Spring 2020

HEURISTIC-BASED THREAT ANALYSIS OFREGISTER-TRANSFER-LEVEL HARDWARE DESIGNS

Wesley Layton Ellington
lellington@smu.edu

HEURISTIC-BASED THREAT ANALYSIS OF

REGISTER-TRANSFER-LEVEL HARDWARE DESIGNS

Approved by:

Jennifer L. Dworak
Department of Electrical and Computer Engineering
Dissertation Committee Chairperson

Eric C. Larson
Department of Computer Science

Sukumaran Nair
Department of Electrical and Computer Engineering

Alfred Crouch
Amida Technology Solutions

HEURISTIC-BASED THREAT ANALYSIS OF REGISTER-TRANSFER-LEVEL HARDWARE DESIGNS

A Thesis Presented to the Graduate Faculty of the

Lyle School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Master of Science

with a

Major in Computer Engineering

by

Wesley Layton Ellington

B.S., Computer Engineering, Southern Methodist University
B.S., Mathematics, Southern Methodist University

May 16, 2020

ACKNOWLEDGMENTS

I would like to thank my mentor and advisor, Dr. Jennifer Dworak for her guidance throughout this project. Her knowledge and support made this investigation possible. I would also like to thank Dr. Eric Larson, whose input was invaluable to the techniques developed.

I am also thankful for the support and experience of Alfred Crouch, industry expert and sponsor from Amida Technology Solutions, as well as for assistance from John Akin, design engineer at ASSET InterTech.

Ellington, Wesley Layton B.S., Computer Engineering, Southern Methodist University
B.S., Mathematics, Southern Methodist University

Heuristic-Based Threat Analysis of
Register-Transfer-Level Hardware Designs

Advisor: Jennifer L. Dworak

Master of Science conferred May 16, 2020

Dissertation completed April 14, 2020 Defense

The development of globalized semiconductor manufacturing processes and supply chains has lead to an increased interest in hardware security as new types of hardware based attacks, called hardware Trojans, are being observed in industrial and military electronics. To combat this, a technique was developed to help analyze hardware designs at the register-transfer-level (RTL) and locate points of interest within a design that might be vulnerable to attack. This method aims to eventually enable the creation of an end-to-end design hardening solution that analyzes existing designs and suggests countermeasures for potential Trojan attacks.

The method presented in this work uses a set of base heuristics to evaluate the signals and logic within an RTL design. These signals and their assignments are ranked according to different heuristic selection criteria to determine if they belong to one of three types for potential behavior modification Trojans. The first type aims to identify locations for highly destructive Trojans that could completely inhibit device function. The second corresponds to locations where an intermittent issue could be created, such as errors in calculation edge cases. The final type considers critical signals used to connect submodules within a design, potentially limiting communication or injecting false data into calculations if attacked. Once ranked, the top-most location for each of these three groups is reported in a ranked list. From this list, markers can be automatically placed in copies of the original design files to indicate where a potential Trojan attack could occur.

This approach was validated by using it to analyze two hardware designs. The results were

investigated manually, where high-level understanding of the designs was used to evaluate the potential implications of each location selected. This validation demonstrated that this automatic process can not only identify signals and locations similar to what a domain-expert might select for Trojan insertion manually but can also locate novel sites for potential Trojans that may not be apparent by an initial human evaluation.

TABLE OF CONTENTS

LIST OF FIGURES xi

LIST OF TABLES xii

CHAPTER

1. INTRODUCTION 1

2. BACKGROUND 4

2.1. Hardware Trojans 4

2.1.1. Types of Trojans 5

2.1.1.1. Physical Characteristics 5

2.1.1.2. Activation Characteristics 6

2.1.1.3. Action Characteristics 6

2.1.2. How Trojans are Inserted 7

2.2. Toward Better Hardware Security 9

2.2.1. The Achilles Tool Flow 9

3. PREVIOUS WORK 11

3.1. Trace Buffer Insertion 11

3.2. Signal Flow Checking 13

3.3. Trojan Placement and Location Analysis 15

3.4. Register Transfer Level Analysis 16

4. TOOL FLOW 19

4.1. Preprocessing 19

4.2. Module Extraction 21

4.2.1. Module Parameters and Local Parameters 21

4.2.2. Ports and Signals 22

4.2.3. Submodule Instances . . . 22
4.2.4. Assignments and Logic Structure . . . 22
4.2.5. Module Connectivity . . . 23
4.3. Scoring and Location Selection . . . 23
4.4. Design Level Ranking and Marker Insertion . . . 24
4.5. Limitations of the Tool . . . 25
4.5.1. Edits to Source . . . 25
4.5.2. Device IP and Auto-Generated Code . . . 26
5. SCORING AND LOCATION SELECTION . . . 28
5.1. Attacker Methodology and Location Types . . . 28
5.2. Process Overview . . . 29
5.3. Signal Scoring . . . 32
5.3.1. Base Signal Metrics . . . 32
5.3.1.1. Initial Weight . . . 33
5.3.1.2. Impact . . . 35
5.3.1.3. Susceptibility . . . 36
5.3.1.4. Controllability . . . 36
5.3.2. Signal Selection . . . 37
5.3.2.1. Highly Destructive Signals . . . 38
5.3.2.2. Intermittent Effect Signals . . . 38
5.3.2.3. Interconnect Signals . . . 38
5.4. Statement Scoring . . . 39
5.4.1. Base Statement Metrics . . . 40
5.4.1.1. Complexity . . . 40
5.4.1.2. Conditional Leaning . . . 41

5.4.1.3. Reachability . . . 44
5.4.2. Statement and Location Selection . . . 47
5.4.2.1. Highly Destructive Assignments . . . 47
5.4.2.2. Intermittent Effect Assignments . . . 48
5.4.2.3. Interconnect Selection . . . 48
5.5. Final Ranking . . . 48
5.5.1. Global Ranking . . . 49
5.5.2. Height Ranking . . . 49
6. EXPERIMENTS . . . 51
6.1. 16-Bit MIPS Processor . . . 51
6.1.1. Destructive Locations . . . 51
6.1.2. Intermittent Locations . . . 53
6.1.3. Interconnect Locations . . . 55
6.2. Navigation Controller . . . 55
6.2.1. Architecture . . . 56
6.2.2. Hand Placed Trojans . . . 58
6.2.3. Tool-Identified Trojan Locations . . . 59
6.2.3.1. Destructive Locations . . . 60
6.2.3.2. Intermittent Locations . . . 61
6.2.3.3. Interconnect Locations . . . 62
6.3. Observations . . . 63
7. CONCLUSION . . . 66
7.1. Design Strategies . . . 66
7.1.1. Least Privilege Policy . . . 66
7.1.2. Interconnect Vulnerabilities . . . 67

7.1.3. Standardization 68
7.2. Closing Thoughts 68
APPENDIX A 70
A.1. 16-Bit MIPS Location Rankings 70
A.2. User Inserted Navigation Controller Trojans 72
A.3. Navigation Controller Location Rankings 72
BIBLIOGRAPHY 76

LIST OF FIGURES

Figure Page

2.1. The Achilles Tool Flow 10

4.1. Automatic Analysis Processing and Analysis Steps. 20

5.1. Scoring and Selection Process 30

5.2. Example logical graph of simple conditional code with Conditional Leaning and Reachability scores. 46

6.1. Navigation controller high-level block diagram. 57

LIST OF TABLES

Table Page

5.1. Values used for calculation of $a_{initial}$ 34

5.2. Example Initial score values. 34

5.3. Criteria used for signal selection by type. 39

5.4. Examples of operator complexity estimation functions based on bit width (n). 41

5.5. Outcome bias of a 2-input AND gate with equal probability inputs with three zeros and one one. 42

5.6. Operators in Conditionals with bias toward 1 or 0. 43

6.1. Overview of unique locations in 16-Bit MIPS Processor 52

6.2. Overview of unique locations in Navigation Controller 59

A.1. Destructive Locations for 16-Bit MIPS Processor 70

A.2. Intermittent Locations for 16-Bit MIPS Processor 71

A.3. Interconnect Locations for 16-Bit MIPS Processor 71

A.4. Behavior Modification Trojans for Navigation Controller 72

A.5. Destructive Locations for Navigational Controller 73

A.6. Intermittent Locations for Navigational Controller 74

A.7. Interconnect Locations for Navigational Controller 75

In the hopes that this project helps create a safer world for everyone, this work is dedicated to my parents, Glenn and Kathrin, and to my siblings, Taylor and Alyx. I would like to thank my loving partner Laura for her support and patience with my pacing about the apartment during the COVID-19 quarantine.

Chapter 1
INTRODUCTION

As the design and manufacture of complex semiconductor devices has become more decentralized, new issues regarding device hardware security have come to light. Many companies now rely on components drawn from multiple sources of third-party intellectual property and large software design packages, as well as a globalized set of manufacturing facilities and supply chains. With each of these comes the opportunity for tampering by an outside or inside source, be it for profit by selling low quality or fake components, or by an entity wishing to do harm to those using a specific component. While most bad hardware discovered is the byproduct of test escapes, design errors, and counterfeiting, some hardware has been discovered to contain malicious components hidden in the design, called hardware Trojans. These hardware Trojans create new challenges for the field of cybersecurity and the hardware security sub-domain. As a result of some recent discoveries and increased public interest in hardware security, the last decade has seen an explosion of interest in hardware Trojans, how they are inserted, who is inserting them, and how to defend against them.

The main concern of this investigation is hardware Trojans and where they can be placed within an existing design. Hardware Trojans differ from the other categories of bad hardware components in that they are intentionally malicious and seek to enable some sort of external attack by an outside source against the component into which they are inserted. While the other types of bad hardware create danger through lack of quality control, underspecification, or reliability issues, the existence of Trojans are indicative that an adversarial entity wishes to do harm to some targeted device and its users. This means that an adversary has found a point of entry that exists in the design or manufacturing stages for a device and may affect a large number, if not all, of the devices manufactured. For this reason, hardware Trojans are sometimes called *genetic*, in that they are built into a design and become inherent to it after manufacture. This makes it necessary that a wide variety

different of kinds of Trojans and how Trojans are created be studied and understood so that appropriate defenses and countermeasures can be created.

This is an unsolved area of research, as new threats emerge constantly. It has become clear in recent years that hardware security is now a necessary component in electronic design methodologies. Thus, in an effort to combat this growing number of hardware Trojans, an end-to-end approach is being developed to harden pristine (Trojan-free) designs to Trojan insertion as well as create countermeasures to help detect and rectify Trojan activity in an otherwise functioning circuit. To create this end-to-end approach, it is necessary that Trojans specific to the design being evaluated are considered on a case-by-case basis. To this end, a method to create new Trojans for insertion into each pristine design is necessary, so that Trojan effects can be studied and countermeasures can be created and inserted into the hardened design automatically. The goal of the work presented in this discussion was to create a method that could automatically identify locations in which an attacker would be interested in placing certain types of hardware Trojan in a pristine circuit. With easily available examples to evaluate, a closed loop tool-chain can later evaluate potential Trojans at these sites, simulate their effects, and develop countermeasures to these Trojans accordingly. While Trojans can be inserted in a variety of ways, alterations at the register-transfer-level (RTL) will be the primary consideration in this discussion.

The method created here relies on two sets of base metrics, one for ranking signals and one for ranking lines of register-transfer-level code (locations), that are used to select optimal sets of signals and locations from each module within the design. These heuristics are derived from important characteristics of the RTL code being studied, such as connectivity information and high-level conditional structures. These characteristics were identified due to their use in instances where researchers inserted Trojans by hand for study. While useful for small experiments, hand insertion takes time and does not allow for large numbers of Trojans to be evaluated within a design easily. This automatic method requires no prior knowledge about the design other than its RTL code, and thus it allows for design level analysis with no user interaction and is completely agnostic to the intended functionality of the design in question.

The remainder of this discussion will be organized into the following chapters: First,

some background information on hardware Trojans, how Trojans are classified, and how this future product aims to help defend against them will be provided in Chapter 2. Then, to better ground the discussion, information about previous work and the state-of-the-art for certain topics relevant to this investigation will be given in Chapter 3. From lessons learned in previous work, a software tool was constructed to help identify potential Trojan locations automatically. The details of this tool's processing steps are described in Chapter 4, and the heuristic scoring mechanisms used to rank locations are described in Chapter 5. To verify the tool's functionality, analysis was conducted on two Verilog designs, and the results are presented in Chapter 6. Finally, in Chapter 7, several recommendations for future hardware designs are made based on observations from this investigation alongside some closing thoughts about the tool's performance.

Chapter 2
BACKGROUND

2.1 Hardware Trojans

As the semiconductor industry has become more widespread, both in terms of application and geographic distribution, serious concerns about device security have emerged. Many of these concerns come from that fact that all design and manufacturing steps are not managed by the same entities, as there are many parties involved in the global supply chains. This indicates the potential for attacks from untrusted fabrication facilities or intellectual property (IP) vendors, as well as a large insider threat from compromised individuals within trusted entities. Since the dawn of the system-on-a-chip (SoC) era, devices have become more and more complex, making the process of functionality testing much more difficult. This is true even before one considers the complexity of verifying that no malicious elements have been added to the design. With more mission and life critical systems being built using these complex designs, it is necessary that work be carried out to investigate how designs can be exploited and what countermeasures can be taken.

In 2008, a landmark publication from the *IEEE Spectrum* titled "Hunt for the Kill Switch" brought many of these fears to the surface [1]. This analysis was by no means the first publication on the topic, but it is seen by some as the first general discussion of the issue for a wide audience. Its analysis, although somewhat surface level, discusses what many speculate to be one of the largest instances of electronic warfare so far and outlines many of the concerns the military and commercial entities have about the modern semiconductor industry. These concerns reach every area of the engineering space, as topics related to the logistics of manufacture, hardware choices, supply-chain details, software, access control, and testing practices are all issues that could have a large impact on the trust that can be placed in a chip. For this reason, several programs, such as Defense Advanced Research

Projects Agency's (DARPA) now concluded "Trust in IC" program, and new fields of research will continue to help study hardware vulnerabilities. Work to ensure secure electronics for commercial and military use is still on-going over a decade after this initial publication.

2.1.1 Types of Trojans

Hardware Trojans come in many different "shapes" and sizes, as their functionality, triggering conditions, and implementations are all specific to the design into which they are inserted. That said, it is useful to have a sort of classification system to help describe Trojans and their components. One of the earliest publications on this was a taxonomy system described by researchers at the University of Connecticut and University of New Mexico in 2008 [2]. Since then, many updates have been provided to chart out hardware Trojans and how they can be understood in terms of their characteristics [3] [4]. Using the definitions in later work [4], Trojans can be easily classified using three high-level descriptors: *physical characteristics*, how a Trojan is implemented using the device it is inserted into, *activation characteristics*, sometimes called the *trigger*, which is how the Trojan activates itself, and *action characteristics*, sometimes called the *payload*, which is how the Trojan affects the function of the design. Each of these areas has various subcategories that are used to give a language to specify implementation and functionality details, allowing for high-level evaluation and discussion of Trojans. The following taxonomy is provided in [4].

2.1.1.1 Physical Characteristics

The *Physical Characteristics* of a Trojan seek to mainly explain how a Trojan manifests within a design in terms of its hardware implementation details, such as changes to masks, insertions of new logic components, or alterations to manufacturing processes. These characteristics say nothing of the logic or operation of the Trojan itself and seek only to describe how it was inserted. This set of criteria covers both Trojans inserted at the RTL and Trojans inserted at manufacturing, as it can describe changes to existing layout information or the addition of entirely new logic. Four criteria are used here:

1. *Distribution* - Trojan's location and reach within the design.

2. *Size* - Approximate number of elements changed to implement the Trojan.

3. *Structure* - Whether or not the layout of the design was recreated to introduce a Trojan.

4. *Type* - Specifies if the Trojan is functional, realized through altering logic, or parametric information. (realized through altering electrical characteristics).

2.1.1.2 Activation Characteristics

The *Activation Characteristics* are used to describe the conditions necessary for the Trojan to take effect during circuit operations. They are broken into two major categories:

1. *Internally Activated* - Trojans are triggered by internal signals and logic within the design. This can include combinations of signals or state machines, such as counters. In some situations, such as with some parametric Trojans, the Trojan is always activated.

2. *Externally Activated* - Trojans are triggered by an outside source using some sort of sensitization within the design. This can include hardware sensors and antennas to receive outside triggering signals.

2.1.1.3 Action Characteristics

The *Action Characteristics* describe the actual effect of a Trojan once it is activated. There are many ways that Trojans can disrupt a circuit, both directly (e.g. changing its functionality) or indirectly (e.g. causing it to heat up and thus slow down or consume more power). Several types of disruption can effect speed, but the mechanism by which this is done will depend on the implementation of the Trojan. Because there is a very large number of ways that a circuit can be tampered with, it is important to understand what sorts of tampering could cause certain issues. This category has three sub-categories:

1. *Transmit Information* - This includes Trojans that "leak" secure information, such as encryption keys, out to an unsecured location.

2. *Modify Specification* - This includes Trojans that change the operating mode of a circuit. For example, this can be done by changing path delay (thus circuit speed), causing a chip to consume too much power, or causing a chip generate too much heat.

3. *Modify Function* - This includes Trojans that alter the functionality of the the design in any way. This can be done by adding or removing digital logic, taking control, altering outputs, corrupting data, etc.

For the purpose of this investigation, Trojans in the *modify function* classification are of high interest. Trojans in this category are sometimes referred to using two sub-categories, Behavior Modifier Trojans and Reliability Impact Trojans [5]. *Behavior modifier* Trojans are usually implemented with payloads that intentionally change logic in a specific way to meet some goal, such as taking over control of a signal or executing a different operation than expected. *Reliability impact* Trojans seek mainly to degrade the operation of an existing function, causing data corruption or slow-down.

Trojans in the *modify function* category are often easiest to implement at the RTL. However, Trojans that leak information often depend on side-channels (physical methods of exporting data as radiation, power-draw, etc.) so evaluating methods for side-channel export would be incomplete at this level of abstraction. While some RTL information leaking Trojans have been proven using RTL elements such as multiplexers [6], this thesis does not take them into consideration. Much work has been conducted on how to detect alterations to digital logic (functional Trojans), as will be described in Chapter 3.

2.1.2 How Trojans are Inserted

Many potential locations for the insertion of hardware Trojans come from the complexity of the integrated circuit (IC) design and manufacturing process. Modern devices rely on a long chain of software packages, design IP (intellectual property available for purchase or reuse), standard logic cell libraries, and massive compilation tools to convert high-level descriptions of designs into synthesized RTL or layout information for manufacturing. Once this layout information is produced, a whole different chain of manufacturing steps is necessary to realize a design in a fabrication facility. For an attacker, this leaves many opportunities to insert malicious elements into the final product, as any stage where a change can be slipped into the product unnoticed is a possible avenue of attack.

Typically, designs are first conceptualized in terms of their high-level requirements and specifications. From there, they are elaborated through behavioral and register-transfer-level

(RTL) source code that is used to define how a design will be constructed from a functional point of view. Gate-level representations can then be created from the RTL source in a process called *logic synthesis* that converts high-abstraction statements into networks of logic gates, standard cells, or lookup tables (LUTS), sometimes called netlists. Finally, this information is converted into the form necessary to the type of technology being used to implement the design. This can include forms such as a bitstream used to program a field-programmable-gate-array (FPGA), or the physical layout data for transistors if the design is being manufactured as an application-specific-integrated-circuit (ASIC). This final process, sometimes referred to as *implementation*, is entirely dependent on technology specific IP and processes. Should the design be destined for full manufacturing, many more steps are necessary to make the lithography masks and define the process steps necessary to realize the design in a fabrication facility.

Trojans inserted at the register-transfer-level (RTL) pose a unique threat as any design created from compromised RTL source code will have the Trojan built into the netlist. This means that any methods of such detecting Trojans through standard gate-level automated testing practices after manufacturing will be defeated, because tests automatically generated for that design will likely take the Trojan's functionality into account, masking its effects. In fact, circuits that fail to correctly implement a Trojan (though some manufacturing defect) will be discarded as defective.

Additionally, designers often rely on third-party IP and external cell libraries to speed up the design process or physically implement designs. This allows designers access to vast resources of existing logic and hardware components for easy inclusion into their work. While useful, this does open the door for more potential security issues, as there is no known way to guarantee that third-party IP is safe and Trojan free, especially when implementations of elements are encrypted or obfuscated to protect secret IP details.

Even after a design is fully synthesized and converted into physical layout data, Trojans can be created through alteration of manufacturing masks or manufacturing processes. By changing the size or spacing of components, or altering how materials are doped or etched, small changes in behavior can be introduced through the introduction of unexpected resistance or capacitance. While slight, seemingly small modulations in electrical behaviors can

lead to changes in delay or can create well hidden Trojan effects.

The approach proposed in this work does not take into account how or at what implementation stage a potential Trojan is inserted into a design, thus ignoring its *physical characteristics* described in subsection 2.1.1.1. Rather, its focus is to locate places within the high-level description that could be dangerous through any attack at any point in the design or manufacturing process.

2.2 Toward Better Hardware Security

The US Government, US Military, and commercial electronic component producers have a vested interest in securing electronic products at every level. As a result, there is great interest in tools that can help to identify and manage hardware vulnerabilities at a large scale. While there are some tools for hardware security verification, such as that provided by Cadence [7] and other vendors, no true end-to-end hardware security solutions currently exist.

2.2.1 The Achilles Tool Flow

To fill the needs of entities aiming to secure their designs and other purchased products, the "Achilles " tool-chain was created in collaboration with Amida Technology Solutions, an industry vendor specializing in data analysis tools. The final tool-chain will seek to not only conduct preliminary analysis on a pristine design, but to automatically insert and emulate multiple types of potential hardware Trojans, place instruments to monitor circuit activity, and provide detection methods to find and classify Trojan behavior when it is present. This thesis focuses on the first two elements of this tool-chain. This is done by feeding in a *golden model*, or pristine circuit design, at the register-transfer-level (RTL) and allowing the completed tool to identify problem locations. The final tool will then evaluate the performance of proposed mitigation strategies in terms of additional overhead and potential Trojan coverage to make informed decisions when evaluating trade-offs. The tool will be comprised of five main components:

1. Golden Model Analysis - Identifying signals and locations within a design that are vulnerable to attack (this thesis) as well as searching for locations to place instruments to monitor circuit activity during operation.

2. Insertion - Placing hardware components for potential Trojans as well as proposed instruments to monitor Trojan behavior.

3. Hardware Emulation - Emulation of circuit behavior within a sandbox. This step allows for data collection for use in Machine Learning based Trojan detection algorithms.

4. Detection - Training of detection functionality for Trojan behaviors. This step seeks to classify Trojans as certain types as well as estimate impact when possible.

5. Countermeasure Analysis - Evaluation of how effective each instrument and detection method is for a given Trojan type. Information from this step can be fed back into step 2 to improve results in an iterative manner.

The work put forth in this thesis will serve as the basis of the first stage of the proposed "Achilles" tool-chain, giving a high-level threat surface analysis by identifying signals that are prone to attack as well as providing insight into what logic could be altered to facilitate a hardware Trojan. The signals identified by this method will be used as a starting point for instrument insertion as well as to give locations for potential Trojans to be inserted within an infected version of the the pristine design.

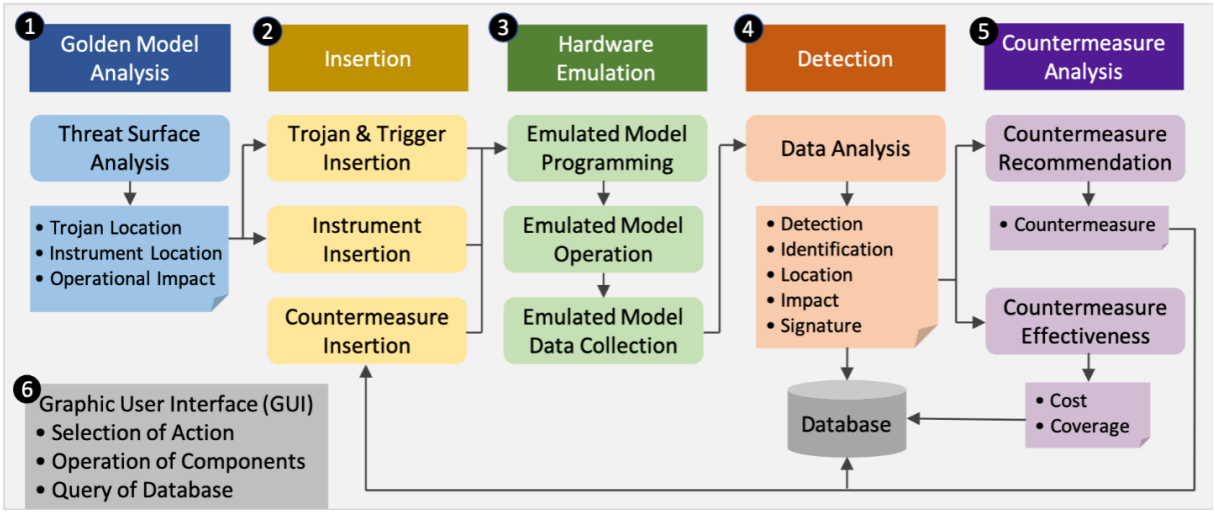

Figure 2.1: The Achilles Tool Flow

Chapter 3
PREVIOUS WORK

To better understand existing approaches to signal selection, hardware Trojan detection, and design analysis, several existing domains of research were investigated. While many of these problem areas are relatively new, they draw on extensive work done by the IC testing, verification, and security communities. There are many areas of research regarding testing and Trojan detection, but only topics that are applicable to functional circuit behavior, rather than parametric issues, will be discussed here.

3.1 Trace Buffer Insertion

Understanding which signals are critical within an design is a task that requires extensive knowledge of the design and its functionality. While there are myriad methods for verification in the pre-silicon stages of design, the coverage of these methods is not complete. This means that some issues are discovered after the design is spun into silicon, or never found at all. To debug a design once it is manufactured, designers must insert test points to expose internal signal values to external testing equipment. By choosing these test points wisely, validation engineers can reconstruct the internal state of a design after testing to help identify potential issues and trace back errors to their point of origin. While some circuits rely on test points that expose an internal signal to directly to an output, it is often beneficial to include testing hardware that can load prepared states and store important data for output. These testing hardware is usually implemented in one of two ways: scan cells and trace buffers. Scan cells are implemented as partial or complete replacements for existing registers within a design, allowing for small increases in area overhead overall. This overhead is justified because they are often crucial for manufacturing defect detection in digital circuits. Trace buffers are built from arrays of memory cells that are read like RAM elements. This often require significant increases in hardware to implement, but offer increased capacity to read out internal circuit

state at the expense of long data extraction times. As a result, it is necessary that testing and validation engineers intelligently select locations for this debug hardware as to minimize the overhead incurred by extra components. Because designs have thousands or millions of unique signals within them, this process often requires algorithmic optimization, as the search space is too large to be processed by human domain experts. As such, several approaches have sought to automate the scan cell and trace buffer insertion processes.

Work for validation and debug by Kanad Basu and Prabhat Mishra of the University of Florida [8] relies on the concept of *signal reconstruction*, that is, the process of solving for a circuit's internal state using an incomplete trace of a few signals within the component during testing after manufacture. If the collected signals are selected correctly, the process of reconstructing the signals within a design becomes a matter of simple Boolean algebra and simulation. A simple metric for this selection is called the *signal reconstruction ratio*, or SRR, that gives the ratio of signals that can be recomputed from trace data for a given location versus the number that cannot be reconstructed. In their work, Basu and Mishra provide an approach for estimating the controllability of signals between two flip-flops within a circuit. This is used as a heuristic to score connected flip-flops and form regions of signals that are deemed necessary for signal reconstruction. Experiments using this method offered improvement over previous trace buffer selection routines of the time [8].

Later work by Kamran Rahmani and Prahbat Mishra expanded on this problem when scan chain cells were to be placed in groups for output to a trace buffer [9]. Specifically, this means that during test, the full chain was not read, but that the values of specific cells were recorded to a trace buffer each on cycle. In this work, they present a greedy algorithm for selecting which cell should be saved at each clock cycle to help ensure better signal reconstruction by maximizing signal observability [9]. While the details of the algorithm are not necessary to this discussion, this method relies on a connectivity graph that helps contextualize signals during each cycle.

Other approaches for trace buffer signal selection have been attempted by Minn Li and Azadeh Davoodi using a new set of metrics that are used to directly relate the state of one signal to another. This is done by creating a list of reachable elements from any point while that point takes on a certain value [10]. This, coupled with other metrics created during

evaluation such as *restorability rate*, a measure of how easy it is to recreate a signal at a given point based on the current solution, allows for the estimation of the dependence of one signal's restorability on another. Together, these new metrics allow the algorithm to have iterative opportunities to expand or change the selection of target signals. This allows the overall SRR to be increased beyond what a single-pass evaluation might yield [10].

In general, research in this area has sought to inform decision-making on how to select signals based on their connectivity, as well as how signal dependence can be estimated from a high level version of the design. This is by no means an exhaustive review of these topics, as many other approaches, such as the Linear Programming approach in [11] have been attempted. Primarily, these concepts help give motivation for looking at high level connectivity graphs for estimating signal importance though metrics and iterative algorithms.

3.2 Signal Flow Checking

More recently, work in the security domain has explored topics related to information flow checking. These gate-level (designs in the form of networks of logic gates) approaches use concepts such as confidentiality and integrity (as defined in [12]) to assign flags to signals of special importance. These flags are then propagated through designs in a modified logic simulation to determine if certain signal paths cross and cause interference with protected signals or if information is leaked from a protection region.

A 2016 paper published by researchers at the University of Arkansas, the University of Florida, and the University of Connecticut presented a new technique called *structural checking* [13]. This method seeks to describe signals within a design at the RTL as *assets*, which are rough high-level classifications of signals, such as timing elements, state information, encryption keys, and other system level concepts. Signals are evaluated by their connectivity, where the assets associated with a signal are checked throughout its propagation. In this way, if an asset is leaked to an unexpected location or a signal with sensitive assets is altered by an other an unrelated signal, the tool can flag it as suspicious. For example, if an encryption key asset is found at an externally observable location that is not protected, the signals related to the leaked asset, such as the leaking output, are marked as suspicious [13]. While useful for the discovery of Trojan locations present in a design, this method does require

some hand labeling of asset types by the user as an input.

This work was expanded on by some of the same researchers from University of Florida [12]. In their approach, signals are evaluated through two main steps, *confidentiality verification*, which is the process of determining if information from one region of the design can be observed from an unsecured location, and *integrity verification*, which is the process of determining if a signal has altered or influenced an unrelated or unsecured signal [12]. Together, these two stages can not only identify if information is leaked or altered, but can also aide in the detection of Trojan trigger conditions. One major advantage to this approach is that it utilizes existing automated test pattern generation (ATPG) tools to help find both valid and suspicious observable points within a design. That said, this reliance on gate-level ATPG algorithms does limit its capabilities to working only on gate-level, synthesized designs.

Prior to this work, three other important methods of Trojan detection were proposed. The first is a tool called "JasperGold" created by Cadence. It is one of the few industry tools that considers hardware security from a design level [7]. Its focus is in verifying that selected memory elements and registers are not read from or altered illegally following formalized rules specified by a user. Because it is mainly tuned for memory related applications, it is not very useful for this investigation. As noted in work by [12], this technique also does not consider all signals within designs, and it may allow Trojans to go undetected.

Work from a joint project among the University of California San Diego, Northwestern Polytechnical University, and Tortuga Logic proposed a method called *Gate-Level Information-Flow Tracking*, which is often referred to as GLIFT [14]. While this paper is not the first application of GLIFT, this work uses it specifically to investigate the Trojan detection problem. This method works by tagging data bits as they propagate through a design, making note of when valid data path rules are violated or information is moved through an unexpected route. While this is useful in some situations, its reliance on formal methods does give false positives when valid and invalid paths are indistinguishable at a high level [12].

Finally, work from the University of Central Florida, Zhejiang University, and the University of Texas at Dallas [15] proposed a method that evaluates signal sensitivity within a design though an expansion of concept called *proof carrying code*. In essence, *proof carrying code* (PCC) seeks to provide a formal method of proving characteristics of a program

(or hardware design) by propagating special metadata throughout execution. This is done by creating a given "sensitivity" score to the pins of a module and propagating the scores during simulation in time with clock cycles according to rules generated from the RTL code. By verifying various theorems, or assertions about the state of circuit elements and their proof information within the circuit at any time step, the tool attempts to verify that no malicious activity is present. In this application, if provable assertions can be made about the security of signals that can be checked through simulation, the design is considered safe. While useful, this method is only as good as the accuracy of the provided sensitivity scores and is liable to attack by bypassing certain paths within the design that do not violate the assertions made to check them [12].

3.3 Trojan Placement and Location Analysis

As hardware security research has advanced, access to designs containing Trojans has become desirable, both for study and benchmarking. While Trojan designs found in the wild are interesting and provide insight into the mind of a real adversary, access to them is limited or nonexistent for most researchers and developers, so they are often a last test for new detection tools. In particular, some Trojans have been found in real-world applications, but in an effort to keep designs secure, the entities that discover them rarely release any details about these Trojans or their implications to user safety or security, making them difficult to study. This means that almost all Trojans available, such as the Trust-Hub database of Trojans [16], have been created by researchers and are thus limited in number and scope. To help expand the body of knowledge, researchers at NYU created a tool by the name of "TAINT: Tool for Automated INsertion of Trojans" [17]. This tool enables the insertion of Trojan look up tables (LUTs) or field programmable gate array (FPGA) cells into programmable media. While this tool is automated for the logical insertion of Trojan components into FPGA designs, it does not help the user identify specific locations that would be vulnerable to attack and only automates the physical insertion of the Trojan elements once a site is hand selected by a user. Regardless, it is a useful tool for creating new infected designs for evaluation of Trojan detection techniques.

Work by Hassan Salmani and Mark Tehranipoor has investigated physical circuit layout

information to determine if there are locations within a design that were vulnerable to Trojan insertion [18] in manufacturing steps. This work introduces the concepts of cell analysis and routing analysis. *Cell analysis* is the process of detecting locations of empty space that are larger than the smallest logic element in the technology that could house unintended logic elements. *Routing analysis* then evaluates locations where signal lines could be routed through metal layers to move information for Trojan functionality. This approach was able to identify interesting locations within synthesized designs that were susceptible to both delay and power based parametric Trojans. While useful for parametric Trojans, this approach does not offer the high-level abstraction required for this investigation.

3.4 Register Transfer Level Analysis

With only a few exceptions (see [13]), the Trojan identification problem has been approached from the gate-level or physical layout information. As a result, it was necessary to branch out further to understand how hardware source code can be digested at the register-transfer-level (RTL) and processed meaningfully without the need for a synthesis tool. Most of the pertinent work in RTL processing comes from the field of testing, as efforts to take design structure into account to speed up test time and optimize pattern length has created several interesting data structures for representing circuits.

In a publication from 1998 written by researchers Indradeep Ghosh, Anand Raghunthan, and Niraj Jha, control and data-flow graphs were used to provide insight into test generation [19]. This was done by first creating state diagrams of a circuit that represented not only its internal state transitions but also the flow of data though the design with each cycle. This structure is called a *control-data flow graph* (CDFG). This graph gives a cycle-wise depiction of data and operation flow within a circuit. This graph can also help understand more about the design's structure and thus it aids in making more decisions made while creating a test set. In this representation, circuits can be presented as special forms of Moore machines, where the transition values can be directly studied and included in the tests. From this analysis, the tool inserts multiplexers into the data- and control-paths that increase the observability of the circuit and allow for better logical control by test patterns. Not only does this identify locations for multiplexer insertion and generate test sets, but it

does so rather efficiently for data-flow heavy designs [19]. While this method does use the RTL code, it also requires a gate-level representation of the code, meaning that it is still reliant on synthesized versions of the design.

Later work by researchers at the Institute of Computing Technology in Beijing presented an approach for seeking out a special type of fault from the RTL description of a design [20]. This investigation was focused on the discovery of potential *transfer faults*, or instances where a value within an assignment or conditional does not properly propagate. This is done by first creating two important data structures: the *process controlling tree* (PCT) and the *data dependency graph* (DDG). The PCT is a high-level representation of the design's conditional elements and assignments in a tree structure that flows from the top of the design down through condition logic at internal nodes and ends with signal assignments at the leaves. The DDG is a more granular deconstruction of the RTL, as it represents the design in terms of smaller expressions and how higher level statements are constructed from operators and signals, rather than considering entire statements at a time. The DDG stores these operators and signals at the nodes within the graph and stores their interconnects as edges. Since this approach is seeking out transfer faults specifically, the edges flowing out from any node within the DDG can represent a location where a value fails to propagate (a potential location for a transfer fault). Together, these two structures are used in an ATPG algorithm named *X-Pulling* that was measured to greatly improve run-time when compared to another ATPG tool of the time.

More recently, these works and others were expanded on by Tobias Strauch to create a high level ATPG scheme. Simply put, this ATPG method uses an abstract fault model called a *gate inherent fault* (GIF) that has been expanded to consider function-agnostic faults from the view of a complex operation's primary outputs (GIF-PO) [21]. These fault models are highly abstract and do not rely on the implementation or logic values of any gate. This allows them to function on complex operations found at the RTL, such as arithmetic operators or conditional logic, rather than single gates and netlist connections. By focusing on path propagation rather than any specific logic value related fault, as is present in most other fault models, this model allows the user to evaluate high level design code at a high abstraction level without synthesis. The example provided in his work shows this method

being used on the Verilog "+" operator that synthesizes into a 64-bit adder, a feature not available in many ATPG methods. While this work is mostly exploratory and does not outperform more traditional ATPG methods, it is interesting in that it is based solely on the RTL analysis. Unfortunately, this approach did not gain much traction, so it was not expanded on further.

While many of these approaches help understand how the RTL source code of a design can be evaluated for testing purposes, there is little work on Trojan detection at this high abstraction level.

Chapter 4
TOOL FLOW

To enable more research and the development of an end-to-end hardware security solution, a new tool flow was created to automatically identify potential locations within a design that were vulnerable to tampering. This tool was constructed using some existing technologies for Verilog prepossessing and parsing, as well as custom written algorithms.

This tool is composed of four main components that flow from one to another as shown in Figure 4.1. Python was used as the main language for development, as it enables not only rapid prototyping and cross platform support, but also has easy access to extensive and robust text processing packages. This text processing functionality was necessary to process and manage much of the extensive Verilog standard. In addition, Python gives access to simple data structure serialization techniques that allow complex data representations to be written out to files easily. This functionality means that complex (and lengthy) processes can be broken down into sub-tasks that can be run separately, allowing for better code modularity.

4.1 Preprocessing

The first stage of the overall procedure is to process the raw Verilog design files into simpler code by evaluating preprocessing directives. This accomplishes two main goals. The first is to remove statements that are not relevant to the actual synthesized design, such as comments, test-bench code, macro definitions, or synthesis specific details. Second, it processes alternate code-blocks used to build multiple versions of a design from the same source code, ensuring only one build target is being processed at a time. If the input design was written properly, this will guarantee that no duplicate logic, module definitions, or unnecessary code is present after preprocessing.

In this tool flow, the Verilator package is used for its preprocessing functionality [22].

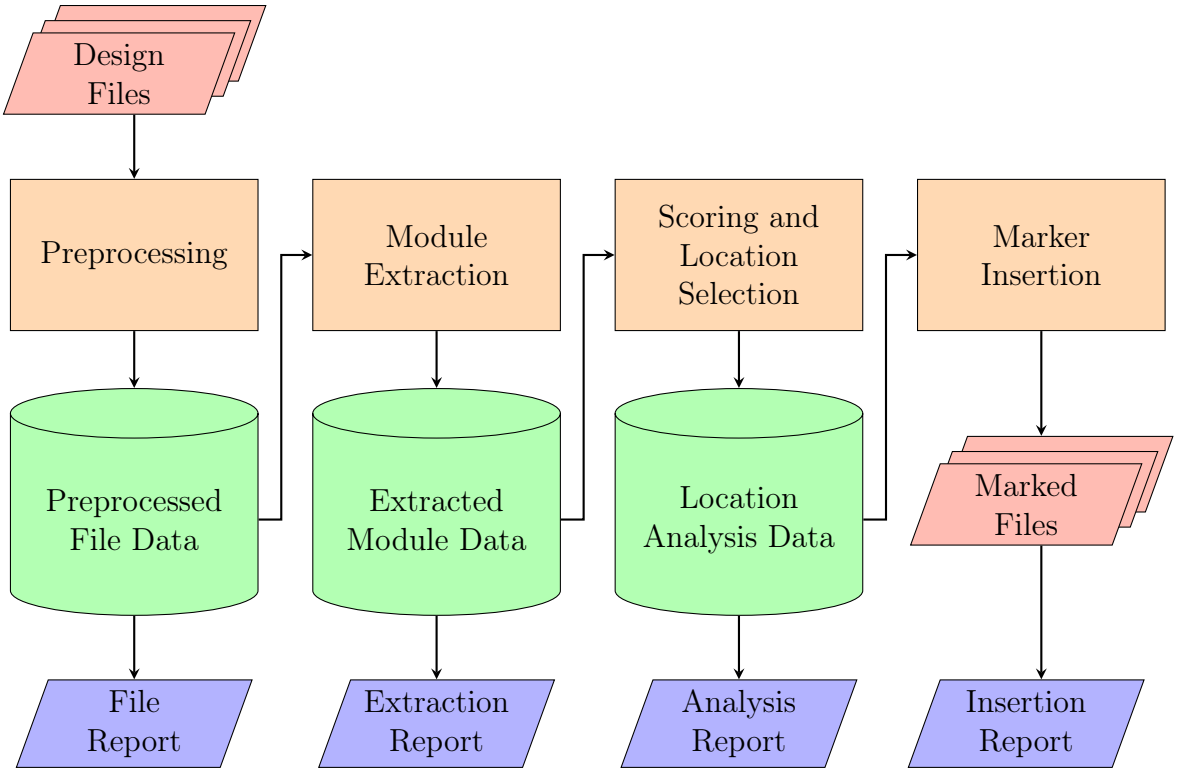

Figure 4.1: Automatic Analysis Processing and Analysis Steps.

While there are several freely available packages that offer some preprocessing functionality, Verilator can insert line directives (lines of Verliog code that record the original source file line number during preprocessing), allowing later stages of the tool to properly address lines from the original source files. Verilator is invoked by a Python wrapper that passes along relevant flags and directories to preprocess the design. Once preprocessed, the Python script routes the output of Verilator into a directory containing the entire design's preprocessed file equivalents in case they should they be desired by the user.

A secondary preprocessing step is then conducted to clean the preprocessed code further. This step removes certain unimportant or unprocessable structures such as attributes, generate statements, and function definitions. This is done to make later stages of processing easier to parse, as well as remove extraneous input that could confuse the extraction steps.

Once the source file processing is complete, the design is traversed once again to discover its design hierarchy. This is done using the Verilog-Perl package by Wilson Snyder [23],

which has inbuilt functionality to discover the hierarchy of a design and output it as a usable XML format. This output is read in by a Python wrapper and converted to a set of lookup dictionaries for finding module names and their submodule instances.

Finally, the preprocessed files are read again, this time searching for the line directives inserted by Verilator. During this pass, a mapping of original source file line to preprocessed file line is created so that later stages can access original source locations if necessary, as well as give exact locations for changes in the marker insertion step. This mapping will be kept along-side the text for the remainder of processing.

The results of these processes are loaded into a queryable data storage structure and written to a serialized file for access by the next stage.

4.2 Module Extraction

With preprocessed and cleaned text now available, the tool now locates the definition of each module found in the design hierarchy. To do this, the files discovered by Verilator are fed into a custom parsing system written in Python. Here the names of the modules are used within regular expression searches for module definition structures. Once detected, the text used to define a module is separated into its own container that will serve as the core of the extraction process.

Once all module definitions have been located, component extraction begins. This is done by systematically passing over the definition of the module and looking for components that are key to its functionality, while skipping over some of the remaining extraneous source text. Using a series of regular expression searches that each look for individual features, extracted information is stored within module specific data structures.

4.2.1 Module Parameters and Local Parameters

Verilog allows the user to define two types of parameters within a module definition. The first and more versatile module parameter allows the user to define variable sizes and values for code reuse purposes. This means that the user can create a general module definition that can be instantiated in various ways to account for variable sized busses, internal signal values, or hard-coded data. The second type is an internal parameter called a local parameter. This is simply a user-defined variable that is inserted during the elaboration step of synthesis.

These cannot be reassigned during instantiation and are useful mostly for readability and maintainability of the source code.

Both of these types of parameters are discovered and stored in a dictionary for later use. During later stages of processing, whenever the value of one of these parameter types is needed, the tool looks up its definition and inserts it in place of the parameter name. This steps allows some preprocessing tasks to partially replace some of the steps necessary during the typical Verilog elaboration process. This can be done without the need to fully elaborate a module or statement in its complete context, as a true synthesis engine would.

4.2.2 Ports and Signals

To understand module connectivity, both within the module and concerning its connections to the outside, it is necessary to compile a complete list of the module's ports (wires and registers used to connect module logic to external component's logic) as well as its internal signals. Again, this process is completed using regular expression searching for Verilog keywords and capturing data like signal type, name, size, and whether or not it is an input or output. The results of these searches are stored so that they can be searched for later when processing other kinds of statements and understanding signal connectivity.

4.2.3 Submodule Instances

Verilog code is inherently hierarchical in structure, as it often reuses many components multiple times across a design. To allow for simple inclusion of an already existing module definition within another module, that module definition can be 'instantiated' by assigning it an instance name, optional parameters, and port assignments. For the purposes of this tool, the type of module and the port assignments are the most important features to extract, as they will aid in understanding connectivity and signal influence later.

4.2.4 Assignments and Logic Structure

The final stage of parsing is discovering the two main components of design functionality: signal assignments and conditional statements. Because Verilog follows a recursive style of code blocks nested within one another, internal logic must be processed accordingly. Unlike submodule instances and signal definitions, the logic itself cannot be understood using simple

search techniques as were used for the previous module features, due to its differences in structure. This is by far the most complex process necessary for module extraction, as it requires not only iterative syntax searches used to find locate each kind of statement, but also requires that the recursive nesting of statements be preserved so that logic can evaluated correctly. While this application does create a simple list of signal assignments for the purposes of discovering connectivity, it is necessary to keep the logic of a module stored as a sort of simplified Abstract Syntax Tree (AST) [24] so that module logic can be understood within its proper context. This tree will serve as the starting point for most of the scoring techniques later, as it is the most direct representation of the actual module's functionality, representing both signal assignment and conditional logic. Once this tree is created from the source, it is stored along side the other extracted components. This is the final stage at which Verilog code of any form is read in by the tool, as a complete representation of the design's hierarchy, module definitions, and functionality has been created.

4.2.5 Module Connectivity

One final step is completed within the module extraction procedure. While it is not truly extracting anything from source, this stage traverses the logic trees of each module and builds a simple graph of how the signals within a module are interconnected. This is done somewhat naively, as it does not take into account assignment type (continuous or procedural). The goal of this step is mainly to understand what signals are attached to each other and in what direction. This will serve as a proxy to signal fan-out and fan-in later when scoring.

4.3 Scoring and Location Selection

Once all files are preprocessed and important features are extracted, the tool begins the location selection procedure. Briefly, this is done by calculating a set of metrics for each signal within a module and matching them with similarly scored logic statements. Since these metrics are dependent on relationships among signals, this is done in several passes in order to propagate both intermodular and intramodular relationships. This process is then repeated using a different set of metrics for the logical statements within that design, assigning them scores based on structure and complexity.

Once score assignment is complete, several subsets of each module's signals are selected for further investigation. These subsets are selected by taking an optimal sample based on pairs of metrics, depending on the type of location being investigated. The two criteria used for detection will dictate what kind of attacks this signal might be vulnerable to, as well as provide a relative ranking of the signals within each subset. Next, the statements within a module pertaining to one of these targeted signals will be evaluated based on their own metrics. To find locations that would be relevant to specific types of potential Trojan (elaborated more in Chapter 5), statements related to selected signals can be evaluated using relevant criteria, further pruning the search space. The results from this step are saved as a list of signals and locations (line numbers in modules) within a design that would be potential sites for attack.

A more detailed view of these metrics and selection criteria will be provided in Chapter 5.

4.4 Design Level Ranking and Marker Insertion

Once the lists of locations for Trojan insertion have been created for each module, a global ranking for each attack type can be constructed. This is done by collecting all of the locations of a certain Trojan type and storing their locations and final scores in a single sorted list. The top elements of this list can then be selected to give a global ranking of the most dangerous locations of each type for the entire design. To ensure more variation between the locations selected, locations from each module are selected in a round-robin style, forcing one location from each module to be selected before a second can be selected from the same module.

Additionally, a ranking based on the module depth can also be built. This allows for some weight to be given to how low or high a module is in the design hierarchy and treat modules more fairly. This is done by first sorting modules according to their height, where a height of zero corresponds to the lowest modules in a design (no submodules) to the the max height of the top module. The global ranking procedure is then run on each of these height groups. While this is not necessary to create a good ranking of locations in the design, it can help a designer better identify what types of problems show up in certain modules.

From experimental investigations, it was found that the global ranking contained most of the interesting locations found by the depth wise ranking. For this reason, the depth wise ranking is considered less important.

Once selected, each potential Trojan location is given a tag based on its type and relative ranking. The source files of each component of the design are then re-opened, and a comment marker containing information about the Trojan location, such as its ranking and type, are written into a copy of the source code. This can be done where all markers in each file are placed into one marked file copy, or a new copy of each file can be generated for each found location. Once complete, a manifest file is generated to give the user a central list of all of the selected locations and where they were placed into the design files.

4.5 Limitations of the Tool

Verilog is a robust and powerful language that allows for the creation of massive electronic designs. That said, it has become quite complex over time, as new functionality has been added constantly since its creation, and extensions like SystemVerilog only increase this complexity further. As a result, it is very difficult to implement a fully functional Verilog parser from scratch. While some open-source Verilog parsing tools exist, they did not meet the needs of this project. So, a custom parser was implemented to allow for the most important features of a Verilog design to be processed, while allowing unsupported structure and syntax to be ignored. This was done using a hybrid "soft parsing" approach using a regular expression matching technique applied in a systematic manner to generate a simplified form of abstract syntax tree (AST), similar to what one might expect from a formal parsing tool.

4.5.1 Edits to Source

In order to meet the limitations of the parser, some small alterations need to be made to designs being evaluated to compensate for complex or ambiguous structures. These necessary edits were intentionally kept minor, so as to not require changes to device logic (with one exception). The changes made for these purposes differ from the text scrubbing preprocessing step mentioned in section 4.1 in that they must be conducted by a user and are not automatic.

- Multiple Instances of a Module - Some designs make use of Verilog's ability to instantiate multiple modules of the same type using one statement. These must be unrolled

to give each submodule instance its own statement.

- Verilog Function Evaluation - While somewhat uncommon, Verilog gives the ability to define simple functions for calculating values during elaboration. For example, this can be used to calculate the size of address signals for buses dynamically. These calculations must be run by hand and the function calls related to them replaced with the static value the function would return during elaboration.

- Ambiguous nesting - Some uses of nested structures (such as IF and CASE statements) can cause issues due to their syntax. To resolve this, some statements with one line of code body may need 'begin' and 'end' keywords added to remove ambiguity.

- Expressions in Module Port Assignment - Verilog allows the user to assign module ports to complex expressions during instantiation. While useful, this is difficult to parse, so any ports assigned in this manner need to be connected to an intermediate signal that is assigned the value of the expression of the original port connection. This may slightly change the gate-level netlist, but does not change the functionality.

All of these issues could be remediated by expansion of the parsing component's functionality.

4.5.2 Device IP and Auto-Generated Code

Many modern designs rely on large libraries of third-party IP as well as automatically generated code from electronic-design-automation (EDA) tools. As a result, some functionality of the circuits being evaluated may be left out, as attempts to obfuscate IP implementations make them difficult to parse or some files may be missing from the RTL source entirely. In addition, since many tools will offer automatic generation of some components using customizable library modules, some implementations are not created by the tool until the elaboration and synthesis step. This means that some of the RTL source provided will not contain all of the necessary details to fully evaluate these modules. To manage these two issues, modules defined in either manner will be removed from the design hierarchy for processing by the tool and will be seen only as submodules within modules that instantiated them. This allows signals going in or out of modules to be considered with no knowledge

of submodule implementation. Usually, these types of modules are at the bottom of the hierarchy anyway, since they are often look-up-tables or implementations of some arithmetic operations that would have a small impact on the overall design, and thus would be unlikely to create a Trojan site that would appear in the upper ranking list. In some cases, however, more complex components, such as JTAG testing networks, are implemented in this way. Because JTAG interfaces are a known security issue, it will be important to consider these components more completely in the future. Future work seeks to expand the reach of the tool and bring more components of target designs into the analysis.

Chapter 5
SCORING AND LOCATION SELECTION

In order to select interesting signals and logic for potential Trojans, analysis of high-level Verilog constructs is necessary. To do this, a novel set of metrics was used not only to understand the signals and statements within each Verilog module, but to also inform what sorts of behavior modification Trojans could potentially be derived from them. A heuristic based approach was developed to score each signal and logic statement according to concepts of connectivity, type, size, and complexity.

5.1 Attacker Methodology and Location Types

To better understand how an attacker might go about manually inserting a Trojan, it was important to consider the decision making process that a human attacker would go trough when considering attacks against a design. The following was used as a model process for how an attacker might go about selecting locations for inserting Trojans in a general sense. This process only considers the placement of the payload, as trigger selection will likely depend on the outcome of the payload insertion process and is outside the scope of this thesis. That said, this process is not fixed, as any stages from those defined below could easily be changed depending on the design or individual in question.

1. The attacker studies high level functionality of device components.
2. The attacker selects a module (or set of modules) to target for attack.
3. The attacker determines what type of effect a Trojan would implement.
4. The attacker inspects signals related to desired effects and selects a target signal.
5. The attacker finds a point in the circuit at which to inject malicious logic.
6. The attacker determines what logic to insert to achieve the desired goal.

This work focuses mainly on steps three, four, and five. For the sake of this thesis, three types of potential Trojan location were defined to approximate some of the potential options an attacker may consider when considering what type of effect they desire in step three. These three types of potential Trojan location are described simply as *Destructive Locations*, locations that cause catastrophic effects on calculations, *Intermittent Locations*, locations that create hard to pinpoint errors or "glitches", and *Interconnect Locations*, locations that could be used to interrupt communication between two or more submodules. In practice, each location will be comprised of two components: a target signal and a target line of code (statement). These two pieces of information define a site within a design that could be a good site to insert a Trojan, and do so with enough specificity as to imply how a Trojan payload could be implemented. These locations and the criteria used to identify them will be described in more detail in later sections. While other types of locations are possible and likely useful for full analysis of a design, this investigation was limited to these three, as they lend themselves quite well to the study of Behavior Modifier Trojans specifically. Future work will expand this list to include even more types of locations.

Using these three classifications of locations as a starting point, scoring and selection processes were implemented to emulate the decisions made by an attacker in stages four and five. The first scoring process will be used to select signals that are thought to be important to a module's functionality, approximating the selection process of step four. From this, logic within a module can be evaluated to find locations that lend themselves to attacks associated with each type of location.

5.2 Process Overview

To implement an algorithmic approach to identifying sites that are in line with the types of locations described above, a process was developed to automatically analyze Verilog using a set of metrics built around how a human might seek out Trojan locations manually. This method relies heavily on a set of metrics defined to capture important features and structures in RTL code that are good locations to place a hardware Trojan. This process is best conceptualized through five stages of scoring, selection, and aggregation as depicted in Figure 5.1.

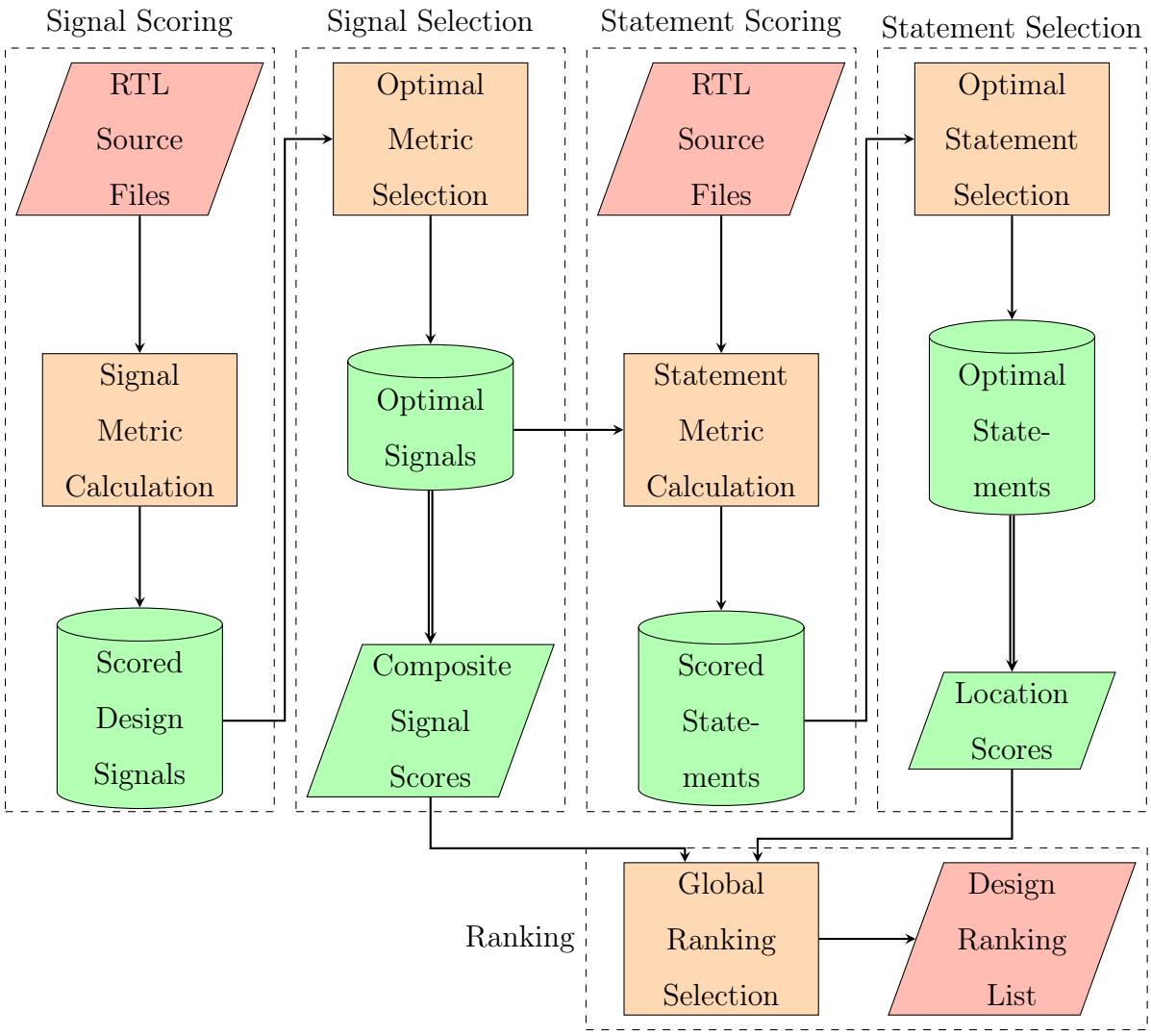

Figure 5.1: Scoring and Selection Process

In the first stage, called signal scoring, the preprocessed and extracted RTL source code is evaluated to find all signal lines, including wires, registers, and ports. These signals are then evaluated using a novel set of metrics that capture connectivity and structure information. These scores make it possible to evaluate the role of each signal within a module without knowing its intended purpose. There are three main metrics calculated for each signal. Put simply, these are *Impact*, a measurement of forward assignment (fan-out), *Susceptibility*, a measurement of backward dependency (fan-in), and *Controllability*, a measurement of how specifically targetable a signal's assignment is terms of conditional structures around it.

These metrics will be discussed in more detail in following sections.

The second stage, called signal selection, uses a set of selection criteria to identify optimal signals within a module for further analysis. This selection is done by considering the metric scores calculated in the first step in pairs by way of calculating a Pareto frontier using two metrics that characterize one of several types of potential Trojan location. This allows for an intelligent selection of signals within a module that will likely lend themselves to a specific type of attack, e.g. the metrics describing maximum *Impact* and maximum *Controllability* could be used to identify *Highly Destructive* signals. The selection process will create a subset of all signals in a module for each of the three types of location being investigated and report them to the next stage.

The third stage is quite similar to the first stage, but this scoring process focuses on the contents of the Verilog logic itself, calculating metrics based on the contents of each statement. These metrics are similar to the signal scoring metrics in the first stage, but they seek to capture functional information about the operators used to conduct calculations rather than signal connection information. To reduce the amount of work necessary to create a full ranking, only statements containing one of the signals selected in stage two will be considered in this scoring step. Two metrics are calculated here: *Complexity*, a measurement of how much gate level logic might be necessary to implement a line of RTL code, and *Reachability*, a measurement of how difficult it would be to reach specific lines of RTL during operation. The results of this stage will be passed to the final stage of selection.

The fourth stage, statement selection, will use the metrics from statement scoring to select specific locations (lines of Verilog source code). This process will also use a Pareto selection approach, where applicable (*Interconnect* locations do not use a Pareto selection), to balance the two metrics used to find locations lending themselves to the types of locations being investigated. At the end of this stage, a ranked list of potential locations of each type is created for each module.

Once a set of potential locations has been created for each type of location in each module, a ranking is constructed using information from every module in a design. To create an ordered ranking, it is necessary to weigh the final ranked signals and statements using a composite score (calculated during the selection process as the hypotenuse of the

two metrics used in Pareto selection) in order to consider only a one dimensional value in ranking creation. This ranking stage will select the top locations of each type from each module and aggregate them into lists for output to the user.

The following sections will address these metrics and selection processes in further detail.

5.3 Signal Scoring

Signal selection begins by creating a comprehensive list of signals contained within a module. This list will serve as the search space for Trojan payloads, as the selection of a signal for attack will determine most of the Trojan's potential functionality. Once all signals have been discovered and the connectivity between them extracted, they will undergo two rounds of scoring and a round of selection. In the first scoring round, signals will be assigned an initial score based purely on their signal definitions. This will serve as the basis for all scores derived in the second round. The second round scores of a signal are all calculated as a function of the initial signals around it, as connectivity is used to understand the signals relative to one another.

With all of the base metrics completed for each signal within a module, signal selection can now begin. In this thesis, three different types of potential Trojan signals are evaluated. Each of these groups is formed from an optimal selection using two of the base metrics from the scoring stage. This allows for a more diverse sampling of the circuit and helps discover interesting features of the module.

5.3.1 Base Signal Metrics

In the scoring stage, each signal will be evaluated based on its definition and usage within the module. Since this method evaluates code at the register-transfer-level (RTL), there are many synthesis-specific concepts that cannot be used for context. This means that this analysis will be more abstract than what might be available in a gate-level method. That said, it is believed that because this is the interface that an adversary inserting Trojans at the RTL would be using, high level understanding of signal function would be more useful than gate specific information. These metrics are intended to measure both the connectivity and structure of the the RTL description provided. Here, connectivity refers to the overall interconnection of signals through assignments, while structure takes into account the

conditional logic used to decide under what conditions assignments take place.

5.3.1.1 Initial Weight

To begin, signals must be given an *initial weight* estimate to measure their importance when compared to other signals within a module. This value will be comprised of only information contained within each signal's definition, as assigning value to a signal's connectivity without some basic concept of its importance would be difficult. For the sake of this discussion, this initial value score will be created automatically, although future investigation may take user assigned initial values into account.

The initial score for signal a is defined as:

$$a_{initial} = a_{port\_type\_score} + a_{signal\_type\_score} + a_{size\_score}$$

Where $a_{port\_type\_score}$ is a value selected based on the port type of a, such as input and outputs. More estimated importance is given to inputs and outputs, as they are often the most vulnerable components of a module for attack. The $a_{signal\_type\_score}$ is a value assigned based on the type of signal a represents, be it a wire or register. Here, registers are given more estimated importance, as they can hold state and may have impacts over several cycles. The $size\_score$ of a is used to estimate the importance of a single bit within a signal, as busses are often used in data paths, but single bits are used for control values. This means that while a signal may be comparatively large (for example 64 bits), designers will likely separate out more important signals into their own single bit line. This score is normalized to the largest signal within a module, assigning the highest score to single bit values:

$$a_{size\_score} = max\_score - max\_score(\frac{a_{size}}{max\_size})$$

Where $max\_score = 2$ and max_size is set to the width of the largest signal in the same module as a. A full enumeration of the values used to assign this initial score are available in Table 5.1.

Consider the following signal definitions:

Port Type	Value	Signal Type	Value	Signal Size	Value
input	2	Wire	1	1 Bit	2
output	2	Register	3	Max	0
inout	3				
internal	1				

Table 5.1: Values used for calculation of $a_{initial}$

```
input           serial;
output          data_ready;
output  [15:0]  result;
reg     [3:0]   counter;
wire            carry;
```

With the method described, each of these would be evaluated based on port type, signal type, and size, giving the scores found in Table 5.2. Here it is clear that the inputs, outputs, and registers usually score higher than other internal components even when accounting for size. This operates on the assumption that control-path signals are usually more important and will have more effect on circuit behavior when altered.

Signal	Initial Score
serial	5
data_ready	5
result	3
counter	5.5
carry	4

Table 5.2: Example Initial score values.

5.3.1.2 Impact

It is necessary to understand the potential influence a signal has on other signals in a module. This is done using a metric referred to as *impact*, which gives higher scores to signals that have a large amount of total fan-out. This score is calculated as the sum of all the initial scores of signals that are influenced by a given signal. The impact score for each signal is defined as follows:

$$a_{impact} = cycle\_multiplier * \sum_{i=0}^{l} F_{i,\ initial}$$

Where F is the set of all l many module signals that are influenced by signal a through forward assignment, F_i specifies a specific signal in the set, and $F_{i,\ initial}$ indicates that the initial score of element i is being considered. The $cycle\_multiplier$ is a scalar used to adjust based on circular dependencies used in state dependent calculations. This encompasses all signals within a module that are downstream of signal a to the outputs. Impact gives a numerical representation of how much of the module's state is tied to signal a and allows for a simple estimation of the impact a signal has on the entire module when changed. Higher scores indicate that a signal has more influence than other signals.

To account for the effects of feedback loops such as state-machine logic or accumulators, the impact score sum is scaled up by a multiplier. This multiplier exists in a range between 1.0 and 1.5 and is defined as:

$$cycle\_multiplier = 1.0 + 0.5 * \frac{1.0}{shortest\_cycle}$$

Where the $shortest\_cycle$ is the smallest number of forward assignments necessary to create a cycle within the connectivity graph. In most cases, the distance will be one as most state registers and accumulators often directly assign to themselves with minor some calculations, although there are some occasions were two or more assignments take place before a cycle is completed.

5.3.1.3 Susceptibility

In addition to the impact, it is also useful to measure how a signal is affected by other signals within a module. Due to this, a metric called *susceptibility* can be calculated from a signal's list of influencing signals. That is, susceptibility is derived from the full list of signals that influence signal a. This is similar to impact, but works in the opposite direction. Susceptibility is defined as:

$$a_{susceptibility} = cycle\_multiplier * \sum_{j=0}^{m} B_{j,\ initial}$$

Where B is the set of all m module signals that influence a given signal a through backward assignment (signal assignment l.h.s. and r.h.s. are inverted). Here, B_j identifies an element of signals and $B_{j,\ initial}$ refers specifically to the initial score of that element. Like impact, this score is a measure of connectivity relative to the initial scores of other signals, although it is a measurement of how important the signals used to create the final assignment of a signal a are, rather than its forward implications. Susceptibility seeks to quantify a signals total fan-in. Higher scores indicate that a signal has more dependency on other signals and that it is likely the result of a more complex calculation and larger upstream cone of influence.

In this instance, the value of *cycle_multiplier* is defined in the same way as it is for impact, as the distance necessary for a forward cycle is the same as the distance used for the same reverse cycle.

5.3.1.4 Controllability

Since Verilog designs are usually heavily based on conditional logic, it is also necessary to quantify the control structure implications of the design. For the sake of signal scoring, this is measured by a metric called *controllability*. This is not to be confused with the traditional concept of controllability used in test pattern generation and fault list selection. Here, controllability is another summed score taken over the signals that are involved in the conditional statements used to reach a specific signal. However, these do not include the signals used in the assignments themselves. The weights of signals directly controlling a conditional statement before the target signal's assignment will be normalized against the

total of all signals that have any conditional influence in a signal's assignment. This seeks to better understand how specifically controllable a signal is. This is a somewhat naive method, as it uses all control statements that affect a signal simultaneously. That said, this issue is remediated in later stages of scoring based on specific locations. Controllability is defined as:

$$a_{controllability} = \frac{\sum_{k=0}^{n} D_{k,\ initial}}{\sum_{i=0}^{p} C_{i,\ initial}}$$

Where D is the set of n signals directly involved in a conditional statement directly around specific assignments of signal a, and C is the total set of all p signals that influence any conditionals that affect a.

5.3.2 Signal Selection

This discussion will investigate three classifications of potential Trojan signal targets: Highly Destructive Signals, Intermittent Effect Signals, and Interconnect Signals.

In order to select the best signals of each type for a module, two criteria are used to select the Pareto optimal subset of signals based on the two metrics. Put simply, a Pareto optimal selection of data points (sometimes called the Pareto frontier, or Pareto efficient points) are the outermost points on a sample of data that represent the extremes of at least two dimensions defining the point. Using the Pareto optimal set ensures that both metrics are being considered and no one extreme of a data point is used as the selection criterion. This serves two purposes: one, that the signals selected are of the best representation of both metrics, and two, that that signals selected are diverse and encompass a more inclusive subset than what is possible from using only one metric to select target signals. Many of the other categories of signals and statements used later will rely on this method of selection. Once the optimal subset of signals is selected, the two metrics used to select them will be combined into a composite score by way of the Euclidean distance between the scores represented on perpendicular axes, similar to calculating the hypotenuse of a right triangle. This allows for a relative ranking to be constructed in such a way that signals that are high in both metrics are rewarded but signals with only one high score are still allowed to compete.

5.3.2.1 Highly Destructive Signals

Highly destructive signals are those that are assumed to have a large impact if altered, potentially destroying all module functionality. Signals in this category are usually associated with many other signals through assignment or have controlling characteristics over some of the most heavily weighted initial scores within a module. This means that they are likely to have a high impact score. In order for these signals to have their potentially destructive nature, they must also be assigned often or be easy to reach in the conditional logic of the module. This means that highly destructive signals will be within the code blocks evaluated most often due to the nature of their controlling conditional logic, which is approximated by the controllability metric of that signal. If an attacker's intention is to destroy as much a circuit's functionality as possible, signals of this type are likely targets because their corruption will have catastrophic effects on other signals.

5.3.2.2 Intermittent Effect Signals

The attacker's goal in inserting a behavior modifier Trojan into a design is unknown, so it is important to consider multiple avenues of attack. While one might assume that the most destructive approach is an obvious choice, creating intermittent issues within a design can be very damaging. In order to consider signals that could be used for this method of attack, the tool considers a second type of signal classification called an *intermittent effect signal*. These signals are expected to be those that are rarely used or that are meant to manage edge cases in normal operations. As such, alterations to their correct values should only be felt in certain instances, making their appearance harder to detect from a functional observation standpoint. To select these automatically, signal scores are optimally selected based on the minimum impact, so as to keep effects small, and on minimum signal controllability, to ensure that the actual site for Trojan insertion is in a hard to reach block within the logic.

5.3.2.3 Interconnect Signals

Since the module-based structure of most Verilog designs lends itself to hierarchical organization, it was also necessary to investigate module interconnects as potential sites for Trojan insertion. This was done with the introduction of a third category of signal that was liable to attack, called *interconnect* signals. These signals are found in high-level modules

that describe some sort of routing between submodules instantiated inside themselves. Simply, *interconnect signals* are sites within a module where a high valued submodule output is wired into another high valued submodule input. To find the most interesting interconnect signals, the tool constructs a set of submodule signal routing pairs and evaluates specific output to input combinations that have the highest scores. Unlike destructive and intermittent behavior modification signals, interconnect signals are selected using information from two connected sub-modules and are not selected optimally in the same way. Instead, a compound score is taken for each pairing and added to a sorted ranking list. When final selection occurs, the top n elements of this list can be selected for further consideration.

	Highly Destructive	Intermittent Effect	Connectivity
Criteria 1	Max Impact	Min Impact	High Output Suscept.
Criteria 2	Max Controlability	Min Controllability	High Input Impact

Table 5.3: Criteria used for signal selection by type.

After the initial scoring stage is complete, each module's signals are evaluated based on the category definitions defined above. Through this process, signals of each type may not be found in a every module. Overall, this step is to inform the user and later stages of processing which signals should be investigated further, by calculating and comparing the heuristics used to give them their rankings. From the results of signal selection, the process of finding the most interesting and dangerous lines of logic in the design source code can begin.

5.4 Statement Scoring

Much of Verilog's usefulness for RTL specification comes from its high-level conditional constructs. These constructs, such as IF statements and CASE statements, allow designers to implement robust logic structures without needing to specify every gate of a design. By making this level of abstraction available, designers can save time in the design process, as

well as create more human readable descriptions of logic functionality. It is then expected that synthesis tools convert this high-level description into a gate-level representation through synthesis, so that the design can be implemented. There are many components necessary to preform this translation from high to low abstraction, so opportunities for ambiguous or dangerously specified lines of source code to manifest into design issues are often masked by the synthesis tool's inherent complexity or the complexity of the final realization of the initial description. For Trojans inserted at the RTL, this gives an attacker the opportunity to insert small changes in source code that have hidden effects due to the details of the synthesis tool or any implementation specific design rules. This, among other considerations, will be used to score and rank the statements that comprise module functionality for potential Trojan sites.

5.4.1 Base Statement Metrics

For the purposes of this investigation, it becomes necessary to evaluate circuit logic through the lens of high abstraction, since there is no gate-level representation of the design available to our tool. As a result, selecting lines of source code to investigate for potential Trojan locations must rely on high-level concepts that are inherently related to how signals are assigned within the RTL domain. This is accomplished using a set of metrics calculated for each statement of Verilog code that implements some sort of logical functionality. This primarily includes assignments and conditional structures. While submodule instantiations are important for evaluating the interconnection of signals within a module, they do not directly provide more information on what those signals are for or how logic elements are specified.

5.4.1.1 Complexity

One of the most directly accessible and apparent metrics that can be used to describe an expression in Verilog is some measurement of complexity. Here, *complexity* is thought of as an estimation of potential gate count for an expression's gate-level representation after synthesis. This enables a context-aware evaluation of high-level operations that takes into account an estimated amount of low-level logic necessary for gate-level implementation. Since the gates available in a cell library used to implement a design and the exact synthesis techniques used

to create complex operators are not known to this analysis, rough approximations based on non-optimized circuit patterns are used as a conservative estimate of operator complexity.

Operator	Gate-Count Approx.
$+$	$n * 5$
$*$	n^2
&	$n - 1$
&&	n
$>$	$\frac{n(n+1)}{2} + 2n$
$==$	$\frac{n}{2} + n$
$>=$	$\frac{n}{2} + \frac{n(n+1)}{2} + 3n$

Table 5.4: Examples of operator complexity estimation functions based on bit width (n).

To evaluate an entire line using this method, all operators are parsed out and stored for processing. The operators are then iterated over, evaluating each of their complexities based on the size of n, the number of bits in the output, according to rules for approximating gate count created for this application. Some examples of these rules are listed in Table 5.4. Single operator scores are totaled and reported as the complete score for that statement. The total complexity score for a statement s can be denoted as:

$$s_{complexity} = \sum_{j=0}^{m} Complex(O_i, n)$$

Where O is the set of all m operator symbols found within statement s and n is the width of the largest signal involved in that statement. The function $Complex(O_i, n)$ returns the value of the expression for gate count approximation depicted in Table 5.4 evaluated for that specific n.

5.4.1.2 Conditional Leaning

While it is impossible to know the values certain signals are likely to hold during operation without simulation data, it is possible to make educated guesses as to how often certain

conditional blocks are evaluated. This is estimated through a metric called *conditional leaning*. All conditional constructs result in one of two outcomes: either the logic with a conditional block is executed, or it is not. How conditional logic like this is synthesized into gate-level implementations may greatly vary from design to design and synthesis tool to synthesis tool. However, the designer does give some information about how decisions are made at a high level. Because this approach assumes no additional information about signal values, it is hard to know with certainty if specific conditional blocks are evaluated more frequently. Despite this, if it is assumed that all signals can take on any value and do so with equal probability, we can study how the operations used in a conditional block can potentially bias the outcome in one way or the other.

The simplest example of this is any basic two input logic gate. Take, for example, an AND gate with two inputs whose values can be 0 or 1 with equal probability. If many trials were attempted where the output of this gate was measured given random inputs, it would have a tendency toward 0 with a split of 25% to 1 and 75% to 0 as more trials were attempted. Table 5.5 shows how this is a function of the truth table of an operation and how the operation of a logical AND will create a noticeable bias toward one outcome.

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

Table 5.5: Outcome bias of a 2-input AND gate with equal probability inputs with three zeros and one one.

This concept can be applied to all two-input gates as well as all logical and relational operators. For instance, consider the equality operator. In a two input, one bit case, this can be implemented with a single XOR gate. As the inputs get larger however the equality operator's implementation grows more complex and the likelihood of two signals having the

same value decreases significantly, as only a small number of conditions will satisfy equality but 2^{2n} total input combinations exist, if each of the two inputs is of size n. Here, equality is only met when two n bit wide values are exactly the same, meaning that only 2^n of the total 2^{2n} input combinations yield true. Since it is impossible to know exactly what the bias of each of each large operator is, they can only be classified generally. Examples of these operators and their assumed effects can be seen in Table 5.6.

Operator	Likelihood of 1
<	Equal
>	Equal
<=	Equal
>=	Equal
==	Very Unlikely
!=	Very Likely
!	Equal
&&	Unlikely
\|\| | Likely |

Table 5.6: Operators in Conditionals with bias toward 1 or 0.

Similar to the calculation of statement complexity above, a score can be given to a whole statement by considering all the operators that make it up. This is done by iterating over the operators used within a conditional and evaluating each of their bias'. It would not be fair to consider an exact probability of any sort, but the conditional leaning can be captured in a value between 1.0 and 0.0, where statements closer to 1.0 are expected to commonly evaluate as true, leanings toward 0.0 are expected to be commonly false, and values at 0.5 are split equally. Each statement is assigned a score of 0.25, 0.5, or 0.75 depending on its estimated leaning during this step.

Since these leaning scores are designed to describe conditional statements, it is important to also consider the effect of conditional block structure for not just IF statements, but for those with multiple components, such as ELSE and ELSE IF statements. In the simple

case of an IF statement with a trailing ELSE statement, the conditional leaning calculated for the IF condition can be complemented (subtracted from 1.0) and assigned to the ELSE statement. More complex rules are necessary to assign values to each component of a chain of IF, ELSE IF, and ELSE statements, as the result of the previous block must be used when evaluating the current block.

Consider the following logic:

```
if (a && b) begin
    // block 1
    end
else begin
    // block 2
    end
```

In this example, block 1 is evaluated if the value of both a and b are 1. This means that this block would have a conditional leaning toward 0 and would be assigned a score of 0.25. Since the ELSE statement defines the behavior should the test of the IF fail, block 2 will be evaluated only when the IF statement is not true. In this case the leaning of the ELSE statement would account for any misses by the IF statement and would tend toward being evaluated more often (toward 1.0). The contents of the ELSE block would be assigned a score of 0.75 as that is the remaining probability. Again, this metric does not seek to assign any direct estimate of the probability a statement is true, as its purpose is more to give a relative measure of how often logic within a conditional might be evaluated. This type of approach is also necessary for managing CASE statements, which have a very similar effect.

It is understood that this is a naive approach, as the characteristics of the input variables used in conditionals will often not match the fifty-fifty assumption used here. Future work will look toward more accurately representing these characteristics in this calculation to account for more true-to-life evaluation.

5.4.1.3 Reachability

By its definition, the conditional leaning of a conditional statement only applies to the outcome of that specific statement. To expand upon this and understand an entire module's

flow, the metric of *reachability* was created. This is a metric defined for all lines of functional logic (assignments and conditionals) within a module, as it seeks to quantify how often each line of logic is evaluated. It is an extension of conditional leaning, but it uses the tree-like structure of conditional statements to assign a score to every level of nested structure and the statements within them. It is calculated by traversing the logic tree and taking a product of all conditional leaning scores along a path from the root of the design (the outermost logic such as ALWAYS statements and continuous assignments) to a given node (any internal statement within control structures).

Consider the following logic:

```
if (a || b) begin           // Conditional 1
    if (c) begin            // Conditional 2
        x = 1;              // Assignment 1
    end
    else                    // Conditional 2 (ELSE)
        x = 2;              // Assignment 2
end
else                        // Conditional 1 (ELSE)
    x = 3;                  // Assignment 3
    if (c && d) begin           // Conditional 3
        y = 1;              // Assignment 4
    end
end
```

Where a, b, and c are single bit signals.

This logic structure can be represented as a simple logical tree, where statements that are parallel (within the same level of nesting) are considered to have the same reachability. Whenever there is a conditional statement, the reachability of the current level is multiplied by the conditional leaning of that statement and used for all children within that conditional statement's control. This can be seen in Figure 5.2, where the conditional leaning of control statements is shown in [] and statement reachability is show in {}. Notice that the reacha-

bility scores for nodes $A3$ and $C3$ are identical. This is because they are at the same level (within the outermost ELSE block). Reachability only changes when entering a conditional block. By default, the top-most statements without higher control structures are given a Reachability of 1.0.

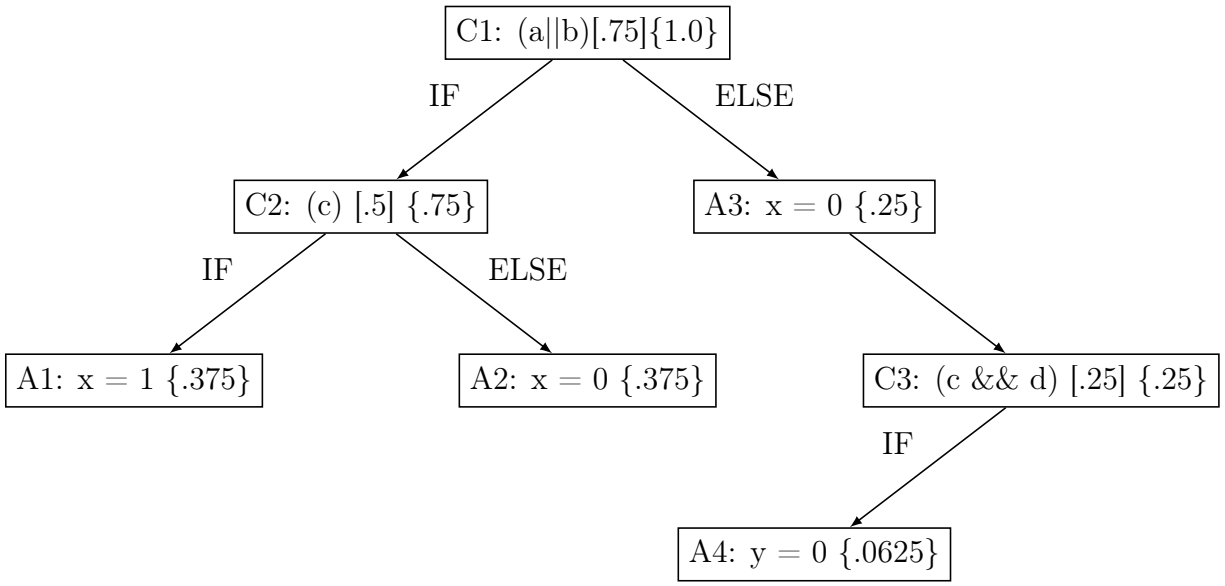

Figure 5.2: Example logical graph of simple conditional code with Conditional Leaning and Reachability scores.

The reachability of any statement can be defined as:

$$s_{reachability} = \prod_{k=0}^{n} L_{i,conditional\_leaning}$$

Where L is the set of all n conditional logic statements that enclose a statement s. In the example listed above, the set L of statement $A2$ would be $\{C1, C2\}$ and would have a reachability score of $0.75 * 0.75 = 0.375$.

5.4.2 Statement and Location Selection

Similar to the categorization of signals in 5.3.2, statements can be categorized similarly into three parallel classes. In fact, in order to select statements for investigation, the signals selected earlier will serve as a starting point for the statement selection process. This is because even though there may be statements that are inherently interesting based on the metrics defined above, they are of little use if they are not in some way involved with a interesting signal from section 5.3.2. To begin statement selection, statements within a module are searched for based on their relationship to a signal of interest. With this subset of statements selected, they are then ranked based on the metrics related to each type of signal.

It is important to note that this method will determine a line of code that can be used to alter a target signal. This means that once a line is found to be interesting, any edits to this statement or values going into it will be potential Trojan sites. Additionally, locations downstream of this location could also be potential Trojan locations. For the sake of this thesis however, these potential locations will not be evaluated, as their impacts will likely be similar to the original location identified.

5.4.2.1 Highly Destructive Assignments

Highly destructive assignments are statements that are assumed to have catastrophic effects if tampered with. This means that small changes in input signals or clever replacements of operators (such as switching a '«' (Left Shift) with a '«<' (Arithmetic Left Shift)) could result in large changes in the value they assign to a target signal. To understand what a statement of this type looks like, consider first arithmetic operations. As noted in 5.4.1.1, arithmetic operations will likely synthesize into a large number of gates using library components or external IP. If an attacker is clever, knowing the details of this implementation might allow a seemingly innocuous change to have significant effects in certain situations. From this, it is reasonable to infer that the higher complexity of a statement allows for better concealment of malicious changes. Additionally, the attacker will likely want to create as much damage as possible in one of these situations, so choosing logic that is used often is likely desirable as well. For these reasons, statements of this type are selected using the

efficient Pareto front created from maximal assignment complexity and maximal reachability.

5.4.2.2 Intermittent Effect Assignments

Contrary to the objectives of a highly destructive signals and assignments, *intermittent effect signals and assignments* should cause issues infrequently or only in certain conditions. In order to ensure that Trojan injection is harder to detect, it is still necessary that the complexity of the target assignment is high; however, it is probable that lower reachability is desired. Together, these help guarantee that a Trojan will have more sporadic effects (assuming it does not completely break the logic), leading to reliability issues or degradation in performance and accuracy. Great care must be taken to evaluate edge cases during testing if this type of Trojan is to be detected. This is a common issue in design verification, to approaches for finding and studying corner cases used in verification will likely be usefull for evaluation Trojan locations of this type.

5.4.2.3 Interconnect Selection

Unlike the previous two types of RTL Trojan locations, interconnection logic does not rely on assignments or conditionals. As such, the definition of a signal that is used to connect two submodules within a larger module will be marked as the location for Trojan insertion. This leaves it up to the user to decide how they would like to interfere with the interconnection. Locations of this type open the door for many types of attack, both within the module they are identified in, as well as in the submodules they connect. As such, attacks on the selected output, input, and the connecting signal should be considered.

5.5 Final Ranking

Until this point, all analysis has been conducted on a per module basis. In order to provide a useful report to the user, it is necessary that the most dangerous locations in the designs are selected in an intelligent way. To do this, the top scoring locations (based on composite scores) of each type are selected and placed into a universal ranking that considers the entire design. This creates the issue of comparing the threat of one Trojan location in a given module with a different Trojan location in another module. For the sake of simplicity, the composite scores from the last stage are evaluated on a logarithmic scale by default, as scores

from different modules differ greatly in magnitude as they are not normalized against each other in any way. Since this function is monotonic, it will not change the order of the ranked locations, but only reduce the scores in magnitude. This keeps the user from assuming that large differences between scores are deeply meaningful and keeps scores smaller and more directly comparable. The goal of this tool is to rank potential Trojan locations rather than give concrete numerical analysis. It should be noted then, that the exact numerical value of each score is arbitrary and should only be used to compare locations within the same design.

A round robin approach is used to help maintain good diversity in the final set of selected locations. This is done by taking the maximum scoring location from the design and then locking that module's locations out for the following rounds. The next location selected must come from a different module, which is then also locked out. This continues until a location is selected from every available module; then the locked out modules are released and the process begins again. Without this process, certain modules with a large amount of internal signals or logic will completely dominate the list, keeping smaller or simpler modules from ever being reported. A process similar to this is also run in the per module location selection stage to ensure that not every location is associated with the same signal.

5.5.1 Global Ranking

To find the locations that cause the most concern, the user might be interested in seeing which locations within the design are scored the highest. The tool can then select n many of the top most locations for each category. Using the selection approach listed above, all locations within the entire design are positioned in one, global ranking for each category. This ranking will place locations that have the highest composite scores, and are thus assumed to be the most likely or dangerous locations for a Trojan, at the top of the list, with other locations of lessening scores placed in descending order below. From this, a report is generated that describes every location selected, as well as its score and some pertinent information about the module it is in and the signal that it targets.

5.5.2 Height Ranking

The kinds of modules used to create a design will vary in composition from the top module down to the bottom of the design hierarchy. At the top of the design hierarchy, modules are

usually very abstract and are used to instantiate and connect numerous submodules. Should one follow a path from the top module down the lowest submodule, module composition will become more and more specific, transitioning from mostly submodule instances to conditional and sequential logic or data tables. Because of the different purposes of modules found at different levels within the design hierarchy, it is also useful to create design level rankings based on the 'height' of modules. Here, height is a measure of how many levels of submodule exist below a module's definition. So, a module that has no submodules within itself would have a height of zero, and the top module would have the maximum possible height for that design. Creating the height-wise rankings is essentially the same processes as creating the global ranking; however, only the modules in each height grouping are considered at a time, and each group is considered a separate global ranking for that height. This type of ranking is mostly useful for small designs where differences in module purpose are obvious. As designs increase in size and complexity, this type of ranking becomes less useful, as some branches of the hierarchy tree will be significantly longer than others, disrupting the usefulness of this type of this ranking. In general, this ranking can be interesting for manually evaluating the types locations being identified in different types of modules, but its results are very similar to that of the global ranking, so it is not very useful beyond this application.

Chapter 6
EXPERIMENTS

To evaluate the effectiveness of the previously described ranking approach, several experiments were conducted. Two designs were processed using the tool-flow and scoring system described in Chapters 4 and 5. These designs range in size from a small 16-bit MIPS processor to a navigation control system implemented using several internal communication protocols. These designs were processed mostly as is, but minor changes to the source code were necessary to meet some of the limitations detailed in Section 4.5.

6.1 16-Bit MIPS Processor

To validate that the tool was able to process designs and locate interesting signals and locations, it was first used to evaluate a simple 16-Bit MIPS processor [25]. This design was selected for its simple structure and small size so that the selections made by the tool would be easy to understand and validate by hand. The goal here was to ensure that the tool was finding the types of signals and locations that were expected to score highly according to the metrics defined in Section 5 and verify that the details critical to the designs were being properly extracted. This discussion will assume some basic knowledge of the MIPS processor architecture at a high-level.

To evaluate this design, the top twenty locations of each of the three types listed in Section 5.4.2 were identified by the tool and then and investigated by hand. Since this design is small, several locations using the same signal were selected when creating the top-ten rankings. Here, only a few unique locations will be discussed in detail, but the full top-ten rankings are available in Appendix A.1.

6.1.1 Destructive Locations

From the analysis, the signal *pc* from the instruction fetch stage was selected as a dangerous signal, and its specific assignment on line 37 was deemed the most destructive location

Type	Rank	Module	Signal	Location Line
Destructive	1	IF_stage	pc	37
	2	alu	r	38
	8	WB_stage	reg_write_data	39
Intermittent	1	ID_stage	ex_alu_cmd	203
	2	register_file	reg_write_dest	54
	12	EX_stage	pipeline_reg_out	52
Interconnect	1	mips_16_core_top	ID_pipeline_reg_out	29
	2	mips_16_core_top	instruction	28
	3	mips_16_core_top	decoding_op_src2	38

Table 6.1: Overview of unique locations in 16-Bit MIPS Processor

in the design. Those familiar with most processor designs will immediately recognize this signal as the program counter, a core component used to keep machine state and determine program execution order. It is understandable that this signal would be a good target for a destructive Trojan, as any alteration to the value of *pc* would cause incorrect instructions to be fetched or could allow malicious code from other blocks of memory to be executed. The line that was selected as a potential Trojan site in the source code is as follows [25]:

```
pc <= pc + {{(‘PC_WIDTH-6){branch_offset_imm[5]}}, branch_offset_imm[5:0]};
```

This statement is used to calculate the next value of the program counter using a branch offset to jump to another location within the instruction memory. Unlike the standard update of the program counter (stepping forward one memory address), this calculation allows for a relative branching increment to be added to the program counter. Attacks at this location could be implemented by changing the branch offset (thus moving to an unintended location), stopping the update from occurring by removing the offset, or completely overwriting *pc*. This specific line was selected for its high complexity. Even though the ‘+’ (addition) operator is used in other lines of this module, the use of the ‘(){}’ (replication) and ‘{}’ (concatenation) operators add to the statement’s complexity. Therefore, it would be easier to insert hard to detect changes within this line.

Another destructive location was identified within the arithmetic logic unit (ALU). Since this module is the center for all complex operations within this circuit, it is reasonable that

the result of computation (signal r) was selected as a highly-destructive location. Specifically, the contents of line 38 were targeted, where the shift right operation is conducted:

```
r = {{16{a[15]}},a} >> b;
```

Shifting operations require a fair amount of logic to implement, so they are associated with a relatively high complexity score. Within this device, both a shift right signed and a shift right unsigned operation are implemented. Since the shift right signed operation requires an extension of the sign bit, it is slightly more complex than the shift right unsigned operation. Attacks on this line could include altering the shift amount, injecting sign extension errors, or using the '>>>' (arithmetic shift) operator in place of the other implementation (simple shift with manual sign bit extension), potentially changing the circuit depending on synthesis details.

Several other locations for attacks on pc and r make up many of the top locations in the ranking. Moving down to the eighth position, the tool identifies an attack on signal reg_write_data within the write-back stage. This signal is the value assigned to a register when write-back operations take place. While this signal does not branch out much within this module, its impact on the register state could have negative impact on later calculations or allow for unexpected edits to memory. This specific assignment is implemented using the Verilog ternary operator, which is usually synthesized into a multiplexer whose control is tied to the outcome of a conditional statement:

```
reg_write_data = (write_back_result_mux)? mem_read_data : ex_alu_result;
```

In this case, the signal being written to the register file is assigned to one of two input signal that are selected by a single bit control value. Attacks here could include forcing data read from memory to be written into the registers unintentionally or clearing register contents.

6.1.2 Intermittent Locations

When considering intermittent locations, it is important to remember that these sites are selected using the *inverse*, or lowest, reachability used in selecting destructive locations. This serves the purpose of making circuit behavior unpredictable or "glitchy" after the Trojan

is triggered, making it harder to detect and diagnose. This means that relative to the destructive locations, the statements selected here are expected to be evaluated less often and are commonly within multiple levels of nested conditional logic.

The first signal selected for possible intermittent locations is the *ex_alu_cmd* signal found within the instruction decode stage of the pipeline. In fact, this signal makes up most of the top ten rankings, as it is assigned in various locations throughout this module (once for each possible ALU operation). This means that the specific line number for this location is not too critical, as several sites could be investigated. What is important is that this signal's value is later used to control the ALU's output function after being stored in a pipeline register. If this value were to be altered slightly in any statement in this module, say to switch the execution of the signed shift and unsigned shift operators, it would likely be difficult to pinpoint, as only specific conditions would cause an error to manifest and then only once the Trojan was triggered. Additionally, because this signal's value is not immediately used for calculation (it must pass through the pipeline register), its effects would only be measurable two cycles later when the result of the execution stage was incorrect.

The next location was identified within the register file, a set of registers used as a sort of working memory within the pipeline. Here, the targeted signal, *reg_write_dest*, is used as the location (index) for assignment of values within the register file, implemented as an array of registers. It is immediately clear that changing this value could result in incorrect assignments to the register file, causing a potential derailment of the intended program.

A third interesting location comes from the execution stage's *pipeline_reg_out* signal. In this implementation of the pipeline, many of the values from the previous stage (instruction decode) are simply passed through the execution stage into the next pipeline register with no change. Since these values are accessible within the execution stage, they can be altered from within a module that should have no effect on them, which could have interesting implications from the point of view of the attacker. Should an attacker want to edit or disrupt values necessary to later stages, they could do so by editing values values of the pipeline register assignment within a module that that is not directly related to the point an error would manifest at later in the pipeline. It is interesting to note that this issue is an artifact of architectural design choices. Had the signals necessary for each stage been

separated out at a higher level (in the top module), modules would only have access to the signals that are necessary for their functionality. This approach violates the principle of *least privilege* [26], as unrelated components can potentially access information not critical to their operation. While this concept is usually applied to users within an information system, it can be applied here, as not every stage of the MIPS pipeline needs access to the entire pipeline state. Better implementation of the least privilege policy could help reduce the number of locations that make these kinds of attacks possible.

6.1.3 Interconnect Locations

Complex designs are usually made from a large number of modules assembled in a hierarchy. This allows for good code reuse, as well as separating functionality into discrete and separate components for high level specification. While useful to the design process, this gives attackers the ability to study module functionality abstractly, meaning that important input and output signals can be selected based purely on their names and module specifications. It is important to study how modules (submodules) are connected within higher levels of the design hierarchy. Since this is a small design, there are only three levels of hierarchy, and almost all submodule interconnection occurs within the top-level module. Each of the interconnect signals identified are used to connect stages within the pipeline with the exception of one used for hazard detection. Attacks on these locations could be simple reassignments to other values, sporadic errors, or even the injection of specific values that cause pre-determined outcomes, such as forcing the processor to stall indefinitely.

6.2 Navigation Controller

The goal of this approach is to locate many potential Trojan locations in a complex design automatically, so it was necessary to verify its functionality on a full, multi-function design. To test their tool, designers behind the aforementioned "Achilles" project (Section 2.2.1) also created a device design (sometimes referred to as the *golden model*) that implements a full control system for a simple navigation application, to use as a bench-marking circuit for their product. This controller consists of components to interface with sensors, user input peripherals, and motor controllers. To facilitate communication between these components, it also contains extensive network of communication devices using various standards (such

as SPI, UART, and AXI). The full details of the design are too large for the scope of this discussion, but a high-level view of the design and its components is elaborated below. For the sake of evaluation, the designers also implemented several hardware Trojans to evaluate the performance of "Achilles" overall as well as the approach put forward in this work. With access to some Trojan examples, it is possible to compare the results of the automatic location analysis to what might be found in a compromised, real-world circuit. It should be noted that the Trojan designs created for this system were developed by a different set of engineers and did not include the primary creator of the metrics and algorithms described in this thesis.

First, some important details of the design will be outlined and the device architecture explained at a high level. Next, some details of the manually created Trojans will be briefly explained to give context to their operation and give some insight into how this design could be attacked. Finally, interesting locations found by the automated analysis will be investigated and compared to hardware Trojans placed by hand. By comparing these two sources of Trojan relevant design locations, the benefits and drawbacks of the automated approach can be better understood.

6.2.1 Architecture

In order to emulate real-world applications, the navigational controller design was built to control a variety of sensors and actuators as well as utilize a host of communication protocols. This was done so that lessons learned while conducting analysis of this design could easily map to other applications in the future that used similar technologies. To keep this setup true to life, it was implemented on a commercial-off-the-shelf (COTS) field-programmable-gate-array (FPGA) development board, so that it could be studied closely by the team working on "Achilles" during all stages of development. This also allows different kinds of sensor data to be collected and used in the detection and learning stages of the project mentioned in Section 2.2.1. While this design does not have the massive scale of some modern FPGA applications, it is complex enough to be a good proxy for some aspects of standard industry and military designs. A high level diagram detailing these components is provided in Figure 6.1.

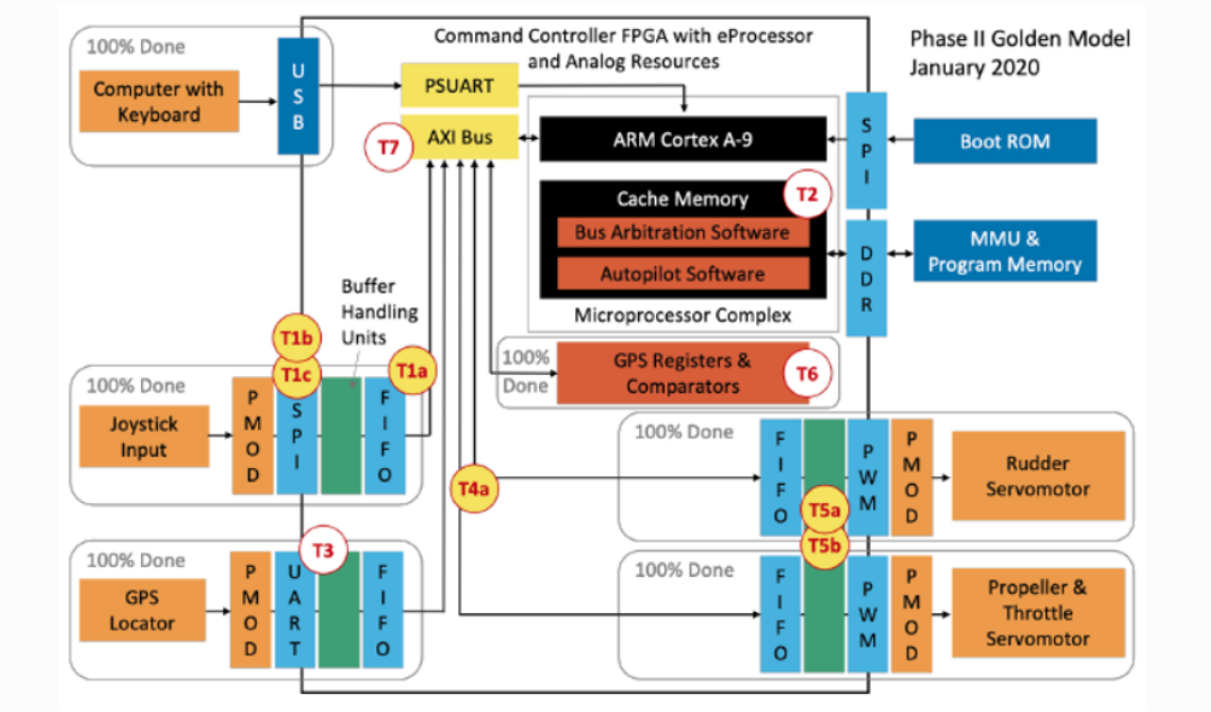

Figure 6.1: Navigation controller high-level block diagram.

Since the goal of this design is to provide a sort of simulation of a real-world application, several physical actuators and input devices are connected to the COTS board to make a fully functioning system. This includes motors and servos for a "propeller" and "rudder" controlled by pulse-width-modulated (PWM) signals generated by the FPGA logic. Location information is provided by a discrete GPS module. A keyboard is connected to allow the user to input GPS coordinates (to simulate an autopilot function), as well as a joystick for manual control of motion components, such as a propeller and rudder. These elements are all stitched together using the FPGA fabric and interfacing wrappers to manage device specific control logic and inter-device communications. The chosen evaluation board also includes an ARM Cortex A-9 processor that acts as the algorithmic center of the design. In this processor, sensor data is converted into desired action commands sent to the actuator peripherals. While each physical component requires a different type of direct interface communication protocol (such as SPI, UART, PWM signals, or PSUART for USB), most components are unified using the AXI bus protocol. This means that all sensor and motion

data will move through the AXI protocol at some point.

For the sake of this experiment, only the programmable logic components of the design were evaluated. This excludes the implementation of the ARM processor and its associated memories as well as the physical actuators and sensors themselves. All logic for interfacing with peripherals was included. Since the design source-code is not available for distribution, some additional details will be provided about the selected signals and locations in later sections and in Appendix A.3.

6.2.2 Hand Placed Trojans

Six Trojans with various triggers and payloads were developed to act as test cases for the "Achilles" tool-chain. They were intentionally defined to have different triggering behaviors and payload activities in order to ensure that a wide variety of different Trojan types could be evaluated. It is important to note that all of these Trojans belong to either the *behavior modification* or *reliability impact* categories. Since these two categories of Trojan often implement their payloads in a similar way, their characteristics are in line with the objective of the scoring system in Chapter 5. Details elaborating the specific activation characteristics and effects of each Trojan can be found in Appendix A.4, and the approximate location of each Trojan is shown in Figure 6.1 as yellow circles. Other Trojans are being developed for this design but were not completed in time for this investigation (shown as white circles).

In general, these Trojans have three main effects on the system: lockout, reliability degradation, or take-over. Trojans T1a and T5c inhibit the user's ability to control parts of the system directly by disabling functions or freezing internal signals in a certain state, making certain controls unresponsive, or interfering with expected behavior. Trojans T1b and T5a interfere with communication protocols, causing control information (PWM signals) or messages (SPI transmissions) to be corrupted, leading to reduced functionality or erroneous and sporadic behavior. Finally, the Trojans T1c and T4a allow control of the system to be taken over by an unexpected peripheral, such as an extra joystick connected to unused I/O pins or though unused buttons built into the COTS evaluation board, allowing an outside source to control the navigation system. With the exception of the take-over Trojans, all of these Trojans are triggered using timers or internal state-machines, a notoriously difficult to find

type of Trojan trigger. More information on these Trojans can be found in Appendix A.3.

6.2.3 Tool-Identified Trojan Locations

Similar to the analysis of the MIPS processor, unique locations that were automatically identified from the navigation controller have been listed in Table 6.2 with the full listings of each location type in Appendix A.3. Since this design is much larger, there is more variety present in what modules were selected in the final round-robin process. With that in mind, it is interesting to note that there are many similar signals, such as signal $o\_m0n\_axi\_rdata$, which is present in many modules and makes up eight of the top twenty positions. This is due in part to the fact that the interfacing modules in this design all have AXI bus connectivity and must interact with the protocol accordingly. Since they are technically different signals, however (each one is specific to the instance it is used in and each is assigned values based on the components it interacts with), they must be considered separately. While some of these duplicate signals are almost identical, slight variations in their usage could be interesting locations to hide novel Trojans, so they not discarded. For the sake of this discussion, only the top ranked signals from of each of these groups will be investigated.

Type	Rank	Module	Signal	Related Trojan
Destructive	1	axi_router	r_axi_router_int-_data	Novel
	2	pwm_if	r_active_time	T5a, T5c
	3	gps_if	o_m0n_axi_rdata	Novel
	12	joystick_if	r_spi_byte_cntr	T1a, T1b
Intermittent	1	spi_if	r_mosi_cpha0	T1b
	3	gps_if	o_m0n_axi_rresp	Novel
	19	axi_router	o_axi_router_rd_en	Novel
Interconnect	1	ship_control_pl-_wrapper	w_m00_axi_router-_addr	Novel
	12	axi_router	w_upm_mask	Novel
	13	propeller_servo-_if	w_esm_data	T4a

Table 6.2: Overview of unique locations in Navigation Controller

6.2.3.1 Destructive Locations

The highest rated destructive location, as well as several other identified locations, target the AXI communication components as sites for potential Trojans. In the case of the top-ranked location, a register within the axi_router module that stores the data present on the bus was deemed most destructive. The axi_router module serves as the central bus and arbitration module for all AXI communication within the design. The selected register is an internal storage location that is written-to and read-from in each write or read operation conducted by the peripheral devices. At this central location, data in this register will be used for all communication transactions, and thus it will impact the operation of every peripheral. This signal was selected because of its massive connectivity, as well as the complex assignment statements used to update it. Attacks here could have catastrophic impacts, as an attacker could disrupt all messages moving through the bus (causing a denial-of-service), corrupt some messages (thus degrading reliability), or replace specific values with others (to create take-over or behavior modification effects). While the tool did find a specific line to alter this signal on, almost any point within this module is a good site to alter it. Since this location is central, a clever attacker could use a Trojan here to impact almost every behavior of the system from afar. No Trojan has been created to utilize this site, but it is recommended that it be investigated further by the "Achilles" group. Locations identified by the automatic analysis that have no counterpart in the hand placed Trojan group, such as this one, are denoted as "novel" in Table 6.2.

The next three unique locations could all be used to disrupt communication protocols in some other way. Position two in the ranking points to a location within the pwm_if module that manages the creation of PWM signals for controlling servo motors. There are two instances of this module within the design, one for controlling the rudder, and one for controlling the propeller. The signal identified (r_active_time) is used to limit the amount of time used for holding a PWM signal in the high or low state, thus determining the pulse width of the signal sent to the servo. The specific location identified is a block of logic used for bounds checking when new commands are received for motor control. Altering this value would allow for erroneous PWM control signals, causing outward malfunctions of the servos. This is closely related to the effects of Trojans T5a and T5c, which both force changes to

the PWM signals by altering the timers used to toggle states.

The third ranked location, and many after it, target the values being written out to the AXI bus during read operations by the AXI controller. Any edits to the $o\_m0n\_axi\_rdata$ signal could be interpreted as providing false information to the host controller. This sort of falsification could be used to confuse navigation algorithms (providing false GPS data) or directly steer the rudder and propeller. Alternatively, blocking output could cause lockup of the system. No hand-created Trojans for this site were identified, but these locations could be used for a variety of attacks.

The fourth unique location, or the twelfth destructive location identified, is a site used in managing the communication with the joystick peripheral via a SPI connection. This signal, $r\_spi\_byte\_cntr$, is used to keep track of the number of bytes transmitted or received during a transaction with the SPI peripheral. Should it be blocked or altered, SPI transmissions could be lost, leading to an overall decline in communication reliability, or in a complete denial-of-service. This signal was exactly identified for use in Trojan T1a, and is tangentially related to the reliability attack of Trojan T1b.

6.2.3.2 Intermittent Locations

As mentioned before, intermittent locations are well suited for reliability attacks, as their effects should only appear some of the time. This means that within this design, several control signals for communication protocols were identified as good sites for intermittent behavior modification. The first of these is the r_mosi_cpha0 signal within the SPI interface module. This signal is used in setting the phase of the data clock, or which clock edge SPI data should be read on. Altering this could cause undefined behavior during transmission, causing corrupted or dropped messages. This location is somewhat related to the attack implemented in Trojan T1b, which also attacks SPI clocking behavior, although in a different way. Several other locations identified in this category could be used for similar Trojans.

The third ranked location, as well as many others like it, target the $o\_m0n\_axi\_rresp$ signal used by the AXI protocol. This signal is a response used during read operations to tell the host controller if there is data to read and if that data is valid during control handshakes between the AXI devices. Should it be altered, attackers could halt read operations by

breaking handshakes and causing slowdown (if intermittently changed) or complete lockout (if forced to a single value). Since this signal is used in every AXI connection, it is important to understand how attacks in each module could manifest differently depending on the role of each module.

Another interesting location is identified at the nineteenth position in the intermittent ranking. While this location does not score as highly as others, it does present a well hidden attack vector. The *o_axi_router_rd_en* is a simple control flag used to indicate that a read operation is taking place within the AXI routing module. This signal is assumed to toggle synchronously with the presence of data on the AXI bus, so if it were interfered with or delayed, messages could be dropped or transactions could be slowed down.

6.2.3.3 Interconnect Locations

The final and most abstract locations found by the tool are the interconnects used by high level modules to route signals between submodules. Since there is no concept of component purpose available to the analysis tool, these signals are mostly selected to highlight what is assumed to be the most important connection edges from module to module, making them good pinch points for small Trojans that simply enable or disable signals. These locations come mostly just as recommendations as to what signals to attack, as no combinational logic is associated with them.

The first eleven interconnects identified occur in the ship_control_pl_wrapper module, which contains almost all of the non-ARM processor logic in this design and is where almost every interface module is instantiated. Again, the signals selected are all related to the AXI bus, although the specific signals selected come in three main types. The first type is the address used by the AXI controller to specify which device it will read from or write to. The second type is an enable flag signal used to set the read or write mode of AXI transactions. The final type is the actual data on the AXI bus, specifically write data being sent out to peripherals. Each of these interconnect locations could be used to implement almost any type of behavior modification Trojan, as small changes to any control flag value here could disrupt communication or changes to write values could cause changes in outward behavior in the peripherals.

Another interesting interconnect signal is identified at ranking position twelve. Here, the w_upm_mask signal is used to identify specific bits within messages that mark important status information. This status information is used to trigger certain alarms within the AXI routing components in the event of errors or if certain devices need priority handling. While this mask value is relatively static, false triggering of alarm signals (achieved by altering the mask) could be a good way to degrade device performance, causing cycles to be wasted managing fake errors.

Most of the remaining locations (thirteen through nineteen) identified are used for the ARM processor to peripheral device communications within each interface module. Specifically, the signals identified are registers used to store information coming from the ARM processor (denoted with esm_rdata) to a peripheral or used to store information going from a peripheral to the processor (denoted with ism_rdata). Here, false information could be fed into the processor (such as bad GPS coordinates), or commands from the processor to peripheral ignored (such as ignoring the propeller halt command). These locations are similar to the top destructive location in that alterations here do not directly disrupt communication protocols, but affect the peripherals themselves more directly. This could allow for the propagation of falsified data to other parts of the design, and thus more fine grained targeting of effects.

6.3 Observations

Considering that this is an automated analysis that uses no prior knowledge of the design to suggest locations for attacks, the results of these experiments are quite promising. In both the MIPS processor and the navigational controller, the tool found useful locations for all three types of Trojan location category. Additionally, the tool was able to find locations that were at least similar to those selected by hand on the navigation controller, as well as recommend several novel locations for potential Trojans. That said, there are some important considerations that should be made as a result of the observations taken from these experiments.

The first, and potentially most obvious, is that the heuristics strongly favor communication related components when evaluating all three location types. This is likely due to

the large emphasis that the base signal heuristics (impact, susceptibility, and controllability) place on the connectivity of signals. Since the signals and statements are selected to be Pareto optimal, this means that either of the two base metrics in each selection can cause a signal to dominate other. As a result, signals involved in communication will usually have high impact scores, causing them to dominate many other signals. While this issue is not too apparent in the MIPS processor, the navigation controller logic is composed of many communication components, making it difficult for locations involved in calculation logic to surface. It should be mentioned that much of the navigation controller's logic is mainly for communication and contains proportionally less computational logic, which also influenced the results seen above.

One potential issue with this scheme also lies in the detection of duplicate signals that serve more or less the same function in several modules. Since these signals are often awarded similar (if not identical) scores, it is hard to know if one specific signal and location can be selected to act as a representation of all similar sites. Future investigation will evaluate these groups and determine if some sort of grouping system can be used to select the best of a set of similar locations, breaking the ties in scoring that they create.

Another potential point of improvement could come from enforcing some sort of "signal distance" metric to ensure that signals selected for the top rankings are not too close to one another. In most designs, the movement of a Trojan a few gate levels forward or backward from its position will not change its effects too much, so it is possible that at the RTL a Trojan identified at one location would be observed as very similar to a Trojan inserted at an assignment nearby in the upstream or downstream cone of influence. For this reason, it may not be of benefit to select signals that are very close to one another, as Trojan effects in that area may be the same. Selecting the better of two close locations could help to ensure more variety and signal coverage in the rankings generated.

In general, the locations selected by this tool are quite useful, in that they find interesting locations that usually have good value for creating novel Trojans. Overall, each selected site, including those not described above but included in the appendix, points to a location that would be a good point of insertion for a Trojan. That said, the user must still determine how an attacker would be most likely to insert a Trojan along with its likely effects. While

a brute force method for automatically inserting Trojan logic elements (malicious AND, OR and MUX gates) into designs is being investigated, more work in this area will be necessary to find more efficient and effective solutions.

Chapter 7
CONCLUSION

Electronic device security is now a major concern for industrial, consumer, and military applications, so it is necessary that new methods of securing products be developed. This work offers an approach to automatically locate potentially vulnerable sites within a register-transfer-level (RTL) hardware design before it is compiled into a gate netlist or implemented in a physical system. This aims to enable researchers and designers to better understand how designs could be exploited, as well as help enable investigation into how designs can be hardened to attack. This approach will be included in a larger tool-chain, described in Section 2.2.1, with the goal of creating an end-to-end solution for securing complex digital designs.

7.1 Design Strategies

From these results, several lessons can be learned about design approaches and how better RTL code could help increase the resilience of a design to Trojan insertion. While none of the following concepts can guarantee that no Trojan can be placed within a design, following some guidelines could help make Trojans difficult to insert or more obvious if they are inserted.

7.1.1 Least Privilege Policy

Like in most information systems, it is important to consider what components of a system need access to which pieces of information in order to function. While this is a standard practice in information technology systems and most software packages [26], it is important to consider the policy of least privilege at the hardware design level as well. An important example of a violation of this policy was detailed in Section 6.1.2, where a Trojan location was identified that would not have existed within this design had the designer adhered more closely to this concept. Here, following some best practices from the field of cybersecurity

can help to limit some of these unnecessary risks. In hardware design, this would likely come in the form of strict module specifications that only allow module inputs and outputs to interact with the bare minimum information necessary for that module's functionality. Following this practice could cause other side affects, however, since limiting information flow in and out of modules would likely lead to a flattening of the design hierarchy into more parallel modules within a top-level module. This would result in more interconnect signals and more potential interconnect Trojan locations, so trade-offs in this area would need to be considered.

Due to their nature, interconnect Trojans are likely easier to find, since their location can be more easily predicted based on hierarchical structure alone. Should the approach described here or one developed later be highly successful in locating interconnect Trojans, it may be possible that changes in design practices like this could lead aid Trojan detection approaches as well.

7.1.2 Interconnect Vulnerabilities

Designs that utilize modules with several connected submodules may benefit from extra protections applied to the submodule interconnect signals, such as JTAG testing hardware [27]. Placing instruments, such as JTAG scan cells, on these interconnects can offer two potential deterrents for an attacker. First, the presence of a JTAG scan cell, or other sensor, on an interconnection signal indicates that this signal will be monitored during testing, so alterations to this signal could be easily detected. The drawback of this is that this does indicate this signal is important to the attacker. Second, since JTAG scan cells require the insertion of a register component, timing analysis will become more constrained in the area around the cell. This means that some Trojans that include extra logic may violate timing rules on a certain path, causing further investigation and potential discovery of the Trojan by a designer later during verification stages. It should be noted that some Trojans can take JTAG behavior into account and disable themselves while a device is in test mode for this reason.

7.1.3 Standardization

Clear and consistent coding practices can help any software product be more maintainable and understandable, and the same is true of RTL code. With this comes the opportunity for designers to notice deviations from design patterns or improperly specified components. While the same practices used in software design may not be directly applicable to hardware specification, some standards do exist to help with hardware code reuse and maintainability. It is recommended that any entity concerned with hardware security utilize some standard for the implementation of a design. One such example of this is the *Reuse Methodology Manual* [28]. In the case of the navigation controller, standardized signal naming conventions helped to identify groups of similar locations even when potential Trojan sites were discovered in separate modules.

7.2 Closing Thoughts

There has been a fair amount of work on the Trojan detection problem; that is, many methods used to identify Trojans already placed in a circuit have already been created and evaluated. This previous work, discussed more in Chapter 3, provides some insight into how Trojans have been located in the past and provides some lessons regarding how hardware attacks can be conceptualized in practice. That said, most of this work is at the gate-level and relies on the strict rules available to low-level abstractions. For the analysis tool presented in this thesis, new methods that relied only on RTL code needed to be developed.

From the experiments presented in Chapter 6, it is clear that automatic identification of potential behavior modification Trojan sites is not only possible but quite useful for analyzing designs. The approach presented here is capable of identifying three major classifications of potential Trojan locations based an a set of base metrics derived from the source code of the RTL design, as described in Chapter 5. Using these metrics to first identify interesting signals to target, the tool gains some understanding of how signals are used within the design with no prior knowledge of the purpose of each component. Once selected, these signals are investigated further, where specific lines pertaining to them in the source code are selected based on a separate set of metrics to describe how dangerous a statement would be in a specific context. Together, these two steps enable the creation of a design level set

of rankings that can quickly identify and partially describe what kinds of attack are best suited for interesting locations. While this tool does not provide suggestions regarding how to implement an attack at any location, work for inserting Trojan gates to emulate real-world Trojans is on-going.

Future work will involve expanding the syntactical functionality of the tool to process more complex design elements, as well as include new classifications of Trojans and the metrics necessary for finding them. At the time of writing, only a subset of the Verilog standard can be parsed. Expanding the tool to manage more complex structures as well as integrate lower-level design information, such as library cell information or timing specifications, could enable more robust and accurate analysis. Additional classification methods will also need to be developed to cover other types of Trojans, such as those that leak information described in Section 2.1.1. Since those Trojan types differ in implementation and may make use of other design details (such as physical layout information), more investigation will be necessary to identify useful metrics to automatically identify them.

While the method presented here is limited in what types of sites it is capable of finding, it achieves its goal of locating potential behavior modification Trojan locations quite well. Evaluation of two separate designs with different functionalities has verified this approach's ability to automatically identify interesting locations for Trojans. Additionally, these experiments have proven that the metrics the tool relies on can be used to identify locations that an experienced designer, or attacker, might find by hand. This will allow for the creation of a closed loop system to select locations, insert Trojans, emulate their behaviors, and evaluate counter-measure instrumentation without the need for complete hand evaluation of design elements. Together, all these elements will help secure electronic components and provide safer and more reliable electronics.

APPENDIX A

A.1 16-Bit MIPS Location Rankings

Rank	Module	Signal	Score	Line	File
1	IF_stage	pc	2.017	37	IF_stage.v
2	alu	r	1.982	38	alu.v
3	alu	r	1.982	28	alu.v
4	alu	r	1.982	40	alu.v
5	alu	r	1.982	26	alu.v
6	alu	r	1.806	36	alu.v
7	IF_stage	pc	1.602	39	IF_stage.v
8	WB_stage	reg_write_data	1.205	39	WB_stage.v
9	alu	r	1.204	34	alu.v
10	alu	r	1.204	32	alu.v

Table A.1: Destructive Locations for 16-Bit MIPS Processor

Rank	Module	Signal	Score	Line	File
1	ID_stage	ex_alu_cmd	0.602	203	ID_stage.v
2	register_file	reg_write_dest	0.602	54	register_file.v
3	ID_stage	ex_alu_cmd	0.602	196	ID_stage.v
4	ID_stage	ex_alu_cmd	0.602	189	ID_stage.v
5	ID_stage	ex_alu_cmd	0.602	182	ID_stage.v
6	ID_stage	ex_alu_cmd	0.602	175	ID_stage.v
7	ID_stage	ex_alu_cmd	0.602	168	ID_stage.v
8	ID_stage	ex_alu_cmd	0.602	161	ID_stage.v
9	ID_stage	ex_alu_cmd	0.602	154	ID_stage.v
10	ID_stage	ex_alu_cmd	0.602	147	ID_stage.v

Table A.2: Intermittent Locations for 16-Bit MIPS Processor

Rank	Module	Signal	Score	Line	File
1	mips_16_core_top	ID_pipeline_reg_out	1.884	29	mips_16_core_top.v
2	mips_16_core_top	instruction	1.869	28	mips_16_core_top.v
3	mips_16_core_top	decoding_op_src2	1.415	38	mips_16_core_top.v
4	mips_16_core_top	reg_read_addr_2	1.365	34	mips_16_core_top.v
5	mips_16_core_top	branch_offset_imm	1.230	26	mips_16_core_top.v
6	mips_16_core_top	pipeline_stall_n	1.146	25	mips_16_core_top.v
7	mips_16_core_top	reg_read_addr_1	1.092	33	mips_16_core_top.v
8	mips_16_core_top	decoding_op_src1	1.079	37	mips_16_core_top.v
9	mips_16_core_top	reg_read_data_2	1.068	36	mips_16_core_top.v
10	mips_16_core_top	EX_pipeline_reg_out	1.034	30	mips_16_core_top.v

Table A.3: Interconnect Locations for 16-Bit MIPS Processor

A.2 User Inserted Navigation Controller Trojans

Trojan Name	Attack Type	Target Module and Signals	Trojan Details
T1a	Joystick Behavior Modifier	**joystick_if** r_spi_byte_cntr r_tag_counter	Activation: Timer set to activate trigger signal after 20 seconds. Effect: Disables input tied to emergency breaking function. Allows other controls to continue operating uninhibited.
T1b	Reliability Attack	**spi_if** o_sclk	Activation: Timer periodically toggles on and off trigger signals. Effect: Rudder and propeller will move on their own every few seconds, ignoring input from user. Control is given back to user, then cycle repeats.
T1c	SPI Port Take-Over	**ship_control_top** O_PMODB7 O_PMODB8 O_PMODB10	Activation: Always active, Trojan signals provided by unused pins. Effect: Allows for input to be taken from another Joystick connected to the controller. Essentially provides access for control from outside source.
T4a	Active Width Take-Over	**propeller_servo_if** **rudder_servo_if** r_eff_rd_valid_reg r_eff_rd_data_reg	Activation: Always active, take over signals provided by extra peripheral inputs. Effect: Supplies rudder movement and propeller speed commands through side channel of other board inputs.
T5a	Adjusted Pulse Reliability Attack	**pwm_if** r_inactive_cntr r_active_cntr	Activation: Timer activates trigger signal after 30 Seconds. Effect: Forward motion of propeller stops, rudder range of motion becomes limited.
T5c	Adjusted Pulse Behavior Modifier	**pwm_if** r_xfer_state	Activation: Timer forces internal counters to hit maximum value early. Effect: Propeller stops functioning and rudder's range of motion is removed.

Table A.4: Behavior Modification Trojans for Navigation Controller

A.3 Navigation Controller Location Rankings

Rank	Module	Signal Name	Signal Function	Score
1	axi_router	r_axi_router_int-_data	Register for storing data in AXI communication	3.177
2	pwm_if	r_active_time	Upper limit for transfer time in PWM output	2.947
3	gps_if	o_m0n_axi_rdata	Output response for data read in AXI transactions	2.709
4	keyboard_if	o_m0n_axi_rdata	Output response for data read in AXI transactions	2.709
5	joystick_if	o_m0n_axi_rdata	Output response for data read in AXI transactions	2.709
6	axi_router	o_axi_router_wdata	Location for writes to AXI bus	2.594
7	propeller_servo_if	o_m0n_axi_rdata	Output response for data read in AXI transactions	2.408
8	rudder_servo_if	o_m0n_axi_rdata	Output response for data read in AXI transactions	2.408
9	propeller_servo_if	o_m0n_axi_rdata	Output response for data read in AXI transactions	2.408
10	rudder_servo_if	o_m0n_axi_rdata	Output response for data read in AXI transactions	2.408
11	uart_if	r_clock_trn_scale-_cntr	Counter to manage transmission state information in UART	2.334
12	joystick_if	r_spi_byte_cntr	Counts number of bytes received from SPI	2.325
13	gps_if	r_urt_word_cntr	Counts number of bytes received from UART	2.149
14	register_rw	o_data	Output register	1.978
15	spi_if	r_transfer_word-_number	Countdown of words remaining in transfer	1.929
16	ps2_if	r_transfer_bit_cntr	Marks which bit is being transfered in PS/2 protocol	1.857
17	ship_control_pl-_wrapper	w_trj_ins_act_mon	JTAG interface input	1.820
18	gps_control_code	o_m0n_axi_rdata	Output response for data read in AXI transactions	1.806
19	register_ro	o_proc_data	Data output on read-only system status registers	1.806
20	ET_Security-_Project_ET_0	VJI___VIR-_OUTPUT	IJTAG Boundry scan output	0.615

Table A.5: Destructive Locations for Navigational Controller

Rank	Module	Signal Name	Signal Function	Score
1	spi_if	r_mosi_cpha0	SPI master to slave clock phase setting. Ensures data is transmitted and read at correct edge of clock	1.538
2	uart_if	r_stop_bit-_transmitted	Flag to specify if stop bit (end of signal pulse) was correctly created	1.518
3	gps_if	o_m0n_axi_rresp	Response to incoming AXI message. Control signal used to ensure complete transmission	1.506
4	keyboard_if	o_m0n_axi_rresp	Response to incoming AXI message	1.506
5	joystick_if	o_m0n_axi_rresp	RResponse to incoming AXI message	1.506
6	spi_if	r_mosi_cpha0	SPI master to slave clock phase setting	1.505
7	uart_if	r_stop_bit-_transmitted	Flag to specify if stop bit (end of signal pulse) was correctly created	1.505
8	spi_if	r_mosi_cpha0	SPI master to slave clock phase setting	1.355
9	gps_control_code	o_m0n_axi_rresp	Response to incoming AXI message	1.177
10	propeller_servo_if	o_m0n_axi_rresp	Response to incoming AXI message	0.993
11	rudder_servo_if	o_m0n_axi_rresp	Response to incoming AXI message	0.993
12	gps_if	o_m0n_axi_rresp	Response to incoming AXI message	0.993
13	keyboard_if	o_m0n_axi_rresp	Response to incoming AXI message	0.993
14	gps_control_code	o_m0n_axi_rresp	Response to incoming AXI message	0.993
15	joystick_if	o_m0n_axi_rresp	Response to incoming AXI message	0.993
16	gps_control_code	o_m0n_axi_rresp	Response to incoming AXI message	0.993
17	propeller_servo_if	o_m0n_axi_rresp	Response to incoming AXI message	0.858
18	rudder_servo_if	o_m0n_axi_rresp	Response to incoming AXI message	0.858
19	axi_router	o_axi_router_rd_en	Flag to specify that a read request is present on the AXI bus	0.605
20	axi_router	o_axi_router_rd_en	Flag to specify that a read request is present on the AXI bus	0.301

Table A.6: Intermittent Locations for Navigational Controller

Rank	Module	Signal Name	Signal Function	Score
1	ship_control_pl-_wrapper	w_m00_axi_router-_addr	Target peripheral address for AXI transmission in AXI port 0	2.086
2	ship_control_pl-_wrapper	w_m00_axi_router-_wr_en	Flag to specify slave device as ready for write from AXI	2.024
3	ship_control_pl-_wrapper	w_m01_axi_router-_addr	AXI Target peripheral address on port 1	1.973
4	ship_control_pl-_wrapper	w_m01_axi_router-_addr	AXI Target peripheral address on port 1	1.930
5	ship_control_pl-_wrapper	w_m01_axi_router-_wr_en	AXI slave device ready for write	1.835
6	ship_control_pl-_wrapper	w_m00_axi_router-_addr	AXI Target peripheral address on port 0	1.810
7	ship_control_pl-_wrapper	w_m00_axi_router-_addr	AXI Target peripheral address on port 0	1.810
8	ship_control_pl-_wrapper	w_m01_axi_router-_wr_en	AXI slave device ready for write	1.805
9	ship_control_pl-_wrapper	w_m01_axi_router-_rd_en	AXI slave device ready for read	1.768
10	ship_control_pl-_wrapper	w_m01_axi_router-_addr	AXI Target peripheral on port 1	1.761
11	ship_control_pl-_wrapper	w_m00_axi_router-_wdata	Data to write on AXI port 0	1.751
12	axi_router	w_upm_mask	Mask to select status bits from AXI message. Used to raise alarm signals when certain bits are detected.	1.519
13	propeller_servo_if	w_esm_rdata	Data from embedded microcontroller sent to propeller logic (sent over AXI)	1.519
14	gps_if	w_ism_rdata	Data sent to microcontroller from GPS sensor (sent over AXI)	1.519
15	keyboard_if	w_ism_rdata	Data sent to microcontroller from keyboard	1.519
16	rudder_servo_if	w_esm_rdata	Data sent from microcontroller to rudder logic	1.519
17	joystick_if	w_ism_rdata	Data sent to microcontroller from joystick logic	1.519
18	gps_if	w_esm_rdata	Data sent from microcontroller to GPS sensor	1.519
19	joystick_if	w_esm_rdata	Data sent from microcontroller to joystick servo	1.519
20	axi_router	w_rsr_status_updated	Flag to specify if AXI router status register has been updated	1.000

Table A.7: Interconnect Locations for Navigational Controller

BIBLIOGRAPHY

[1] S. Adee, "The hunt for the kill switch," *IEEE Spectrum*, vol. 45, no. 5, pp. 34–39, May 2008. 4

[2] Xiaoxiao Wang, M. Tehranipoor, and J. Plusquellic, "Detecting malicious inclusions in secure hardware: Challenges and solutions," in *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, June 2008, pp. 15–19. 5

[3] R. S. Chakraborty, S. Narasimhan, and S. Bhunia, "Hardware trojan: Threats and emerging solutions," in *2009 IEEE International High Level Design Validation and Test Workshop*, Nov 2009, pp. 166–171. 5

[4] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 10–25, Jan 2010. 5

[5] A. Crouch, E. Hunter, and P. L. Levin, "Enabling hardware trojan detection and prevention through emulation," in *2018 IEEE International Symposium on Technologies for Homeland Security (HST)*, 2018, pp. 1–5. 7

[6] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware trojan design and implementation," in *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, 2009, pp. 50–57. 7

[7] "Jaspergold security path verification app." [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html 9, 14

[8] K. Basu and P. Mishra, "Efficient trace signal selection for post silicon validation and debug," in *2011 24th Internatioal Conference on VLSI Design*, Jan 2011, pp. 352–357. 12

[9] K. Rahmani and P. Mishra, "Efficient signal selection using fine-grained combination of scan and trace buffers," in *2013 26th International Conference on VLSI Design and 2013 12th International Conference on Embedded Systems*, Jan 2013, pp. 308–313. 12

[10] M. Li and A. Davoodi, "A hybrid approach for fast and accurate trace signal selection for post-silicon debug," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 7, pp. 1081–1094, July 2014. 12, 13

[11] J. Yang and N. A. Touba, "Efficient trace signal selection for silicon debug by error transmission analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 3, pp. 442–446, March 2012. 13

[12] A. Nahiyan, M. Sadi, R. Vittal, G. Contreras, D. Forte, and M. Tehranipoor, "Hardware trojan detection through information flow security verification," in *2017 IEEE International Test Conference (ITC)*, Oct 2017, pp. 1–10. 13, 14, 15

[13] T. Le, J. Di, M. Tehranipoor, and L. Wang, "Tracking data flow at gate-level through structural," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, May 2016, pp. 185–189. 13, 16

[14] W. Hu, B. Mao, J. Oberg, and R. Kastner, "Detecting hardware trojans with gate-level information-flow tracking," *Computer*, vol. 49, no. 8, pp. 44–52, Aug 2016. 14

[15] Y. Jin, B. Yang, and Y. Makris, "Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, June 2013, pp. 99–106. 14

[16] H. Salmani and M. Tehranipoor, "Trust-hub.org." [Online]. Available: https://trust-hub.org/benchmarks/trojan 15

[17] V. Jyothi, P. Krishnamurthy, F. Khorrami, and R. Karri, "Taint: Tool for automated insertion of trojans," in *2017 IEEE International Conference on Computer Design (ICCD)*, Nov 2017, pp. 545–548. 15

[18] H. Salmani and M. M. Tehranipoor, "Vulnerability analysis of a circuit layout to hardware trojan insertion," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1214–1225, June 2016. 16

[19] I. Ghosh, A. Raghunathan, and N. K. Jha, "A design-for-testability technique for register-transfer level circuits using control/data flow extraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 8, pp. 706–723, Aug 1998. 16, 17

[20] Zhigang Yin, Yinghua Min, and Xiaowei Li, "An approach to rtl fault extraction and test generation," in *Proceedings 10th Asian Test Symposium*, Nov 2001, pp. 219–224. 17

[21] T. Strauch, "A novel rtl atpg model based on gate inherent faults (gif-po) of complex gates," *arXiv.org*, Dec 15 2016, copyright - © 2016. This work is published under http://arxiv.org/licenses/nonexclusive-distrib/1.0/ (the "License"). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2019-04-13. 17

[22] "Verilator." [Online]. Available: https://www.veripool.org/wiki/verilator 19

[23] "Verilog-perl." [Online]. Available: https://www.veripool.org/wiki/verilog-perl 20

[24] K. D. Cooper and L. Torczon, "Chapter 5 - intermediate representations," in *Engineering a Compiler (Second Edition)*, second edition ed., K. D. Cooper and L. Torczon, Eds. Boston: Morgan Kaufmann, 2012, pp. 221 – 268. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780120884780000050 23

[25] Doyya, "Educational 16-bit mips processor," Mar 2012. [Online]. Available: https://opencores.org/projects/mips_16 51, 52

[26] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, no. 7, p. 388–402, Jan 1974. 55, 66

[27] "IEEE Standard for Test Access Port and Boundary-Scan Architecture," Institute of Electrical and Electronics Engineers, Standard, Feb. 2013. 67

[28] M. Keating and P. Bricaud, *Reuse methodology manual: for system-on-a-chip designs*, 3rd ed. Kluwer Academic, 2002. 68

[29] "Welcome to python.org." [Online]. Available: https://www.python.org/