validation Program CAVP Testing: Glock Ciphers”. https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program/Block-Ciphers. Accessed 06 July, 2018.
8. Nahiyan, Adib, et al. "AVFSM: a framework for identifying and mitigating vulnerabilities in FSMs." *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016.
9. M. Kwiatkowska, G. Norman, et al. PRISM 4.0: Verification of probabilistic real-time systems. In Proc. 23rd International Conference on Computer Aided Verification, pp. 585{591. 2011.
10. Poisson distribution. https://en.wikipedia.org/wiki/Poisson_distribution. Accessed 6 November 2017.
11. H. Shacham, M. Page, et al. On the effectiveness of address-space randomization. In Proc. 11th ACM Conference on Computer and Communications Security, pp. 298{307. 2004.
12. Jean-Fran¸cois Monin. Understanding Formal Methods. Springer, 2003.
13. Robert C. Armstrong and Jackson R. Mayo. Leveraging complexity in software for cybersecurity. In Proc. 5th Cyber Security and Information Intelligence Research Workshop, 2009.
14. Daniel Williams, Wei Hu, Jack W. Davidson, Jason D. Hiser, John C. Knight, and Anh Nguyen-Tuong. Security through diversity. IEEE Security & Privacy, 7(1):26–33, 2009.
15. Jackson R. Mayo, Benjamin A. Allan, Robert C. Armstrong, Geo↵rey C. Hulette, Todd M. Bauer, Jason R. Hamlet, Moses D. Schwartz, Jennifer Trasti, Benjamin G.

Davis, and Mitchell T. Martin. Leveraging complexity for unpredictable yet robust cyber systems: LDRD final report. Sandia Report SAND2013-8701, October 2013.

16. Brandon K. Eames, Alexander V. Outkin, Sarah Walsh, Jackson R. Mayo, Jason R. Hamlet, John M. Eldridge, Robert C. Armstrong, Mathew P. Napier, Gregory D. Wyss, Eric D. Vugrin, Michael L. Holmes. Fundamental Trust Analysis. Sandia Report, Sept. 2016

17. Porter, Roy, et al. "Dynamic Polymorphic Reconfiguration for anti-tamper circuits." 2009 International Conference on Field Programmable Logic and Applications. IEEE, 2009

18. Cady, Camdon R. Static and Dynamic Component Obfuscation on Reconfigurable Devices. No. AFIT/GE/ENG/10-06. Air Force Institute of Technology Wright-Patterson AFB OH School of Engineering and Management, 2010.

19. H. Salmani, M. Tehranipoor, and R. Karri, "On Design vulnerability analysis and trust benchmark development" IEEE Int. Conference on Computer Design (ICCD), 2013.

20. B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, M. Tehranipoor, “Benchmarking of Hardware Trojans and Maliciously Affected Circuits”, Journal of Hardware and Systems Security (HaSS), April 2017.

21. H. Salmani, M. Tehranipoor, and R. Karri, "On Design vulnerability analysis and trust benchmark development" IEEE Int. Conference on Computer Design (ICCD), 2013.

22. Lin, Lang, Wayne Burleson, and Christof Paar. "MOLES: malicious off-chip leakage enabled by side-channels." Proceedings of the 2009 international conference on computer-aided design. ACM, 2009.

23. Rajendran, Jeyavijayan, et al. "Towards a comprehensive and systematic classification of hardware trojans." Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on. IEEE, 2010.

24. Chakraborty, Rajat Subhra, Seetharam Narasimhan, and Swarup Bhunia. "Hardware Trojan: Threats and emerging solutions." High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International. IEEE, 2009.

25. Tehranipoor, Mohammad, and Farinaz Koushanfar. "A survey of hardware trojan taxonomy and detection." IEEE design & test of computers 27.1 (2010).

26. Trust-Hub, http://trust-hub.org/benchmarks/trojan, accessed July 16th, 2018.

27. Collection of Digital Design Benchmarks, http://ddd.fit.cvut.cz/prj/Benchmarks/ Accessed July 16, 2018.

28. Benchmark Circuits, http://www.pld.ttu.ee/~maksim/benchmarks/ Accessed July 16, 2018.

29. Mitra, Subhasish, Nirmal R. Saxena, and Edward J. McCluskey. "Common-mode failures in redundant VLSI systems: A survey." IEEE transactions on reliability 49.3 (2000): 285-295.

30. Avizienis, Algirdas, and J-C. Laprie. "Dependable computing: From concepts to design diversity." Proceedings of the IEEE74.5 (1986): 629-638.

31. Littlewood, Bev. "The impact of diversity upon common mode failures." Reliability Engineering & System Safety 51.1 (1996): 101-113.

32. Narasimhan, Seetharam, and Swarup Bhunia. "Hardware trojan detection." Introduction to Hardware Security and Trust. Springer, New York, NY, 2012. 339-364.

33. Jin, Yier, and Yiorgos Makris. "Hardware Trojan detection using path delay fingerprint." Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on. IEEE, 2008.

34. Waksman, Adam, Matthew Suozzo, and Simha Sethumadhavan. "FANCI: identification of stealthy malicious logic using boolean functional analysis." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013.

35. Becker, Georg T., et al. "Stealthy dopant-level hardware trojans." International Workshop on Cryptographic Hardware and Embedded Systems. Springer, Berlin, Heidelberg, 2013.

36. Huang, Shi-Yu, and Kwang-Ting Tim Cheng. Formal equivalence checking and design debugging. Vol. 12. Springer Science & Business Media, 2012.

37. Godefroid, Patrice, Michael Y. Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing." Queue 10.1 (2012): 20.

38. Porter, Roy, et al. "Dynamic Polymorphic Reconfiguration for anti-tamper circuits." 2009 International Conference on Field Programmable Logic and Applications. IEEE, 2009.

39. McDonald, Jeffrey T., et al. "Functional polymorphism for intellectual property protection." Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on. IEEE, 2016

40. McDonald, J. Todd, Yong Kim, and Daniel Koranek. "Deterministic circuit variation for anti-tamper applications." Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research. ACM, 2011.

41. McDonald, Jeffrey T., et al. "Evaluating component hiding techniques in circuit topologies." 2012 IEEE International Conference on Communications (ICC). IEEE, 2012.

42. Cady, Camdon R. Static and Dynamic Component Obfuscation on Reconfigurable Devices. No. AFIT/GE/ENG/10-06. Air Force Institute of Technology, Wright-Patterson AFB, OH, School of Engineering and Management, 2010.

43. Roy, Jarrod A., Farinaz Koushanfar, and Igor L. Markov. "EPIC: Ending piracy of integrated circuits." Proceedings of the conference on Design, automation and test in Europe. ACM, 2008.

44. Baumgarten, Alex, Akhilesh Tyagi, and Joseph Zambreno. "Preventing IC piracy using reconfigurable logic barriers." IEEE Design & Test of Computers 27.1 (2010).

45. Chakraborty, Rajat Subhra, and Swarup Bhunia. "Security against hardware Trojan attacks using key-based design obfuscation." Journal of Electronic Testing 27.6 (2011): 767-785.

46. P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W. Y. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik, "Reverse Engineering Digital Circuits Using Structural and Functional Analyses," IEEE Transactions on Emerging Topics in Computing, 2014.

47. S. Park and S.B. Akers, "An Efficient Method for Finding a Minimal Feedback Arc Set in Directed Graphs," ISCS, 1992.

48. J. Zhang, F. Yuan, and Q. Xu, "DeTrust: Defeating Hardware Trust Verification with Stealthy Implicitly-Triggered Hardware Trojans," CCS, 2014.

49. https://networkx.github.io/, accessed Aug. 8, 2018.

50. http://scikit-learn.org/stable/, accessed Aug. 8, 2018.

51. Karri, Ramesh, et al. "Trustworthy hardware: Identifying and classifying hardware trojans." Computer 43.10 (2010): 39-46.

52. Baumgarten A, Clausman M, Lindemann B, Steffen M, Trotter B, Zambreno J Embedded Systems Challenge. http://isis.poly.edu/vikram/iowa state.pdf

53. Pyverilog. https://pypi.org/project/pyverilog/. Accessed Aug. 30, 2018.

54. Silva, Sara, and Jonas Almeida. "GPLAB-a genetic programming toolbox for MATLAB." Proceedings of the Nordic MATLAB conference. 2003.

55. Searson, Dominic P., David E. Leahy, and Mark J. Willis. "GPTIPS: an open source genetic programming toolbox for multigene symbolic regression." Proceedings of the International multiconference of engineers and computer scientists. Vol. 1. Hong Kong: IMECS, 2010.

56. Almeida, M. A., Emerson Carlos Pedrino, and Maria C. Nicoletti. "A Genetically Programmable Hybrid Virtual Reconfigurable Architecture for Image Filtering Applications." Graphics, Patterns and Images (SIBGRAPI), 2016 29th SIBGRAPI Conference on. IEEE, 2016.

57. Poli, Riccardo, et al. A field guide to genetic programming. Lulu. com, 2008.

58. Banzhaf, Wolfgang, et al. Genetic programming: an introduction. Vol. 1. San Francisco: Morgan Kaufmann, 1998.

59. NuSMV. http://nusmv.fbk.eu/. Accessed Sept. 5, 2018

60. Philp, Kenneth W., and Norman D. Deans. "Comparative redundancy, an alternative to triple modular redundant system design." Microelectronics Reliability 37.4 (1997): 581-585.

61. Berkeley Logic Synthesis and Verification Group, “ABC: A System for Sequential Synthesis and Verification.” https://people.eecs.berkeley.edu/~alanmi/abc/ Accessed Sept. 5, 2018.

62. Eén, Niklas. “Temporal Induction Prover.” http://web.archive.org/web/20050526102314/http:/vlsicad.eecs.umich.edu/BK/Slots/cache/www.cs.chalmers.se/~een/Tip/ Accessed Sept. 5, 2018.

## DISTRIBUTION

| | | | |
|---|---|---|---|
| 1 | MS0161 | Legal Technology Transfer Center | 11500 |
| 1 | MS0359 | D. Chavez, LDRD Office | 1911 |
| 1 | MS0620 | Vivian Kammler | 5845 (electronic copy) |
| 1 | MS0671 | Jason Hamlet | 5827 (electronic copy) |
| 1 | MS0671 | Mitchell Martin | 5827 (electronic copy) |
| 1 | MS0671 | David Torres | 5827 (electronic copy) |
| 1 | MS0899 | Technical Library | 9536 (electronic copy) |
| 1 | MS9158 | Jackson Mayo | 8753 (electronic copy) |

Sandia National Laboratories