University of California
Santa Barbara

Verification Techniques for Hardware Security

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Nicole Chan Fern

Committee in charge:

Professor Kwang-Ting (Tim) Cheng, Chair
Professor Forrest Brewer
Professor Tim Sherwood
Dr. Çetin Kaya Koç, Lecturer
Professor Huijia (Rachel) Lin

June 2016

The Dissertation of Nicole Chan Fern is approved.

Professor Forrest Brewer

Professor Tim Sherwood

Dr. Çetin Kaya Koç, Lecturer

Professor Huijia (Rachel) Lin

Professor Kwang-Ting (Tim) Cheng, Committee Chair

June 2016

Verification Techniques for Hardware Security

by

Nicole Chan Fern

Acknowledgements

I would like to acknowledge my advisor, Professor Kwang-Ting (Tim) Cheng for the support and guidance he has provided over the past 5 years. Under his mentorship I have been given the opportunity to find and define my own research question, and explore the area of hardware security. Through this process I have matured as an independent researcher and thinker, and for this I will be forever greatful.

I would also like to thank my collaborators Ismail San and Professor Çetin Koç for introducing me to the world of cryptography! Peter Lisherness has provided valuable advice about being a graduate student at UCSB in addition to facilitating an internship opportunity at Apple. My lab-mates Amirali Ghofrani, Miguel Lastras, Fan Lin, C.K. Hsu, Rui Wu, Chong Huang, Leilai Shao, and Yuyang Wang provide continuing company and support.

This work was financially supported by the National Science Foundation as well as the Semiconductor Research Corporation. I would also like to thank the Xilinx University Program for their generous donation of several FPGA boards.

Finally, I would like to thank my family and friends. Without their support, especially my husband, Jacob, and my parents, I would not have been able to complete this work.

Curriculum Vitæ
Nicole Chan Fern

Education

2016	Ph.D. in Computer Engineering (Expected), University of California, Santa Barbara.
2013	M.S. in Computer Engineering, University of California, Santa Barbara.
2011	B.E. in Electrical Engineering, The Cooper Union for the Advancement of Science and Art

Work Experience

2013	Hardware Verification Engineer in the Silicon Engineering Group at Apple Inc in Cupertino, California
2012	Software Security Intern in the Cisco Security and Government Group at Cisco Systems in Knoxville, Tennessee

Publications

- **N. Fern**, I. San, Ç. Koç, and K-T. Cheng. "Hardware Trojans in Incompletely Specified On-chip Bus Systems", Design, Automation, Test in Europe (DATE) Conference, 2016.
- **N. Fern** and K-T. Cheng. "Detecting Hardware Trojans in Unspecified Functionality Using Mutation Testing", International Conference on Computer Aided Design (ICCAD), 2015.
- **N. Fern**, S. Kulkarni, and K-T. Cheng. "Hardware Trojans Hidden in RTL Don't Cares - Automated Insertion and Prevention Methodologies", International Test Conference (ITC), 2015.
- **N. Lesperance***, S. Kulkarni, and K-T. Cheng, "Hardware Trojan Detection Using Exhaustive Testing of *k*-bit Subspaces", Asia South-Pacific Design Automation Conference (ASP-DAC), 2015.
- P. Lisherness, **N. Lesperance***, and K-T. Cheng, "Mutation Analysis with Coverage Discounting", Design, Automation Test in Europe (DATE), 2013.
- **N. Lesperance***, P. Lisherness, and K-T. Cheng, "Coverage Discounting: Improved Testbench Qualification by Combining Mutation Analysis with Functional Coverage", SRC TechCon, 2013.
- **N. Lesperance***, M. Leece, S. Matsumoto, M. Korbel, K. Lei, and Z. Dodds, "PixelLaser: Computing Range from Monocular Texture", Advances in Visual Computing, LNCS 6455, pp. 151-160, 2010.

*Published under maiden name

Activities and Academic Honors

- Received 3rd place in the 2016 Test Technology Technical Council's E.J. McCluskey Doctoral Thesis Competition Semi-Finals
- Awarded the ECE Department Dissertation Fellowship for Spring 2016
- Teaching Assistant at UCSB for 6 academic quarters for courses requiring a weekly lecture (ECE154, Computer Architecture) and lab-based courses (ECE152A/B, students created hardware designs using a combination of FPGA boards and discrete circuit components) (2012 – 2015)
- Selected to teach a seminar on Matlab and also provide supplementary presentation of topics in signal processing and control systems (2010 – 2011)
- Winner of the Leon Machiz Prize for excellence in electrical engineering (2011)
- Winner of the Class of 1907 Award for the best enrolled or graduating student in calculus (2011)
- Winner of the Jesse Sherman Book Award for Outstanding Average in Electrical Engineering (2010, 2011)
- Dean's List at Cooper Union (8 consecutive semesters)

Abstract

Verification Techniques for Hardware Security

by

Nicole Chan Fern

Verification for hardware security has become increasingly important in recent years as our infrastructure is heavily dependent on electronic systems. Traditional verification methods and metrics attempt to answer the question: does my design correctly perform the intended specified functionality? The question this dissertation addresses is: does my design perform malicious functionality in addition to the intended functionality? Malicious functionality inserted into a chip is called a *Hardware Trojan*.

This work is devoted to developing both new threat models and detection methodologies for a less studied but extremely stealthy class of Trojan: Trojans which do not rely on rare triggering conditions to stay hidden, but instead only alter the logic functions of design signals which have unspecified behavior, meaning the Trojan never violates the design specification.

The main contributions of this work are 1) precise definitions for dangerous unspecified functionality in terms of information leakage and several methods to identify such functionality, 2) satisfiability-based formal methods to test potentially dangerous unspecified functionality for the existence of Trojans, and 3) numerous examples of how the proposed Trojans can completely undermine system security if inserted in on-chip bus systems, communication controllers, and encryption IP.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Problem Scope

Electronic devices and systems are now ubiquitous and influence key aspects of modern life such as freedom of speech, privacy, finance, science, and art. Concern about the security and reliability of our electronic systems and infrastructure is at an all-time high. Failure to protect these systems results in not only great financial hardship, but can give rise to life-threatening situations as seen when patient medical records became inaccessible within all MedStar Health hospitals after their computer networks were attacked by ransomware [1], and when cyber-attacks left nearly a quarter million people in Western Ukraine without power or heat for several hours in the dead of winter [2].

Securing electronic systems is extremely difficult because an attacker only needs to find and exploit a single weakness to perform malicious actions whereas defenders must protect the system against an infinite set of possible vulnerabilities to ensure it is secure. To address the attacker/defender asymmetry, security research has focused on developing threat models which classify attacker capabilities, motivations, and goals, and creating a taxonomy of vulnerabilities to address. For example, Ravi *et al.* [3] give an overview of the threats embedded systems face and the existing catalogue of mitigation techniques.

Security techniques target detection/prevention of a class of vulnerability given a specific threat model. Identifying and disclosing (in a responsible manner) new vulnerabilities to the security community is essential to the process of making systems more secure. The task of enumerating and addressing all threat models and vulnerabilities is never complete, but this dissertation contributes to this task by discovering and developing mitigation strategies for a novel class of vulnerability: the opportunity in most hardware designs for an attacker to hide malicious behavior entirely within **unspecified functionality**.

Verification and testing is a major bottleneck in hardware design, and as design complexity increases, so does the productivity gap between design and verification [4]. It is estimated that over 70% of hardware development resources are consumed by the verification task. To address this challenge, commercial verification tools and academic research developed over the past several decades has focused on increasing confidence in the correctness of *specified functionality*. Important design behavior is modeled then various tools and methods are used to analyze the implementation at various stages in the chip design process to ensure the implementation always matches the golden reference model.

Behavior which is not modeled will not be verified by existing methods, meaning any security vulnerabilities occurring within unspecified functionality will go unnoticed. In modern complex hardware designs, which now contain several billion transistors, there always exists unspecified functionality. It is simply impossible to enumerate what the desired state of several billion transistors or logic gates should be at every cycle when behavior depends not only on the internal state of the system, but also on external input from the environment the device is embedded in. It is only feasible to model and verify aspects of the design with functional importance.

Because behaviors at a good fraction of signals for many operational cycles are unspecified, an **attacker can modify this functionality with impunity without detection** by existing verification methods. This thesis explores how an attacker can embed malicious behavior *completely* within unspecified functionality whereas most related research explores how to detect violations of specified behavior occurring under extremely rare conditions, where the main challenge is identifying these conditions.

Malicious functionality inserted into a chip is called a *Hardware Trojan*. Hardware Trojans are a major concern for both semiconductor design houses and the U.S. government due to the complexity of the chip design ecosystem [5, 6]. Economic factors dictate that the design, manufacturing, testing, and deployment of silicon chips is spread across many companies and countries with different and often conflicting goals and interests. If a single party involved deems it advantageous to insert a Hardware Trojan, the consequences can be catastrophic.

Goals of previously proposed Hardware Trojans range from denial of service attacks such as forcing premature circuit aging [7] and on-chip system bus deadlock [8] to subtler attacks which attempt to gain undetected privileged access on a system [9], leak secret information through power or timing side channels [10], or weaken random number generator output [11].

The process of transforming a document specifying how a chip should behave to physical silicon is extremely complex, involving thousands of engineers and an entire ecosystem of design tools and fabrication services. Figure 1.1 gives an overview of the design life cycle, and the Trojan threats faced at each stage. Hardware Trojans are extremely hard to defend against because they give the attacker the ability to modify the circuit at any stage in the design lifecycle to introduce new vulnerabilities, not just identify and exploit those already existing in the system [12, 13].

Before fabrication (pre-silicon), the design is modeled using an executable Register

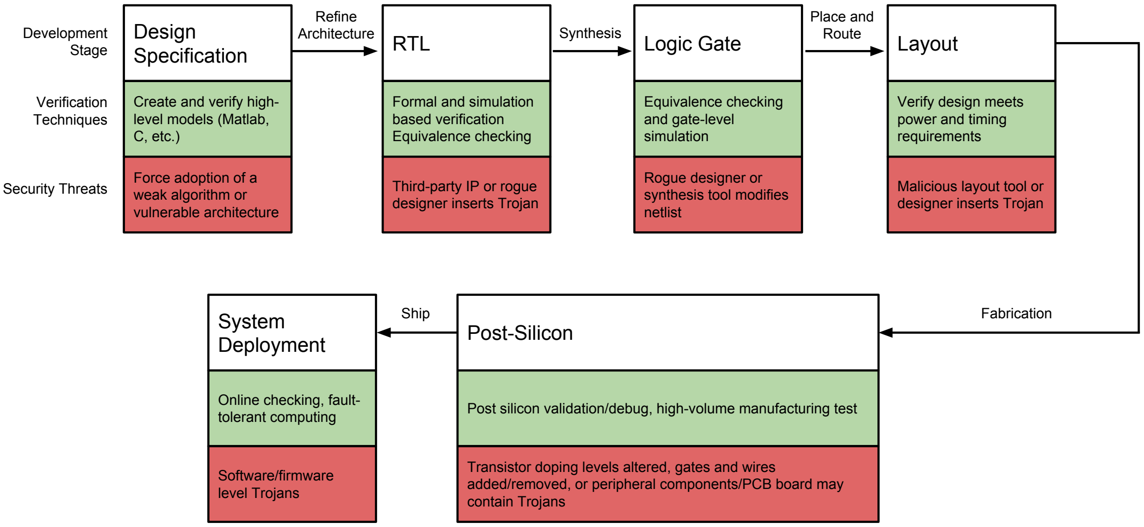

Development Stage
Verification Techniques
Security Threats
Design Specification
Create and verify high-level models (Matlab, C, etc.)
Force adoption of a weak algorithm or vulnerable architecture
Refine Architecture
RTL
Formal and simulation based verification Equivalence checking
Third-party IP or rogue designer inserts Trojan
Synthesis
Logic Gate
Equivalence checking and gate-level simulation
Rogue designer or synthesis tool modifies netlist
Place and Route
Layout
Verify design meets power and timing requirements
Malicious layout tool or designer inserts Trojan
Fabrication
Post-Silicon
Post silicon validation/debug, high-volume manufacturing test
Transistor doping levels altered, gates and wires added/removed, or peripheral components/PCB board may contain Trojans
Ship
System Deployment
Online checking, fault-tolerant computing
Software/firmware level Trojans

Figure 1.1: Threats Present in the Hardware Development Lifecycle

Transfer Level (RTL) hardware description language such as Verilog or VHDL and the model is thoroughly analyzed and tested for errors. A modern design can contain several hundred Verilog/VHDL modules, some of which are provided by external (potentially untrusted) 3rd parties. Before fabrication, an RTL model must be transformed into a physical design layout, which specifies how transistors are to be arranged on the silicon wafer. This process is performed by a mix of extremely complex Electronic Design Automation (EDA) tools (which are potentially untrusted) and manual effort.

This thesis focuses on detecting Hardware Trojans inserted in the Verilog/VHDL code, which in some aspects is more challenging than the detection of Trojans inserted in the gate-level netlist, physical design layout, or during or after fabrication. The EDA ecosystem contains mature tools to ensure that once an RTL design is considered “golden”, all subsequent transformations produce designs whose behavior matches that of the golden model. This work increases confidence that the RTL reference model, used as a guidepost for the rest of design development, itself is Trojan-free, which is essential to detecting any Trojans inserted at subsequent stages in the design lifecycle.

Many of the Trojans proposed in literature hide from the verification effort using extremely rare triggering conditions. Examples of stealthy Trojan triggers are counters, which wait thousands of cycles before changing circuit functionality, or pattern recognizers, which wait for a "magic" value or sequence to appear on the system bus [14] or as plaintext input in cryptographic hardware [15, 16, 17]. Trojans with these triggering mechanisms generally deploy a payload which clearly violates system specifications (such as causing a fault during a cryptographic computation).

Existing pre-silicon Trojan detection methods assume Trojans violate the design specification under carefully crafted rare triggering conditions, and focus on identifying the structure of this triggering circuitry in the RTL code or gate-level netlist [18, 19, 20, 21]. The Trojans proposed in this dissertation do not rely on rare triggering conditions to stay hidden, but instead **only** alter the logic functions of design signals which have **unspecified behavior**, meaning the Trojan **never** violates the design specification.

Addressing this Trojan type requires a different approach from both existing Trojan detection and hardware verification methods, since traditional verification and testing excludes analysis of unspecified functionality for efficiency, and focuses primarily on conformance checking. Identifying unspecified functionality that either contains a Hardware Trojan or could be exploited by an attacker in the future is difficult because by definition unspecified functionality is not modeled or known by the design/verification team.

1.2 Proposed Solution

To address the threat of Trojans modifying unspecified design functionality, this thesis provides 1) precise definitions for dangerous unspecified functionality in terms of information leakage, 2) methods to test this potentially dangerous unspecified functionality for

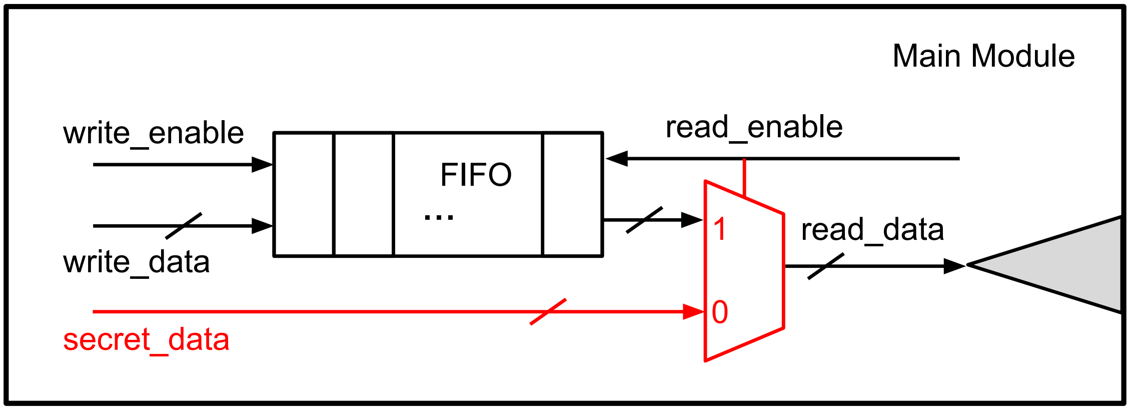


Figure 1.2: Trojan-infested FIFO (Trojan Circuitry affecting unspecified functionality to leak information shown in red)

the existence of Trojans, 3) an abstraction-level agnostic method to identify potentially dangerous unspecified functionality without requiring any prior or specialized knowledge about the design, and 4) numerous examples of how the proposed Trojans can completely undermine system security if inserted in on-chip bus systems, communication controllers, and encryption IP.

By definition, unspecified functionality should never significantly influence design behavior. One way to quantify the influence a signal, x, has in a design is to assign x different values during the conditions where x is unspecified, then check if any primary output signals or registers differ. Normally it does not matter if x propagates to other signals in the circuit when x is unspecified as long as the design functions correctly, but a key insight emphasized throughout this dissertation is that if the value of x can propagate to points the attacker can observe, it is possible for an attacker to insert a Trojan which ties x to a signal originally not accessible to an attacker. This allows an attacker to learn the value of this signal and **leak important information from the circuit without violating design specifications**.

For example, consider the simple first-in first-out (FIFO) buffer shown in Figure 1.2, which writes the data currently at the input write_data to the buffer when the signal write_enable is 1 and the FIFO is not full, and places data from the buffer on the read_data output when the signal read_enable is 1 and the FIFO is not empty. What should the value of read_data be when read_enable is 0 or the FIFO is empty? It may seem logical to assume the value of read_data under such conditions retains its value from the previous valid read, but what if the FIFO has never been written to or read from before? In this case the value is unknowable, and cannot be specified.

It is very likely the verification effort will only examine the value of read_data when read_enable is 1 and the FIFO is not empty because it is assumed that any circuitry in the fan-out of read_data is only used during a valid FIFO read. However, this means an attacker can modify the FIFO design to leak secret information on the read_data output during all cycles for which the unverified conditions hold. This malicious circuitry is shown in red in Figure 1.2. It should be emphasized that these conditions occur quite frequently making this Trojan behavior hard to flag using existing pre-silicon detection methodologies relying on the identification of behavior occurring only under rare conditions.

Chapter 2 focuses on a special class of unspecified functionality, RTL Don't Cares [22]. We present a novel Trojan which leaks information by modifying only existing don't care bits. RTL don't cares have long plagued chip verification due to hard-to-diagnose "X-bugs." In Chapter 2, we provide a formal automated X-analysis technique based on combinational equivalence checking which both prevents the insertion of this new Trojan type and also has the potential to uncover accidental X-bugs as well. We demonstrate our prevention methodology on an Elliptic Curve Processor design susceptible to the insertion of a Trojan capable of leaking all key bits by modifying only don't cares.

Chapter 3 presents a methodology to identify dangerous unspecified functionality beyond RTL Don't Cares [23]. Based on mutation testing, this method is capable of identifying signals and conditions corresponding to dangerous unspecified functionality in FSM, C, SystemC, TLM, RT, and gate-level models. This chapter also provides mutant ranking heuristics to prioritize analysis of the most dangerous functionality. The presented methodology integrates easily with existing verification flows and can be run alongside Coverage Discounting [24], a technique which reflects error propagation abilities of the testbench in functional coverage metrics. This method applied to a UART controller discovered an entire class of Trojan exploiting undefined behavior in bus protocols along with poorly tested interrupt functionality.

Chapter 4 provides general guidelines for identifying dangerous unspecified functionality for an important class of design: **on-chip bus systems** [25]. Regardless of the specific bus topology and protocol, bus behavior is never fully specified, meaning there exist cycles/conditions where some bus signals are irrelevant, and ignored by the verification effort. Chapter 4 presents a general model for creating a covert Trojan communication channel between SoC components by altering existing on-chip bus signals only when they are unspecified and demonstrates how a Trojan channel can be inserted undetected in several widely used standard bus protocols such as AMBA AXI4 and APB. To illustrate how a Trojan channel can give an attacker a powerful foothold in a complex system, a Trojan channel is inserted in an SoC design running a multi-user Linux OS. An on-chip memory (OCM) is available to all users, but access is managed by the kernel to ensure memory isolation and privacy. The channel allows an unprivileged attacker running software on the system to access root-user memory transactions.

Chapter 5 provides an automated procedure to detect Trojan circuitry modifying unspecified design functionality. Building upon Chapters 3 and 4, which focus on identifying possibly dangerous functionality but do not classify the functionality as containing a Trojan or not, the analysis procedure in Chapter 5 takes as input unspecified functionality given as a list of (x, $\mathcal{C}$) pairs. For a design specified as RTL code or a gate-level netlist, a signal x, which is unspecified when a condition, $\mathcal{C}$, holds, our procedure determines if a Trojan is using x to leak information during $\mathcal{C}$. For the FIFO in Figure 1.2, the (x, $\mathcal{C}$) pairs would be (`read_data`, `read_enable` $==0$), and (`write_data`, `write_enable` $==0$). The Trojan circuitry shown in red is detected by our procedure if the signal `secret_data` propagates to the boundaries of the design.

For gate-level designs, the procedure formulates the Trojan detection problem in terms of *combinational equivalence checking* [26], a mature formal verification technique for which commercial tools with the ability to analyze industry-scale designs exist. For designs written in Verilog or VHDL, the procedure traverses the control/data-flow graph to construct formulas for the assignment of all circuit output signals and recent advances in Satisfiability Modulo Theory (SMT) solvers [27] provide the ability to determine if x can affect the formula evaluation under $\mathcal{C}$, thereby indicating the presence of a Trojan. Designs analyzed include an adder coprocessor with an AXI4-Lite bus interface and a UART communication controller with a Wishbone Bus interface.

Chapter 6 details a **post-silicon** Trojan detection methodology developed to activate Trojans using rare conditions to trigger when malicious behavior occurs [28]. The goal of the proposed detection method is to overcome the weaknesses of existing post-silicon Trojan detection strategies which bias test vectors based on controllability and observability metrics. Our method instead uses the observation that to reduce the area/power/timing footprint of the Trojan, an attacker is likely to only use k of n controllable signals for triggering, where $k << n$, and we target exhaustive testing of all k-subspaces.

Chapter 7 summarizes our contributions and gives several possible directions for future research.

Chapter 2

Hardware Trojans Hidden in RTL Don't Cares

2.1 Introduction

In this chapter we present a novel Trojan which leaks information by modifying only existing don't care bits along with a formal automated X-analysis technique based on combinational equivalence checking which both prevents the insertion of this new Trojan type and also has the potential to uncover accidental X-bugs as well. We demonstrate our prevention methodology on an Elliptic Curve Processor design susceptible to the insertion of a Trojan capable of leaking all key bits by modifying only don't cares.

2.1.1 Hardware Trojans

Many Trojan taxonomies have been proposed [12, 29, 13], which categorize Trojans based on the design phase they are inserted, the triggering mechanism, and malicious functionality accomplished (payload). Most existing Trojans can be divided into the following categories:

1. The logic functions of some design signals are altered, causing the circuit to violate the system specification
2. The Trojan leaks information through side-channels, and no functionality of any existing signals is modified

In this chapter, we addresses a third, less studied type of Trojan:

3. The logic functions of **only** those design signals which have **unspecified behavior** are altered to add malicious functionality without violating system specifications

The key difference between Categories 2 and 3 is that Trojans in Category 3 alter the design in the boolean/functional domain, whereas Trojans in Category 2 only manipulate non-boolean side channels, and require characterization of these side channels for detection.

In this chapter, the unspecified behavior necessary to implement Trojans in Category 3 results from **don't cares specified by the designer in the RTL code**. We present techniques to both insert and prevent insertion of Category 3 Trojans in the RTL code or gate-level netlist. An attacker can assign values or tie other internal design signals to RTL don't cares to accomplish malicious functionality, such as leaking secret information. Prevention of this new Trojan type requires **refining the system specification** (Verilog code) by first identifying the "dangerous" don't cares, then disambiguating them by selecting static values to assign.

Unlike Trojans in Category 1, which often rely on rare triggering conditions to avoid causing incorrect design behavior during testing and normal design operation (ex. [14, 15, 16, 17]), our proposed Trojans are theoretically impossible to detect even if all possible input sequences are tested. Moreover, the proposed Trojans are undetectable even if a perfect sequential equivalence checker is used to check if a Trojan-infested RTL or gate-level implementation conforms to a golden RTL design. This is because the design behavior being maliciously modified by the Trojan payload is itself unspecified in the original specification! Therefore, existing pre-silicon detection methodologies targeting identification of nearly unused circuitry, or rare node values [19, 20] do not address this new Trojan type.

IP watermarking by embedding secret information in the assignment of don't care values [30] is conceptually similar to the proposed Trojan insertion methodology, as both view RTL don't cares as an opportunity for the insertion of extra functionality.

To the best of our knowledge, [31] is the only work which recognizes the potential to implement malicious behavior in unspecified design functionality. [31] defines unspecified functionality as incompletely specified state transition and output functions, given a digital system specified as a finite-state machine (FSM). The process of logic synthesis takes an incompletely specified FSM M and transforms M to a completely specified gate-level FSM, M', which may contain additional unwanted state transitions and output assignments while still conforming to the original FSM.

The method proposed in [31] uses state reachability as a metric for trust. First the designer must manually categorize all design states as either protected or non-protected in a golden symbolic FSM model (M). If a path to a protected state exists in the gate-level implementation (M'), but does not exist in M, M' is considered untrusted.

Our work differs significantly from this approach and overcomes the following limitations of the state reachability based method for machines specified using symbolic FSM models proposed in [31]: 1) analysis must be performed on a *symbolic* representation of the design state space, 2) the labeling of protected v. non-protected symbolic states must be done manually and it is likely most designers would not have a clear idea or guidance for this labeling task, and 3) either full reachability analysis of protected states is required, making the method unscalable to modern designs, or the T flip-flops in the circuit can be modified so *no* transitions from unprotected to protected states are allowed.

While it is common for protocols and controller modules to have reference state diagrams or state tables, from which a symbolic representation can be built and analyzed, often the only available specification for a complex design before logic synthesis is described in HDL such as the RTL code. We focus only on analyzing RTL don't cares since these precisely represent the freedom given to the synthesis tool for optimization and the freedom available to the attacker for implementing malicious functionality.

The second major difference between our work and [31] is that our notion of dangerous unspecified functionality is based on information leakage potential instead of protected state reachability. This has the main advantage that the designer is only required to identify attacker-observable signals, avoiding the high-effort manual categorization of symbolic states as protected or non-protected.

2.1.2 RTL X's and Don't Cares

X's appearing in RTL code have different semantics for simulation and synthesis. In simulation, X's represent unknown signal values, whereas in synthesis, X's represent don't cares, meaning the synthesis tool is free to assign the signals either 0 or 1.

During simulation there are two possible sources of X's: 1) X's specified in the RTL code (either explicitly written by the designer or implicit such as a case statement with no default), and 2) X's resulting from uninitialized or un-driven signals, such as flip-flops lacking a known reset value or signals in a clock-gated block. X's from source 1 are don't cares, and are assigned values during synthesis, meaning they are *known* after synthesis, whereas X's from source 2 may be unknown until the operation of the actual silicon.

The Trojans we propose take advantage of source 1 X's, and clearly, if the design logic is fully specified, and don't cares never appear in the Verilog code, these Trojans cannot be inserted. However, don't cares have been used for several decades to minimize logic during synthesis [32], and forbidding their use can lead to unacceptable area/performance/power overhead. For the case study presented in Section 2.4, replacing all X's in the control unit Verilog with 0's results in almost an 8% area increase for the block.

[33] and [34] give an industry perspective and overview of the many problems caused by RTL X's during chip design/verification/debug along with a survey of existing techniques and tools which address X-issues. Simulation discrepancies between RTL and gate-level versions of a design due to X-optimism and X-pessimism, and propagation of unknown values due to improper reset or power management sequences [35] are all issues addressed by existing research and commercial tools.

Our work presents yet another issue resulting from the presence of X's in RTL code, and provides further incentive to allocate verification resources to these existing X-analysis tools. However, existing tools aim to uncover *accidental* functional X-bugs, while the proposed Trojans can be considered a special pathological class of X-bug specifically crafted with malicious intent to avoid detection during functional verification.

X-analysis tools which focus only on providing RTL simulation with accurate X semantics, perform X-propagation analysis only for scenarios occurring during simulation-based verification, or formal methods which only analyze a limited number of cycles (ex. the reset sequence) do not adequately address the proposed threat. Through the examples in the remainder of the paper we aim to highlight the aspects of this new threat that differ most from the existing X-bugs targeted by commercial and academic tools.

The rest of this chapter is organized as follows: Section 2.2 states the threat model we are addressing, and presents two simple examples to illustrate how typical usage of don't cares in Verilog code can potentially lead to undesired information leakage, Section 2.3 presents an automated methodology to analyze all don't care bits in a Verilog design and classify them based on their information leakage potential, Section 2.4 shows the application of this methodology to an Elliptic Curve Processor design with several hundred don't care bits, and Section 2.5 summarizes our contributions.

2.2 Defining Malicious Don't Cares

2.2.1 Threat Model

The Trojans we are proposing are inserted at RTL or gate-level with the aim of avoiding detection by equivalence checking against a Trojan-free RTL model. The Trojans can be inserted by a malicious CAD tool, disgruntled employee, or any person with access to the RTL code and the ability to modify either the RTL or the netlist.

Equivalence checking at the RT level of abstraction becomes *conformance* checking in the presence of RTL don't cares, because a single RTL specification simultaneously represents several possible valid gate-level implementations. When performing equivalence checking between an RTL and gate-level implementation, the gate-level implementation needs to match *only one* of all possible valid gate-level implementations specified by the RTL [26]. The proposed Trojans take advantage of this by transforming the design into a malicious, but valid implementation.

Our prevention methodology assumes the existence of a Trojan-free RTL model to perform X-analysis on, and provides an improved Trojan-free model that can be used detect any Category 3 Trojan through equivalence checking or simulation-based verification methods. Although requiring the existence of a Trojan-free RTL model may seem limiting, one should remember that before an attacker can successfully insert a Trojan by defining RTL don't cares there must exist a Trojan-free version of the RTL code containing the X's that the attacker hopes to exploit.

2.2.2 Illustrative Examples

Example 1: To illustrate how don't cares can be exploited to perform malicious functionality, a contrived example is presented for illustrative purposes. The module given in Listing 2.1 transforms a 4-bit input by either inverting, XORing with a secret key value, or passing the data to the output unmodified. The choice between the 3 transformations is selected using a 2-bit control signal, `control`. When `control=11`, Line 17 specifies that `tmp` can be assigned any value by the synthesis tool to minimize the logic used.

Listing 2.1: simple.v

```
1  module simple (clk,reset,control,data,key,out);
2  input clk, reset;
3  input [1:0] control;
4  input [3:0] data, key;
5  output reg [3:0] out;
6  reg [3:0] tmp;
7  //tmp only assigned a meaningful value
8  //if control signal is 00, 01 or 10
9  always @ (*) begin
10   case(control)
11     2'b00: tmp <= data;
12     2'b01: tmp <= data ^ key;
13     2'b10: tmp <= ~data;
14   //Trojan logic ------
15   //2'b11: tmp <= key;
16   //---------------------
17     default: tmp <= 4'bxxxx;
```

```
18     endcase
19   end
20   always @ (posedge clk) begin
21     if (~reset) out <= 4'b0;
22     else out <= tmp;
23   end
24   endmodule
```

An attacker can take advantage of the implementation freedom given by the RTL by assigning `key` to `tmp`, causing the secret key value to appear at the output of this module. The Trojan can be inserted in the RTL code by uncommenting Line 15, or at gate-level by modifying the netlist after synthesis.

It should be emphasized that in either case, since the assignment of `tmp` during the `control=11` condition is **unspecified**, it is impossible to detect the Trojan even if the design can be exhaustively simulated, or a perfect equivalence checker can compare the golden and Trojan implementations. Cadence Conformal LEC [36] was used to perform 2 experiments: equivalence checking between the golden and Trojan RTL, and equivalence checking between golden RTL and a Trojan-infested netlist. In both cases, the equivalence checker was unable to detect the presence of the Trojan functionality.

The don't cares assigned to `tmp` in Line 17 are *useful* to the attacker because:

1. The don't care assignment is reachable

2. A primary output (which the attacker can observe) differs depending on the value of the don't care bits

Example 2: In the previous example, all the don't care bits are dangerous and should be disambiguated in the RTL code. The following example, similar to Example 1 with the addition of a 3-bit FSM with 5 reachable states, illustrates that not all don't cares are dangerous, and that the goal of any Trojan prevention or X-analysis technique is to identify only the dangerous X's and allow the synthesis tool to use the remaining don't cares for logic minimization.

Listing 2.2: simple_state.v

```
module simple_state(clk,reset,control,data,key,out);
input clk, reset;
input [1:0] control;
input [3:0] data, key;
output reg [3:0] out;
reg [3:0] tmp;
reg [2:0] counter, next_counter;
reg [3:0] pattern;
//Truncated Counter 0-4
//5,6, and 7 never appear
always @(*) begin
  if (counter < 3'h4)
     next_counter <= counter + 3'b1;
  else next_counter <= 3'b0;
end
always @(posedge clk) begin
  if (~reset) counter <= 3'b0;
  else counter <= next_counter;
end
always @(*) begin
  case(counter)
    3'd0: pattern <= 4'b1010;
    3'd1: pattern <= 4'b0101;
```

```
24        3'd2: pattern <= 4'b0011;
25        3'd3: pattern <= 4'b1100;
26        3'd4: pattern <= 4'b1xx1;
27        default: pattern <= 4'bxxxx;
28      endcase
29    end
30    always @ (*) begin
31      case(control)
32        2'b00: tmp <= data;
33        2'b01: tmp <= data ^ key;
34        2'b10: tmp <= ~data;
35        2'b11: tmp <= data ^ {pattern[3], pattern[2:0] & counter};
36      endcase
37    end
38    always @ (posedge clk) begin
39      if (~reset) out <= 4'b0;
40      else out <= tmp;
41    end
42    endmodule
```

In Listing 2.2 there are 6 total assignments of 1-bit don't care values. One could replace these X's in the Verilog code with 6 1-bit signals, $dc_0, dc_1, ..., dc_5$. The attacker can then choose to assign other internal design signals (such as key bits) to the don't care bits or leave them for the synthesis tool to assign. Line 27 can be re-written as:

`default: pattern <=` $\{dc_0,\ dc_1,\ dc_2,\ dc_3\}$`;`

Line 27 is unreachable (and thus `pattern` will never be assigned $dc_0 - dc_3$) because the variable `counter` only takes on values 0-4. These X's are safe, and cannot be used to leak information, therefore are best left in the RTL to aid in logic optimization. A more interesting X-assignment occurs on Line 26, which can be re-written as:

$$\texttt{3'd4:}\ \ \texttt{pattern <= }\{\texttt{1, }dc_4\texttt{, }dc_5\texttt{, 1}\}\texttt{;}$$

The assignment of $\{dc_4\texttt{, }\ dc_5\}$ to `pattern[2:1]` is reachable, however, by manual inspection, one can see that the only assignment influenced by `pattern` (Line 35) contains a bitwise AND between `counter` and `pattern[2:0]`, which prevents dc_5 from propagating further, but not dc_4! This is because when `counter = 3'd4`, Line 35 effectively becomes:

$$\texttt{2'b11:}\ \ \texttt{tmp <= data \^{} }\{\texttt{1, }dc_4\texttt{, 0, 0}\}\texttt{;}$$

In this example only 1 of 6 don't cares is dangerous and necessary to remove. In a design with hundreds of don't cares, it is expected that only a small subset is dangerous, which motivates why it is valuable for an X-analysis tool to take a fine-grain approach and distinguish between unreachable, reachable but non-propagating don't cares, and don't cares that have the potential to propagate to outputs or attacker observable points.

2.2.3 Formal Definition

The following sets of signals can be defined for any design:

S: all signals

D: don't care bits in the RTL code, where $D \subseteq S$

I: signals an attacker can influence, where $I \subseteq S$, and $D \subseteq I$

O: signals an attacker can observe, where $O \subseteq S$

The following sets are defined for each dc_i, $dc_i \in D$:

O_i: observable signals which differ when $dc_i = 0/1$, $O_i \subseteq O$

P_i: set of primary input sequences which cause signals in O_i to differ

Sets O and I can be determined based on the design specification, but it is reasonable to assume that O consists of all primary outputs and I consists of all primary inputs and don't care bits. Through the examples in the previous section, we have seen that a don't care bit, dc_i, is dangerous **iff** $P_i \neq \emptyset$. We will call an input sequence, $T_i \in P_i$, a **distinguishing input sequence** for dc_i. Our solution for identifying dangerous don't cares, given in Section 2.3, determines if $P_i \neq \emptyset$. For scalability reasons, our solution may over-approximate the set of don't cares classified as dangerous.

If dc_i is classified as dangerous, dc_i should be specified in the Verilog code, instead of being left as a don't care bit. If the circuit designer cannot afford to specify all dangerous don't cares due to tight area and timing constraints, the following metric based on $|P_i|$ provides a ranking that can be used to replace the don't cares most accessible to an attacker. $|P_i|$ reflects the information leakage potential of dc_i because if $|P_i|$ is large, then there exist many conditions under which the attacker can learn the value of dc_i. Considering the probability of each sequence in P_i occurring during normal circuit operation, and how many sequences the attacker can force to occur can also improve the accuracy of this metric. An attacker can force a distinguishing input sequence T_i, if all signals in T_i are in I.

2.3 Identification of Dangerous Don't Cares

2.3.1 Methodology

The problem of finding if a distinguishing input sequence exists has been formulated in [37] as a sequential equivalence checking problem. In [37], the analysis was performed to find X-bugs, not prevent Hardware Trojans, but like the Trojans we are proposing, X-bugs result from reachable X-assignments that affect primary outputs in the design.

A key difference between X-bugs and the proposed Trojan type is that in many designs, for example, a serial multiplier, or the Elliptic Curve Processor analyzed in Section 2.4, the values at primary outputs during intermediate cycles in the computation typically don't matter, as long as the final computation result is correct. X's propagating to primary outputs during intermediate cycles generally aren't considered X-bugs if the final result is unaffected, however, information leakage can still occur during these intermediate cycles if the attacker can observe the primary outputs of the circuit.

The equivalence check is performed between 2 near identical versions of the design: one where $dc_i = 0$ and one where $dc_i = 1$. If the designs are identical under all possible input sequences ($P_i = \emptyset$), dc_i cannot possibly be used to leak design information.

We build upon this idea further by addressing the relationship between multiple don't cares in the design, and we formulate the problem in terms of combinational equivalence checking and state reachability analysis.

While combinational equivalence checking between two nearly identical designs is efficient and scalable, state reachability analysis is not. In the Elliptic Curve Processor case study presented in Section 2.4, we illustrate how commercial code-reachability tools can be used in place of symbolic state reachability analysis to re-classify don't cares erroneously marked as dangerous after combinational equivalence checking as safe.

Consider the generic example circuit in Figure 2.1, where the sequential behavior has been removed by making all flip-flop inputs pseudo primary outputs (PPOs) and all flip-flop outputs pseudo primary inputs (PPIs). There are n don't care bits in the design, and it is clear that dc_i and dc_j have the ability to block each other from propagating. dc_h is in the fan-in cone for signal a, and can also influence the propagation of dc_i and dc_j, while dc_k is completely independent from dc_i, dc_j, and dc_h.

Combinational equivalence checking can be performed between 2 versions of the original design: $C_{dc_i=0}$, and $C_{dc_i=1}$, by constructing the miter in Figure 2.2 and checking the

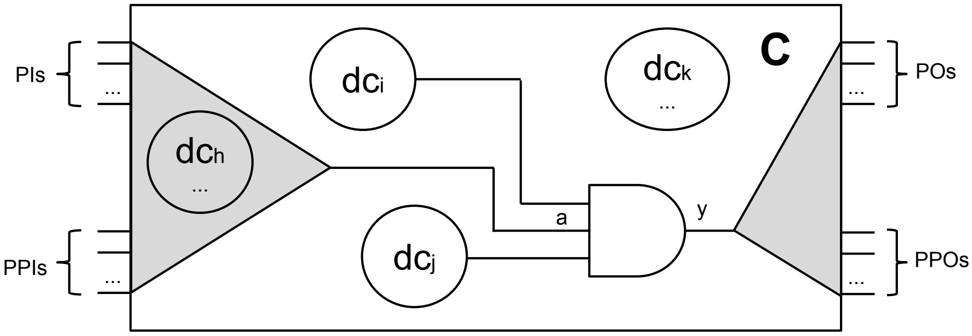


Figure 2.1: Generic Circuit with Don't Care Bits

satisfiability of node z. If z is UNSAT, then dc_i is safe. Otherwise, the equivalence checker returns a distinguishing input vector (since we are performing combinational analysis, the distinguishing sequence is now just a single vector). Note that when analyzing dc_i, all remaining $n-1$ don't care bits are made primary inputs. This ensures the distinguishing input vector contains information about how the remaining don't care bits are constrained if dc_i is to successfully leak information.

Since we are not considering the sequential behavior of the design, the distinguishing input vector could require that the pseudo primary inputs be assigned a value that can never occur, in other words an *unreachable state*. State reachability analysis can be performed before analysis of all don't care bits, and a logic formula, L, describing the set of unreachable states can be incorporated into the miter circuit as shown in Figure 2.3 to prevent the equivalence checker from finding distinguishing input vectors containing these states.

State reachability is a hard problem, but recent advances in model checking [38] and techniques such as [39], which over-approximate the set of reachable states, can aid in addressing non-trivial designs. Additionally, since don't cares can often be traced back to single-line assignments in the Verilog code, dead-code analysis and code reachability tools can help easily eliminate don't care assignments that are unreachable.

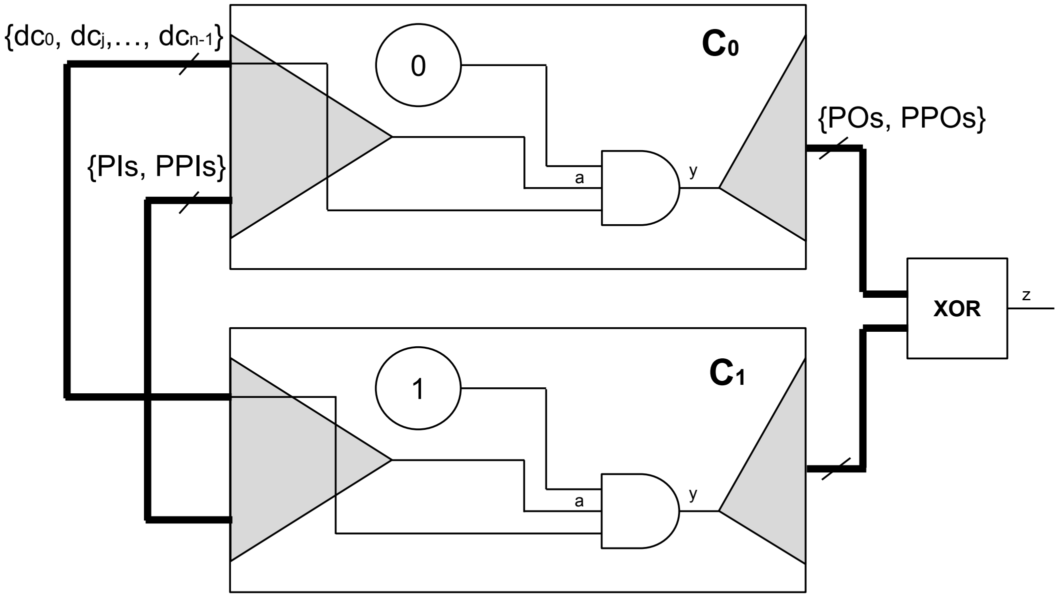


Figure 2.2: Equivalence Checking Formulation

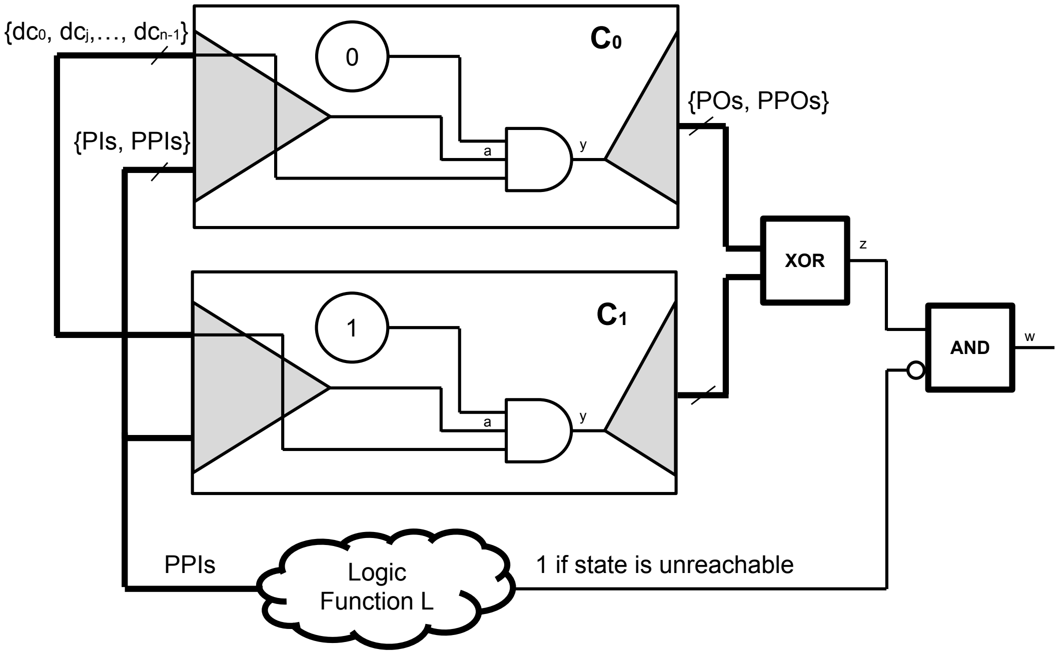


Figure 2.3: Equivalence Checking Formulation Excluding Unreachable States

For Trojan prevention, an over-approximation is ideal because it ensures that a dangerous don't care will never be classified as safe due to the elimination of a distinguishing input vector containing a state erroneously marked as unreachable.

Our analysis uses the robust Verilog parsing capabilities of Yosys [40] to identify don't care bits in the design, and create and write $C_{dc_i=0}$, and $C_{dc_i=1}$ in the Berkeley Logic Interchange Format (BLIF). The logic synthesis tool ABC [41] is then used to perform combinational equivalence checking using the `cec` command. ABC is also used to compute the set of unreachable design states (which is possible for the toy examples in Section 2.2.2) using the `ext_seq_dcs` command.

2.3.2 Existing X-Analysis Tools

Our experiments use ABC and Yosys because of their public availability and transparency, however we are aware that many commercial X-analysis tools and formal engines exist with the capability to perform similar analysis, such as Jasper X-prop [42], Atrenta Spyglass [43], Cadence Incisive [44], and Synopsys Magellan [45], to name only a few.

Our intention is not to argue that the existing tools are incapable of performing the necessary analysis but that settings do not exist in these tools for analyzing don't cares in a security context. We illustrate our approach in the general terms of equivalence checking and state reachability to provide a clear guide to be used by others looking to extract the same information by taking advantage of access to existing commercial tools with advance debug capabilities and optimized runtimes.

2.3.3 Methodology Applied to Examples 1 and 2

Our tool correctly classifies all 4 don't care bits in Example 1 as dangerous. If Example 2 is analyzed without state reachability analysis, our tool classifies all don't cares as dangerous except for dc_5, which is classified as safe due to the bitwise AND on Line 35 which always prevents propagation. All the counterexamples for $dc_0 - dc_3$, assign `counter` to 101, which can never occur.

ABC is used to perform reachability analysis and describe the forbidden states as a logic function, L, which takes as input all pseudo-primary inputs and outputs 1 if the input combination can never occur. We then modify the miter circuit in Figure 2.2 to form the circuit in Figure 2.3 by adding an extra AND gate, with inputs L' and z. If the modified network is satisfiable, then the don't care is dangerous.

Augmenting our methodology to include state reachability information results in the correct categorization of $dc_0 - dc_3$ as safe, leaving only dc_4 classified as dangerous.

2.4 Elliptic Curve Processor

We now present a case study in which manual inspection was used to identify don't cares in the control unit which provide an opportunity for a Trojan to leak all key bits. We then show how our automated prevention method classifies the don't cares which make this exploit possible as dangerous in addition to unearthing several previously unknown opportunities for information leakage.

2.4.1 Background

Elliptic curve cryptography (ECC) is a public key cryptosystem whose fundamental operations use the mathematics of elliptic curves to perform key agreement and generate/verify digital signatures. ECC is currently used in SSH and TLS, and offers more security/key bit than RSA [46].

Like other cryptographic algorithms, ECC operations can be accelerated if implemented in hardware. Our case study examines a publicly available Elliptic Curve Processor (ECP) which performs the point multiplication operation optimized for an FPGA implementation [47].

Point multiplication is the fundamental operation on which all ECC protocols are built, and the reader should refer to [47] for more background on the mathematics behind this operation. Point multiplication takes as input, elliptic curve parameters, an initial point on the elliptic curve, P, and a secret k, and computes $G = [k]P$, which is P "added" to itself k times using the formulas for elliptic curve point addition and doubling. ECC is secure because it is very difficult to discover k knowing only G and P.

2.4.2 The Hardware Trojan

The Trojan inserted into the ECP allows an attacker who only is allowed to observe primary output signals to discover the secret k. This design contains a state machine with 38 states (shown in Figure 2.4), multiple register files, and several custom arithmetic units used by various scheduled operations. The final point, G, is computed when State 38 is reached. Much like the Trojans given in Section 2.2, the ECP Trojan exploits don't cares specified in a `case` statement during the assignment of control signals in the state machine logic.

Listing 2.3 shows a snippet of the control logic. The control logic output signals `cwl` and `cwh` are fed to all functional blocks in the ECP and control routing, register access, and the operation of the custom ALU.

The design has 3 primary outputs: `sx`, `sy`, and `done`. `sx` and `sy` are both 233-bit signals that hold the x and y coordinate of the final curve point G. The `done` signal indicates when `sx` and `sy` are valid. We assume that the attacker can only observe these output signals, and cannot examine the register file, input signals, or any other internal signals. For each state in Figure 2.4, `cwl` and `cwh` are assigned values, however, **there**

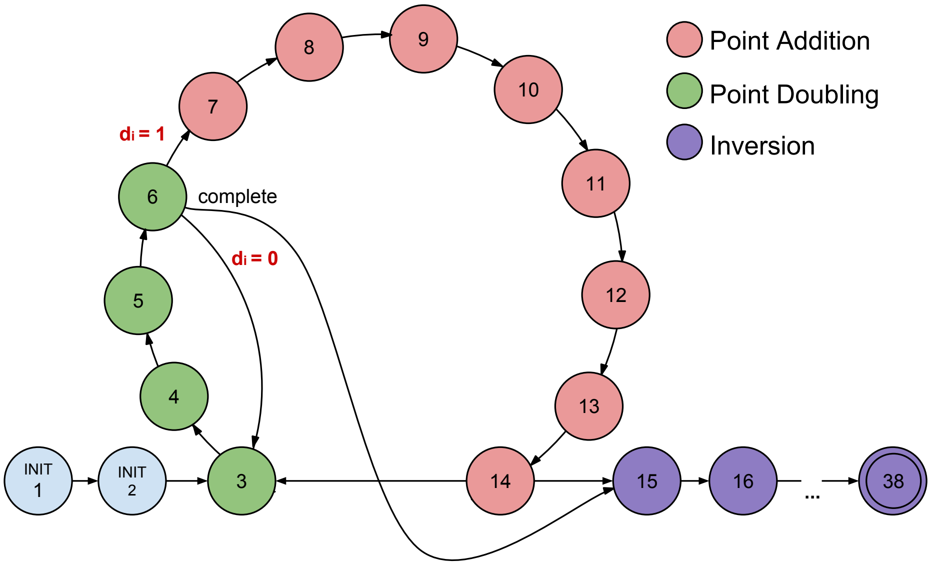


Figure 2.4: ECP State Machine

are many don't cares used to optimize the assignment logic as seen in Listing 2.3. Replacing all the X's with 0's results in an area increase of 8% for the control unit after synthesis.

The assignment of don't cares to bits in `cwl` and `cwh` do not provide opportunity for Trojan insertion in most cases, since the don't cares are specified based on the designer's knowledge of which control bits are relevant during specific states. However, in State 15, control signals for register bank 2 (write-enable and bits in both address ports) are marked as don't care. This can be seen by examining Line 15 in Listing 2.3 and the register file control code in Listing 2.4.

Listing 2.3: Snippet showing X's assigned to control signals

```
1 always @(state) begin
2   case(state)
3     6'd0: begin
4       cwl <= 10'h000;   /* Init L2R Step 1 */
```

```
      cwh <= 23'h4x8484;
    end
    6'd1: begin
      cwl <= 10'h000;
      cwh <= 23'h4x808D;  /* Init L2R Step 2 */
    end
     ...
     6'd15: begin  /* Inv 1 */
       cwl <= 10'hx0D;
       /* NOTICE cwh[7:4] == xxxx  */
       cwh <= 23'h0x04x0;
     end
      ...
  endcase
end
```

These don't cares specify that a gate-level implementation can choose to write or read data to a choice of addresses, effectively making the contents of certain registers in bank 2 unknown during State 15.

Listing 2.4: Snippet from register bank module

```
/* Bank 2 Address Assignments
cwh[6:4] ARE X WHEN STATE==15 */
assign rb2_addr1 = {2'b0, cwh[4:3]};
assign rb2_addr2 = {2'b0, cwh[6:5]};

/* cwh[7] IS X WHEN STATE==15
Leads to primary output Sy being X!  */
```

```
assign rb2_we = cwh[7];
assign rb2_din = (cwh[22] == 1'b1) ? c1
          : ((cwh[15] == 1'b0) ? c0 : c1);
...
```

One of these registers directly influences the primary output signal `sy` in the middle of the point multiplication. The Trojan implemented in Listing 2.5 uses this ambiguity to replace unknown bits in `sy` with key bits.

Normally, an unknown value in a circuit output during an intermediate cycle in the computation is **not considered an error, because it does not affect the final point computed** during the point multiplication. We emphasize that with the knowledge of this new Trojan type, *any* X-propagation to primary outputs during *any* cycle must be prevented.

Listing 2.5: Snippet showing Trojan RTL

```
assign sx = (key != 233'b1) ? a0 :`BASEPOINT_X;
/* TROJAN ----------------------------
When state == 15, the signal a2 is X due to write-enable and an
    address bit being set to X in the control logic */
assign sy = (key != 233'b1) ? ((state == 6'd15) ? {201'bx, key}:
    a2) : `BASEPOINT_Y;
/* ORIGINAL---------------------------- */
/* assign sy = (key != 233'b1) ? a2 : `BASEPOINT_Y; */
```

2.4.3 Automated X-Analysis

The ECP design has 572 primary input bits, 467 primary output bits, and 11232 state elements, resulting in a gate count over 300000. There are 538 don't care bits in the design analyzed by our tool. 282 correspond to assignments made during states $0-38$ to bits in `cwl` and `cwh`, 33 correspond to the `default` assignments ($state > 38$, which should be unreachable) of these signals (see Listing 2.3), and 233 are from a `default` assignment in the `quadblk` module.

Combinational equivalence checking between 2 very similar designs scales well, and each don't care only requires a few minutes of analysis by ABC. Using only combinational equivalence checking, the 538 don't cares are separated into 2 groups: definitely and possibly dangerous (307 bits), and definitely safe (231 bits).

Note that the dangerous don't cares in Row 1 of Table 1 correspond exactly to the don't cares selected by our original manual analysis to implement the Trojan in Listing 2.5! Rows 2 and 3 highlight additional don't cares which an attacker may be able utilize to leak up to 33 bits of information during various states.

The distinction between definitely and possibly dangerous don't cares requires state reachability analysis, because the distinguishing input vector may contain an unreachable state. For example, the variable `nextstate` is assigned don't cares (see Row 4 of Table 2.1) only if the current state variable `state` is outside the $0 - 38$ range, which a quick analysis of the RTL code will reveal can never occur.

Full blown state reachability analysis does not scale well, and we were unable to extract the exact set of unreachable states using ABC. However, we were able to determine that the lines of code containing the X-assignments in Rows 4-6 in Table 2.1 are unreachable using Spyglass, an RTL lint tool from Atrenta [43].

Spyglass performs static analysis on RTL code in order to check that certain "design rules" are not violated. For example, the rule `NoAssignX-ML` is violated if the right-hand side of any assignment contains an X. We first checked the design against the `NoAssignX-ML` rule to identify all the relevant X-assignments, confirming that the don't cares identified by Spyglass were consistent with the don't cares extracted using Yosys.

Table 2.1: Classification of Don't Cares in Elliptic Curve Processor

Row #	**# Don't Care Bits**	**Signal(s) Affected**
Class 1: Definitely Dangerous (35 bits)		
1	2	`cwh[4]`,`cwh[7]`, when `state`==15
2	1	`cwh[12]`, when `state`==2
3	32	`cwl`, for various states ≤ 38
Class 2: Possibly Dangerous (272 bits)		
4	6	`nextstate[5:0]`, when `state` > 38
5	23	`cwh[22:0]`, when `state` > 38
6	10	`cwh[9:0]`, when `state` > 38
7	233	`d[232:0]`, when `cwh[19:16]`==1 or `cwh[19:16]`==15
Class 3: Definitely Safe (231 bits)		

It should be noted that the `NoAssignX-ML` rule does not perform code reachability or X-propagation analysis.

Next the `Av_dontcare01` rule, which identifies *reachable* X-assignments was checked, revealing that the don't cares in Rows 4-6 in Table 2.1 are unreachable, meaning they can be classified as definitely safe.

The don't cares in Row 7 originate from the `quadblk` module and are assigned in a `default` statement. While the assignment condition is possible, the *propagation* of these don't cares is gated by an enable signal, `cwh[20]`. When `state` < 38, the assignment condition and (`cwh[20]`==1) can never be satisfied simultaneously. Since Spyglass only analyzes assignment reachability, these don't cares remain in the possibly dangerous category. A formal property checker could be used to prove that `cwh[20]`==1 && `state` < 38 can never be satisfied if the overhead of removing these don't cares is too costly.

We remove the opportunity for Trojan insertion by replacing the don't care bits listed in Table 2.1 with 0's and use Synopsys Design Compiler (ver I-2013.12-SP2) to synthesize the design and measure the area overhead of the modification. The don't cares in Rows

Table 2.2: Area overhead of Specifying Don't Cares in Elliptic Curve Processor

	% Area Increase	
Don't Cares Replaced w/ Static Values	`ecsmul`	`quadblk`
Class 1	0.04	–
Classes 1 and 2	1.80	3.87
All Don't Care Bits	8.00	3.87

1-6 and Row 7 are from the `ecsmul` and `quadblk` modules respectively.

Table 2.2 shows how replacing only dangerous, both dangerous and possibly dangerous, and all don't cares affects the area overhead of the `ecsmul` and `quadblk` modules. Even though using only combinational equivalence checking over-approximates the number of dangerous don't cares, being cautious and removing all don't cares in Classes 1 and 2 is still preferable to the 8% area increase resulting from indiscriminately replacing every don't care bit (305 total) in `ecsmul`.

2.5 Summary

In this chapter we present a novel Trojan type that utilizes RTL don't cares to leak internal circuit node values without changing circuit functionality [22]. We then formulate the insertion and prevention of such Trojans in terms of don't care analysis, and illustrate, through several examples, how the characteristics of our proposed Trojans compare with already known X-bugs targeted by existing X-analysis tools. We present an X-analysis methodology tailored to aid in the prevention of this new Trojan type, and validate our technique on an Elliptic Curve Processor design with 538 don't care bits. Our technique classified a manageable number of don't care bits as dangerous, leading to a negligible area increase after replacing them with safe values.

Chapter 3

Identifying Dangerous Unspecified Functionality

3.1 Introduction

In this chapter we present a general methodology based on mutation testing to identify unspecified functionality (beyond RTL don't cares) that is susceptible to modification by information leakage Trojans. After applying our method to a UART controller, we discovered an entire class of Trojan exploiting undefined behavior in bus protocols along with poorly tested interrupt functionality, despite the presence of a sophisticated verification infrastructure.

In Chapter 2 we introduced a class of Trojans which leak information by only modifying RTL don't care bits, and used combinational equivalence checking techniques to differentiate don't cares which can be exploited by an attacker to leak information and those which are harmless and should remain in the design for optimization during synthesis. This analysis technique relies on the maturity of combinational equivalence checking tools for RTL designs, making it hard to generalize to SystemC, C, and other high level modeling languages. Additionally, only unspecified functionality captured by don't care bits can be analyzed. This chapter builds upon the ideas presented in Chapter 2, but our proposed mutation-based method is more general since mutation testing is applicable to FSM, C, SystemC, TLM, RT, and gate-level models, only requiring that the model be executable and that a testing scheme exists.

The analysis methodology presented in this chapter randomly samples possible design modifications (known as mutations in mutation testing [48]). We filter out modifications that are not dangerous (do not affect unspecified or poorly tested functionality) by monitoring functional coverage and signals observable to the attacker/user. After our analysis, the verification team is presented with a list of design modifications ordered from most dangerous to least dangerous which are representative of functionality which either needs to be specified or better tested to ensure the absence of Trojans.

The rest of the chapter is organized as follows: Section 3.2 summarizes existing related work on mutation testing, Section 3.3 illustrates how a Trojan only modifying unspecified functionality can leak design information, Section 3.4 reviews Coverage Discounting, a technique our method builds upon, Section 3.5 introduces our methodology for detecting dangerous unspecified functionality, Section 3.6 applies our methodology to a UART design, and Section 3.7 summarizes our results and contributions.

3.2 Related Work

The goal of mutation testing is to gauge the effectiveness of the verification effort by inserting artificial errors (faults) into the design code then recording how many faulty versions of the design (mutants) are detected. Mutation analysis is motivated by the observation that if the test bench is unable to detect artificial errors, it is likely that real design errors are also going unnoticed.

Mutation testing has been used for software security analysis to verify security protocols, determine program susceptibility to buffer overflow attacks, and identify improper error handling [48, 49]. In the hardware domain, mutation testing is primarily used for test bench qualification [50]. Fault models and fault injection tools exist for SystemC [51], TLM [52], and RTL [53].

Two well known drawbacks of mutation analysis are 1) long runtime and 2) large manual effort required to analyze undetected mutants, some of which may be redundant. Redundant mutants are those under all possible inputs, can never cause any change in the design “care” outputs.

Coverage Discounting [24], further detailed in Section 3.4, is a technique which identifies undetected mutants which cause changes in functional coverage. In doing so 1) redundant mutants are filtered out from analysis, 2) the remaining undetected mutants are associated with specific functional coverpoints making analysis easier, and 3) the coverage score is revised to reflect the error propagation and detection properties of the test bench. Our technique builds upon Coverage Discounting by identifying mutants which cause changes in attacker-observable signals (in addition to those which cause changes in functional coverage) to filter out redundant mutants while highlighting mutants related to functionality vulnerable for use in information leakage Trojans.

3.3 Information Leakage Trojans

We return to the simple FIFO example given in Figure 1.2 in Chapter 1, to motivate how mutation testing can identify functionality susceptible to modification by information leakage Trojans. Data is written to the FIFO when it is not full and `write_enable == 1`, and data is read from the FIFO when the FIFO is not empty and `read_enable == 1`. Data can be written to and read from the FIFO simultaneously.

What is the correct value of read_data when read_enable == 0? To save energy, it would make sense to maintain the value of read_data from the last read operation, but it is unlikely that this particular behavior is explicitly specified or tested. The red circuitry in Figure 1.2 shows a simple Trojan which leaks information by routing a secret internal design signal to read_data whenever a read operation is not occurring. Because the value of read_data is able to propagate to the boundary of the main module, the attacker is able to learn the value of secret_data.

Listing 3.1 gives the Verilog code describing the read behavior of the FIFO in Figure 1.2. To illustrate the potential of mutation analysis to highlight the weakness in the verification infrastructure allowing this Trojan to exist undetected, consider a fault which changes the AND operator (highlighted in pink) to an OR operator in Line 2 of Listing 3.1. This fault causes read_data to update whenever the FIFO is not empty, even if a read operation is not occurring. If a read operation occurs, the read pointer will increment as seen in Line 7 of Listing 3.1 and in the following cycle, even if read_enable == 0, read_data will be updated with the value of the next FIFO item.

Listing 3.1: FIFO Read Behavior

```
1 //Memory Access Behavior
2 if (read_enable && !buffer_empty)
3   read_data <= mem[read_ptr];
4 ...
5 //Pointer Update Behavior
6 if (read_enable && !buffer_empty)
7   read_ptr <= read_ptr + 1;
```

During testing, it is likely that a read operation will occur, but the FIFO will not immediately become empty, meaning the spurious updating of read_data can be observed if the waveforms of the fault-free and faulty design are compared. However, it is unlikely that this fault will cause any tests to fail since the fault does not cause the read pointer to spuriously update, and when `read_enable == 0`, the test bench has no incentive to check the value of `read_data`. Notice that the functionality affected by this fault is useful for an attacker because:

1. Observable signals at the boundary of the main module deviate from the fault-free version during testing (indicating that information can be leaked during normal operation without requiring the attacker to force the design into a rare state)
2. The fault is undetected

The methodology presented in Section 3.5 would flag this fault for analysis, forcing the verification team to define behavior for the `read_data` signal when `read_enable == 0` then write a test case or checker for this behavior in order to detect the fault, resulting in an improved test bench able to detect the Trojan in Figure 1.2.

3.4 Coverage Discounting

Before detailing our method for identifying dangerous unspecified functionality, we review Coverage Discounting, the technique our methodology is based upon.

3.4.1 Motivation and Procedure

The goal of verification metrics are to 1) identify if test vectors adequately stimulate the design, and 2) determine if the test bench is capable of detecting errors. Coverage metrics, such as code, toggle, and functional coverage are widely employed and reflect how well the design is activated by test bench, but do not qualify the error propagation abilities of the verification infrastructure. Mutation testing, on the other hand, meets both goals, but is time-consuming and the results are difficult to analyze.

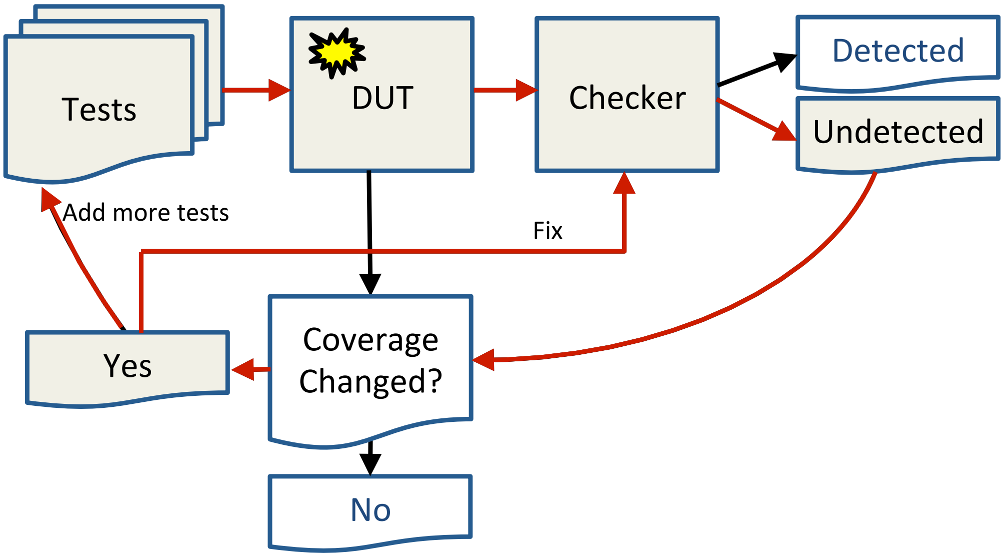


Figure 3.1: Coverage Discounting Flow

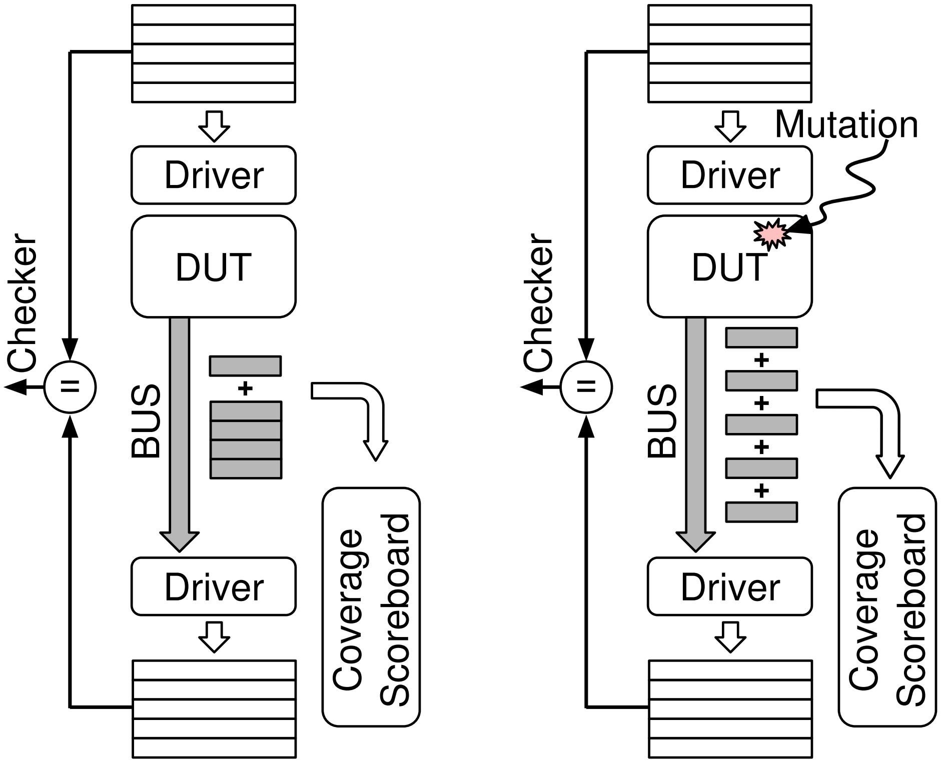
Coverage Discounting incorporates the information mutation testing provides about the error propagation abilities of the test bench into the coverage score meaning after Coverage Discounting, the verification team can target improvement of the coverage score as is typical in the verification flow instead of analyzing artificial faults.

To accomplish this, the decision procedure shown in Figure 3.1 is used to determine if the coverage score needs to be revised downward (the red path shows the conditions necessary for decreasing the coverage score). Before fault injection, coverage is recorded for the fault-free design during all tests. Then for each fault injected in the design:

1. Run all tests on the faulty design and record coverage
2. If all of the tests pass (the fault goes undetected) examine the coverage score: if **coverage changed** under the undetected fault, **subtract the differing coverage** from the original coverage score

To regain the discounted coverage, a verification engineer must write more tests or improve the checkers. Coverage discounting relies on the observation that if a test bench cannot even detect the difference between a design which covers certain functionality and a design which does not, the test bench is incapable of detecting potential errors associated with that functionality.

3.4.2 Example



(a) Original Design

(b) Mutated Design

Figure 3.2: Discounting Example: A Bus Interface Controller (a mutant disabling burst mode is not detected, in turn causing the burst mode coverpoint to be discounted)

Suppose we are validating the bus interface controller shown in Figure 3.2a. Listing 3.2 details the portion of the controller which enables burst transactions and defines a coverpoint, `BURST_MODE`, which is considered covered when `burst == true`.

Listing 3.2: Original Design

```
1 if (transaction_len >= 5)
2   burst:=true; transaction_len:=transaction_len - 4;
3 else
4   burst:=false; transaction_len:=transaction_len - 1;
5  ...
6  BURST_MODE : coverpoint burst;
```

Note that this controller has a bug which is undetected: the number of packets sent in burst mode is assumed to be 5 in the condition (Line 1 in Listing 3.2) and 4 during the transaction length calculation (Line 2 in Listing 3.2). This causes 4 packets, followed by 1 packet to be sent instead of 5 together.

The test bench shown in Figure 3.2a sends a transaction through the controller and captures the output on the far side of the bus. The checker (a collection of assertions, error checks, and other test bench components that together output a binary pass/fail verdict for each test) ensures that the received data matches the sent data. If so, the test will pass, and as long as a transaction with length ≥ 4 is sent by the test bench, the burst mode will be enabled and considered **covered**. However, consider the following mutant, where an error has been inserted into the controller's logic:

Listing 3.3: Mutated Design

```
if ( false )
  burst:=true; transaction_len:=transaction_len - 4;
else
  burst:=false; transaction_len:=transaction_len - 1;
```

The overall effect of this mutation is illustrated in Fig. 3.2b. The output seen on the far side of the bus is identical in both the original and mutated design – it merely arrives later due to being broken up into smaller pieces – so this mutation is undetected by the checker. Since the checker cannot distinguish between normal and burst mode operation, clearly the burst functionality has not been meaningfully covered.

Using only mutation analysis, it is easy for a validation engineer to locate mutated source code and analyze its local behavior, but it can be difficult to use this information to determine a mutation's effect on design functionality and uncover test or checker deficiencies. Coverage discounting, in this example, explicitly links the transaction_len $>= 4 \Rightarrow$ false mutation with the burst mode coverpoint because

1. The burst mode coverpoint is no longer covered in the mutated design
2. The mutation is undetected by the test bench

By providing a more accurate coverage score and explaining the meaning of the mutation in terms of design function, it is more likely that the real bug in this controller will be exposed.

3.5 Identification Methodology

3.5.1 Threat Model

Our method assumes the attacker can modify the design at the RT abstraction level or higher. This is because Trojans inserted in the gate-level netlist or later design stages (meaning there is a golden RTL model available) can be detected using commercial equivalence checking tools which exist as part of the standard chip design flow once the X-analysis method proposed in Chapter 2 is applied to ensure any X's in the golden RTL model cannot be used to implement Trojans.

Similar to Chapter 2, we assume the goal of the Trojans our method targets is to leak information to signals at the boundary of the IP core being analyzed or to internal registers/signals the attacker can observe. Since we identify scenarios where the test bench is unable to detect changes in attacker-observable or IP boundary signals, Trojans performing other functions affecting unspecified behavior besides information leakage are also targeted by our method.

Our methodology can be performed at any level of abstraction, but since identifying unspecified design behavior is central to Trojan prevention, performing analysis at abstraction levels describing the design behaviorally as oppose to structurally makes interpreting the results of mutation testing easier. Moreover, simulation speed improves when more detail is abstracted away from the design, and any improvements in the design specification or functional tests resulting from our technique carry over to all subsequent refined versions of the design relying on the same verification infrastructure.

To clarify, our technique does not directly mark specific lines of code, wires, or gates as being part of a Trojan, in the same way that mutation analysis does not directly find bugs. Rather, our analysis technique highlights design functionality that is susceptible to Trojan insertion and calls for refinement of the specification or more thorough testing of the at risk functionality. If a Trojan already exists in this functionality, then it is likely that the test bench improvements will detect the Trojan, and if there is no Trojan, the improvement effort increases the chances that Trojans inserted in this functionality later on in the design life cycle will be detected.

3.5.2 Mutant Selection

Given the wide variety of mutation models available [48] what is the criteria for selecting the best model for Trojan detection, and can mutation analysis aid in detection of other Trojan types? An underlying assumption of mutation analysis is the Coupling Effect Hypothesis [48]: “Complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults.” The example in Section 3.3 illustrates this concept because the more complex fault (the Trojan in Figure 1.2) can be detected if the simpler fault (highlighted in Listing 3.1) is detected.

If our method targeted Trojans with rare triggering conditions, this hypothesis would not hold, since these Trojans are examples of pathological faults. However, since information leakage Trojans are most effective when they affect a large number of observable signals during a large number of cycles, the uniform structural sampling that simple mutation models provide (such as those used by Certitude [53]) should be effective enough to highlight the most vulnerable unspecified functionality if enough faults are injected. A more thorough analysis of the effect different mutation models have on the success of our technique is a topic for future research.

3.5.3 Mutant Injection and Analysis

In a security context, for a fault to be dangerous, it must be 1) undetected by the test suite and 2) cause changes in attacker-observable signals. Figure 3.3 shows how undetected faults can be classified based on their influence over attacker-observable signals and functional coverage. Dangerous faults fall into Regions A and B.2 in Figure 3.3.

Identifying Attacker-Observable Signals: The labeling of attacker-observable signals depends on the design and attack model. For example, if an attacker can run a malicious user-level software program which interfaces with the hardware Trojan, certain registers will be marked as attacker-observable in addition to network interfaces. If the design being analyzed is a peripheral or co-processor, and it is assumed the main processor may contain a Trojan, the bus interfaces between modules are considered attacker-observable. If the attacker has physical access to the device, then all chip output pads are attacker-observable.

A key point to note is that even if the correct values of some attacker-observable signals are unknown to the verification team, our technique only requires discovering differences in the simulation trace between the faulty and fault-free designs.

What about undetected faults affecting specified functionality (Regions B.1 and B.2 in Figure 3.3)? The faults in Region B.1 do not affect attacker-observable signals but should be examined because they indicate design functionality is not adequately tested! The faults in Region B.1 cause the coverage score to be revised downward during Coverage Discounting [24], which is detailed in Section 3.4. Discounting separates faults affecting design functionality from redundant faults by recording changes in functional coverage caused by each fault. Discounting can be applied to any design where it is possible to define and record functional coverage.

We are able to add our analysis to the existing Coverage Discounting flow with only the additional overhead of tracking attacker-observable signals. The following flow both identifies test bench weakness affecting specified functionality and highlights dangerous unspecified functionality:

1. Record values of attacker-observable signals and functional coverage in the original design during all tests

2. Analyze the design and generate a set of faults, then inject each fault and re-run all tests, recording the same information as in Step 1

3. Only examine **undetected faults** (ones which do not cause any tests/assertions to fail)

The following details the actions that should be taken for every **undetected fault**, based on the region in Figure 3.3 the fault belongs to:

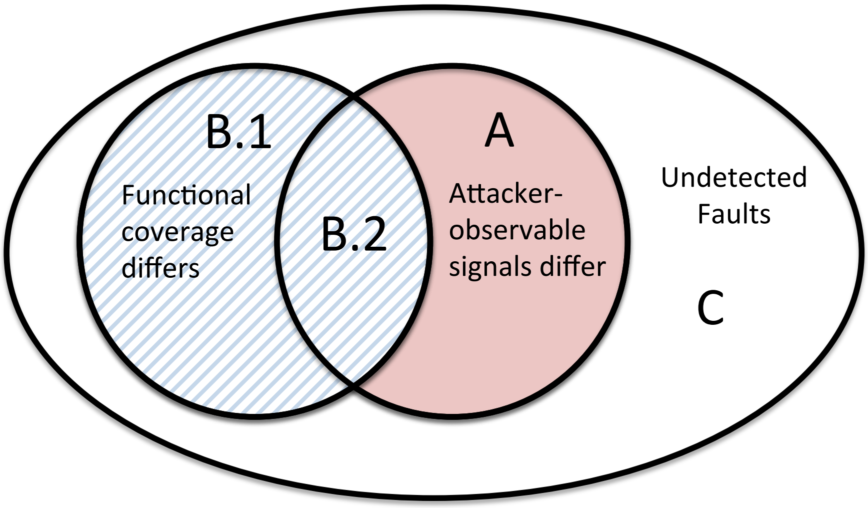

B.1
Functional coverage differs
B.2
A
Attacker-observable signals differ
Undetected Faults
C

Figure 3.3: Scenarios for Undetected Faults

- **Region A:** Functional coverage did **not** change under the fault, but a change in some attacker-observable signals occurred. It is likely that the fault affects **unspecified** design functionality susceptible to the insertion of information leakage Trojans. Behavior for the functionality affected by the fault must be specified, and then the test bench must be improved to check this newly defined behavior.

- **Region B (Regions B.1 and B.2):** Functional coverage changes under the fault meaning **specified** design functionality has been modified and this modification has gone unnoticed by the test bench. This indicates a weakness in the test bench, and the verification engineer must examine why this change in functionality went undetected and fix the test bench.

- **Region C:** Neither functional coverage nor attacker-observable signals change meaning the fault is likely redundant (for example changing the loop condition in `for(x=0;x<10;x++)` to `for(x=0;x!=10;x++)`), and is not worth examining.

3.5.4 Ranking Undetected Faults

Although less than the total number of undetected faults, the number of undetected faults in Region A of Figure 3.3 can still be too costly to completely analyze.

It is desirable that the undetected faults highlighting unspecified functionality most advantageous for an attacker to exploit be analyzed first. Since mutation analysis is an iterative process, after improving the test bench so that it is capable of detecting the most dangerous faults, it is likely that many other previously undetected faults will now be detected and not require analysis.

If there are not enough resources to dedicate to ensuring that all undetected faults are eventually detected, the proposed ranking metrics provide confidence that precious man-hours are spent analyzing only the most severe threats.

The following metrics are easily observed during the injection of each fault:

1. Number of attacker-observable bits differing
2. Total time attacker-observable signals differ
3. Number of *distinct* tests producing differences in attacker-observable signals

Metric 1 captures whether the fault affects a few specific signals or broadly impacts the set of attacker-observable signals. For example, if a design has 10 attacker-observable signals, and Fault A causes 8 to differ at some point during testing, while Fault B only causes 3 to differ, a Trojan related to the functionality of Fault A can potentially leak more bits of information during a given cycle.

However, the number of cycles a given signal differs is also proportional to the information leakage potential of a Trojan based on a particular fault. Metric 2 is the sum over all tests and all attacker-observable signals of the total time each signal differs. If the 8 signals differing under Fault A only differ for 2 cycles each, while the 3 signals differing under Fault B differ for 10 cycles during testing, a Trojan formed from Fault B may be more useful to an attacker.

Metric 3 gauges how likely information leakage will occur under normal usage scenarios. Presumably, the verification tests at the very least exercise typical design functionality. If the attacker cannot force the design into states activating the mutated functionality, faults that lead to observable differences across many tests are more useful for developing Trojans which provide information leakage capabilities during more design states.

3.5.5 Method Overhead and Coverage

One well known disadvantage of mutation testing is the long runtime required to apply the entire test suite to each faulty version of the design. While our methodology also requires simulation of all tests for each mutant, the additional overhead needed to record attacker-observable signals and coverage, then compute differences with the fault-free design, is negligible in comparison.

Another disadvantage of mutation testing is the amount of manual effort required to analyze each undetected fault. The fault ranking mechanism presented in Section 3.5.4 somewhat alleviates this problem by allowing the verification engineer to review the functionality with the largest risk of hiding an information leakage Trojan to be analyzed first. Once the most dangerous fault is detectable by the test bench, all other dangerous faults can be re-evaluated, and those which are now detected due to test bench improvements no longer need to be analyzed.

The simplest and most effective method of decreasing the runtime of mutation analysis is to simulate fewer faults. A method identifying the minimal fault set required to expose all verification holes for a design unfortunately does not exist. Developing metrics to determine when a sufficient number of mutants have been simulated is one of the hardest problems to address in mutation testing. Future research will investigate how the metrics in Section 3.5.4 and additional simulation data can be used to develop these metrics.

3.6 UART Controller Case Study

We analyze a UART (universal asynchronous receiver/transmitter) design from OpenCores [54] using the methodology presented in Section 3.5. After analysis of just 4 of the most dangerous undetected faults returned by our method, we identify unspecified bus functionality and poorly tested interrupt functionality vulnerable to the insertion of information leakage Trojans. After further defining portions of the bus specification and correcting an error in the interrupt checker, the test infrastructure is able to detect these faults as well as an example Trojan inserted in the UART bus functionality. We now provide the case study details.

The test bench used in this case study is a propriety OVM-based suite provided by an EDA tool vendor consisting of 80 directed tests with contained random stimuli, functional checkers, and 846 functional cover points. This test bench is representative of a typical mature regression suite.

There are 38 attacker-observable bits. 32 bits belong to the `wb_dat_o` signal, which is the data bus the UART places values onto when the bus master (often a processor core) issues read requests. The signals `wb_ack_o`, `int_o`, and `baud_o`, are single bit signals which acknowledge bus transactions, signal interrupts, and define the baud rate respectively. These 35 signals comprise the interface to the processor core, while the remaining 3 attacker-observable signals are the off-chip serial output, request to send, and data terminal ready signals. For this experiment, we make the assumption that the attacker

Table 3.1: Categorization of Undetected Faults

110 Undetected Faults		
Region A	**Region B**	**Region C**
30 faults	2 faults	78 faults

is able to see all 38 signals.

Mutation analysis is performed using the commercial mutation analysis tool Certitude [53]. Certitude faults are simple modifications made to the design source code, for example replacing an AND operator with an OR operator, or tying a module port to a static 0 or 1.

Fault Classification: 1183 faults are injected one by one in the design, and tests are run until the fault is detected. Out of the 1183 faults, 110 are not detected by any of the 80 tests. The classification of these faults into Regions A, B, and C in Figure 3.3 is presented in Table 3.1. Using our methodology, the number of faults requiring manual analysis (those in Regions A and B) is reduced from 110 to 32.

3.6.1 Wishbone Bus Trojan

The 3 ranking metrics presented in Section 3.5.4 are equally weighted to identify the most dangerous faults. The 3 most dangerous faults (1411, 1412, and 1413) all affect the following line, which assigns the output enable control bit to 1 in the Wishbone Bus [55] interface if all 4 conditions are true:

Listing 3.4: Assignment of Output Enable

```
assign oe = ~wb_we_is & wb_stb_i & wb_cyc_i & wbstate==2'b01;
```

Each of the 3 faults changes 1 of the bitwise AND operators (highlighted in pink) to a bitwise OR operator. For example, fault 1411 changes the assignment to:

Listing 3.5: Fault 1411

```
assign oe = ~wb_we_is & wb_stb_i & wb_cyc_i | wbstate==2'b01;
```

effectively setting `oe` whenever `wbstate==2'b01`, even if another condition is false, which in the original fault-free design, would have prevented `oe` from being set.

When `oe` is set, the 8-bit data bus lines (coming from the UART register file) are re-sampled, and placed in the correct byte lane on the 32-bit data output bus (`wb_dat_o`) as seen in the following code:

Listing 3.6: Assignment of Data Output Bus

```
if (oe)
  case (wb_sel_is)
    4'b0001:wb_dat_o <= {24'b0,wb_dat8_o};
    4'b0010:wb_dat_o <= {16'b0,wb_dat8_o,8'b0};
    ...
```

If `oe` is not set, the data bus retains its previous value.

Why Are Faults Modifying `oe` Undetected? To understand why spurious changes on the data output bus are not detected by the test bench, which includes a Wishbone bus protocol checker, we must elaborate on the functionality of the protocol control signals involved in the assignment of `oe` (Listing 3.4): `wb_stb_i` (STB_I in the bus specification document), `wb_we_is` (WE_I), and `wb_cyc_i` (CYC_I).

From the specification, STB_I, set by the bus master, selects a particular slave, and "a SLAVE shall respond to other WISHBONE signals only when this [STB_I] is asserted...", "the cycle input [CYC_I], when asserted, indicates that a valid bus cycle is in progress", and "the write enable input [WE_I] indicates whether the current local bus cycle is a READ or WRITE cycle" [55].

In the original design, `oe` is only set during a valid read transaction. Under the

3 faults, `oe` is incorrectly set during write transactions, when the UART slave is not selected, and when a valid bus cycle isn't in progress. However, during these cycles the bus master never captures data from `wb_dat_o`, so the extra data bus changes never cause incorrect data to be read from or written to the UART registers.

Example Output Enable Bus Trojan: This analysis indicates the UART design may be infested with a Trojan that can leak information with impunity on the data bus as long as all the conditions in Listing 3.4 are not simultaneously met and the test bench would be none the wiser! Specifically, the Trojan can take advantage of the fact that the value of `wb_dat_o` is **unspecified** during a write transaction or invalid cycle.

We implement this bus Trojan by changing the assignment of `oe` to match Listing 3.5. We then choose to leak the value `0xdeadbeef` over the bus only during write transactions and invalid cycles by modifying the code in Listing 3.6 to the following code, shown in Listing 3.7 (the Trojan is Lines 2-3):

Listing 3.7: Output Data Bus Trojan

```
1 if (oe) begin
2   if (wb_we_i | ~wb_stb_i | ~wb_cyc_i)
3     wb_dat_o <= 32'hdeadbeef;
4   else
5   case (wb_sel_is)
6     4'b0001:wb_dat_o <= {24'b0,wb_dat8_o};
7     4'b0010:wb_dat_o <= {16'b0,wb_dat8_o,8'b0};
8     ...
```

Figure 3.4 illustrates the ability of the Trojan to leak 32 bits of data during every write transaction while not interfering with read transactions. For example, at 135ns, the UART responds to a read request with the correct data, not `0xdeadbeef`. Simply placing `0xdeadbeef` on the data bus is good for illustrative purposes, but may not be

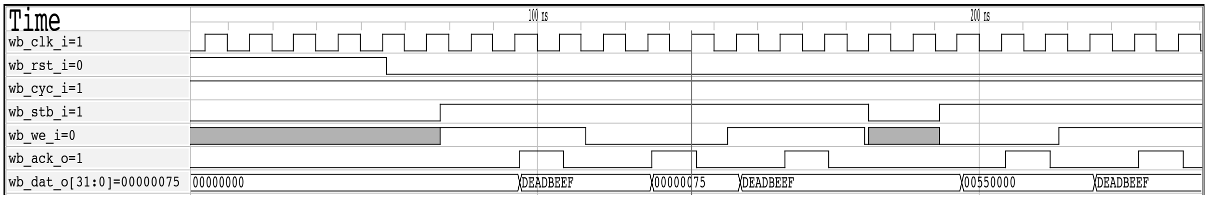


Figure 3.4: Output Enable Trojan Waveform for Bus Protocol Test

useful to an attacker. One should note that Line 3 in Listing 3.7 can be changed to assign *any* 32-bit value to `wb_dat_o`, including other secret internal design signals!

This Trojan is fundamentally different from Trojans relying on rare triggering conditions for stealth as it is active during every write transaction, which is certainly not a rare design state, as evidenced by Figure 3.4. It is very unlikely that this Trojan would be detected by existing methods targeting the identification of rarely used logic.

Improving the Bus Checker: To detect faults 1411, 1412, and 1413, the following additional check is added to the existing bus protocol checker: `wb_dat_o` can not change unless the design has been reset, or a read request is being acknowledged. In addition to detecting the 3 faults, the Output Enable Bus Trojan is also detected.

In a traditional verification setting, it would be unnecessary and cumbersome to add this additional check, and the 3 faults would be considered a waste of time to analyze because they do not affect the correctness of normal read/write operations. Our work is the first to highlight the relationship between undetected faults affecting attacker-observable signals and hardware Trojans, providing motivation to analyze and improve the test bench to detect these seemingly meaningless artificial errors.

Through mutation analysis, which is a random sampling of very specific design modifications, we have actually found a more general class of Trojan, the bus protocol Trojan. The bus protocol Trojan takes advantage of unspecified functionality such as data bus values when no valid transactions are taking place, and the value of the data output bus during a WRITE cycle. The FIFO Trojan in Section 3.3 actually belongs to this class of Trojan.

3.6.2 Interrupt Output Signal

After improving the test bench to detect the 3 faults related to the Wishbone bus, the ranking metrics presented in Section 3.5.4 identify Fault 918, which affects the interrupt mechanism, as the most dangerous fault. This fault affects *specified* functionality, and belongs to Region B.2 in Figure 3.3. Interestingly, this fault is not highlighted by Coverage Discounting, but is by our technique.

The UART uses a single bit signal, `int_o`, to notify the host processor of pending interrupts. There are 5 different events which can cause an interrupt, and the Interrupt Identification Register (IIR) indicates the highest priority interrupt currently pending. A commonly used interrupt is the received data available (RDA) interrupt, which fires when a threshold number of characters is received.

Fault 918 causes `int_o` to become unknown for many cycles during 49 of the 80 tests. More specifically, Fault 918 causes the RDA interrupt pending signal `rda_int_pnd` to become X instead of 1 under certain conditions, making it possible to selectively suppress the RDA interrupt (and consequently `int_o`) without the test bench noticing.

Although the test bench checks if IIR bits are set correctly when conditions for each interrupt type are met, and *most of the time* checks that `int_o` reflects the IIR interrupt pending bit within 10 clock cycles, the behavior of `int_o` is not checked if `int_o` becomes X. Moreover, even if a Trojan set `int_o` to a non-X value in order to leak information, as long as `int_o` becomes both 0 and 1 within 10 clock cycles, the interrupt checkers would not notice that `int_o` is changing spuriously with respect to the IIR interrupt pending bit. This oversight in the test bench is an example of poorly tested *specified* functionality, since the value of `int_o` is clearly being checked in the interrupt checker, but not thoroughly enough.

It is interesting to note that Fault 918 did not cause a change in functional coverage, perhaps suggesting that the coverage model is not detailed enough to highlight meaningful verification holes in the interrupt functionality illustrating the potential of our analysis technique to highlight and qualify the verification of important design functionality outside of the coverage model.

3.7 Summary

In this chapter we propose an automated methodology to identify unspecified functionality vulnerable to modification by information leakage Trojans [23]. Our method is applicable to a wide range of abstraction levels, and also works to identify poorly tested specified functionality in addition to dangerous unspecified functionality. We demonstrate the effectiveness of our approach by finding an entire class of information leakage Trojans related to unspecified bus functionality after analyzing the 3 most dangerous faults highlighted by our method in a UART controller design. Our method also led to the discovery of poorly tested interrupt functionality vulnerable to Trojan insertion. We then close the verification loop by improving the design checkers to detect these faults and show that this improvement leads to the detection of an actual Wishbone bus Trojan.

Chapter 4

Trojan Channels in Partially Specified SoC Bus Functionality

4.1 Introduction

This chapter focuses on Trojans in SoC on-chip buses. In Chapter 3 we presented a general method to identify dangerous unspecified functionality in any in any type of design, including bus systems. Mutant simulation and analysis is expensive, but necessary if one cannot identify dangerous unspecified functionality directly by inspection. Since bus systems are characterized by well-defined protocols and set of common topologies, this chapter directly presents a general model for dangerous unspecified bus functionality. Our work takes inspiration from the Wishbone bus Trojan presented in Chapter 3, and generalizes the Trojan to other bus protocols and more complex bus topologies.

The ability to manipulate the bus system is extremely valuable to an attacker since the bus controls communication between critical system components. A denial of service Trojan halting all bus traffic can render an entire SoC useless. Any information transferred to/from main memory, the keyboard, system display, network controller, etc. can

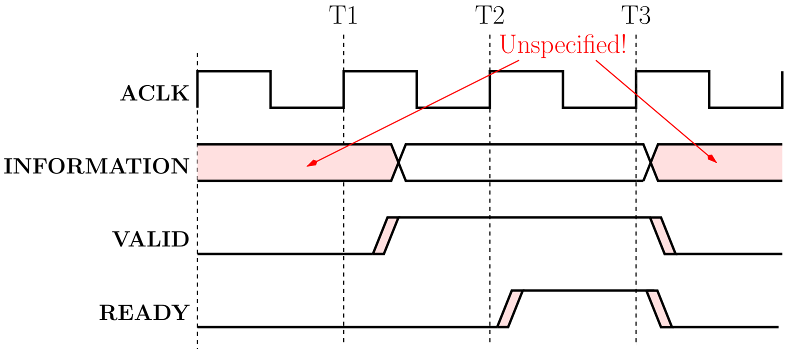


Figure 4.1: AXI Bus Protocol VALID/READY Handshake [56]

be passively captured or actively modified by Trojans inserted in the interconnect.

There exist many different bus protocols designed to optimize different design parameters such as area/timing overhead, power consumption, and performance [57]. Regardless, all protocols employ signals to mark when valid bus transactions occur and handshakes to provide rate-limiting capabilities, meaning valid and idle bus cycles can be clearly differentiated. While bus protocols clearly define the desired values for each data or control signal during *valid* transactions, the values of these signals during idle cycles are **unspecified** and largely ignored by bus protocol checkers, formal verification properties, and scrutiny during simulation-based verification. Trojan behavior during these cycles will not be detected by traditional verification methodologies. For example, Figure 4.1 shows the VALID/READY handshake used by each channel in the widely used AXI4 protocol. When VALID is LOW, the information lines can take on *any* value, including Trojan information.

The Trojans we propose in this work operate entirely within idle bus cycles, with the goal being to provide a covert communication channel built upon existing bus infrastructure. This Trojan channel can be used to connect Trojan components spread across the SoC in addition to enabling information leakage from legitimate components not possible in the original design. Unlike previously proposed bus Trojans, which lock the system bus, modify bus data, and allow unauthorized bus transactions [8, 58], our Trojans never hinder normal bus functionality or affect valid bus transactions.

In Section 4.2 we review the current solutions addressing bus architecture security issues, and motivate why these are not adequate for detecting bus Trojans hiding in partially specified bus functionality. Section 4.3 outlines the threat model and introduces the Trojan channel model and circuitry, and Section 4.4 provides complete details for AMBA AXI4 and APB. The overhead of creating a 2-way information leakage channel between slaves with varying channel parameters in an AXI4-Lite interconnect is explored in Section 4.5, then in Section 4.6 a Trojan channel is inserted in a full SoC system running multi-user Linux to demonstrate how a malicious unprivileged software program can access root-user data. Several detection methodologies are outlined in Section 4.8, and Section 4.9 summarizes the results and contributions of this chapter.

4.2 Related Work

4.2.1 Bus Security

The following are bus security issues being addressed in literature and industry:

1. Malicious snooping of bus data

2. Enforcing bus slave access control policies

3. Deadlock prevention (malicious and accidental)

4. Data integrity, data tampering prevention

Previously proposed bus Trojans include denial of service attacks accomplished by indefinitely asserting the LOCK signal in one of the bus masters or the WAIT signal in a bus slave, observing bus transactions between other components, corrupting bus data, and allowing a master to access forbidden address ranges [8, 58].

In [58], the authors present a secure AHB bus architecture to detect the above mentioned Trojans at runtime. A watchdog timer is added to detect bus deadlock, and to prevent snooping multiplexors are added on all data lines to zero the lines visible to components uninvolved in the current transaction, however this additional circuitry was shown to have significant impact on the maximum bus operation frequency.

Encryption of bus data [14, 59] has been proposed as a method to prevent bus snooping. Key maintenance, along with the overhead of encryption circuitry limits the widespread adoption of this countermeasure. While encryption of bus data prevents snooping, it does not prevent the existence of a Trojan communication channel.

To prevent illegal peripheral access, [58] adds registers holding the allowable access ranges for each bus master ensuring unauthorized requests are blocked and recorded. ARM TrustZone Controllers are commercial IP blocks which provide access control mechanisms to memory regions and bus peripherals and are compatible with other ARM bus IP and AMBA protocols [60].

Both these measures monitor *valid* bus transactions for violations. Since our proposed Trojans never modify existing or create new valid bus transactions, these countermeasures will not detect communication on the Trojan channel. Moreover, neither of these countermeasures address rouge communication between 2 slaves.

Extensive research on formal verification of bus protocols has been performed to ensure deadlock avoidance and fairness [61, 62, 63]. The properties checked using formal methods can be re-used during protocol compliance checking of specific bus implementations using either formal or simulation based methods. The availability of commercial compliance checking verification IP (ex. [64] for AMBA protocols) and pre-packaged SystemVerilog assertions suites [65] illustrate the importance of verifying the correctness of *specified* bus functionality.

During idle bus cycles, when VALID signals are de-asserted, there are no properties/assertions to capture what the correct behavior is, because it is not relevant to the protocol. Our proposed Trojans exploit this fact, and operate exclusively during these cycles to avoid violating assertions or detection during property checking.

4.2.2 Hardware Trojan Detection

Many Trojans proposed in literature hide from the verification effort by only performing malicious functionality under extremely rare triggering conditions. Detection methods targeting this Trojan type identify “almost unused” logic, where rareness is quantified by an occurrence probability threshold. This probability is either computed statically using approximate boolean function analysis [18, 19] or based on simulation traces [20, 21].

The Trojans we propose in this work only modify signals under conditions during which they are *unspecified*, and to be detected by the existing methods, the occurrence of such conditions must be sufficiently rare. We argue that this is seldom the case. For example, our proposed Trojan communication channel can be used to snoop data destined for Slave A by placing data from valid writes to Slave A into a FIFO from which the data is read and leaked to Slave B’s bus interface whenever the channel is idle. The FIFO write condition is a valid data transfer to Slave A, and the leakage condition causing the data to appear at Slave B’s bus interface is an idle channel, neither of which are inherently “rare” conditions.

4.3 Trojan Communication Channel

There are many bus standards, providing the ability to optimize with respect to area/-timing overhead, power consumption, and performance parameters. Common among all standards are control signals marking when valid bus transactions occur. During idle cycles, the value of many control and data signals are unspecified, allowing a powerful Trojan communication channel to be built using the existing bus infrastructure. This section first gives our threat model, then details how to insert such a channel for any bus topology and protocol.

4.3.1 Threat Model

Since a covert communication channel is useless without a sender and receiver of information, we assume that at least one component connected to the system bus contains a Trojan utilizing the information received on the channel, and that there is another Trojan to either leak data from the component it resides in or snoop bus data otherwise not visible to the receiver and send it over the channel.

Although it is possible for the Trojan to create new bus transactions adhering to the bus protocol during unused cycles, verification infrastructure often includes bus checkers which count and log all valid bus transactions. For this reason, our proposed Trojans do not suppress, alter, or create valid bus transactions, but instead re-use existing bus protocol signals to define a new "Trojan" bus protocol allowing communication between different malicious components across the SoC.

Trojan Insertion Stage: It is assumed the Trojans are inserted in the RTL code or higher-level model, meaning no golden RTL model exists to aid in Trojan detection at later stages in the design cycle. A complex SoC requires hundreds of engineers to design and test, and relies on third party IP and tools to meet time to market demands.

A single rouge design engineer or malicious 3rd party IP or CAD tool vendor has the potential to implement a Trojan communication channel.

4.3.2 Trojan Channel Components

The structure and size of the Trojan channel circuitry depends on the following:

1. **Bus Topology:** Determines necessity of FIFO and extra Leakage Conditions Logic at receiver interface

2. **Bus Protocol:** Defines Leakage Conditions Logic and selection of signal(s) to mark valid Trojan transactions

3. **Trojan Channel Connectivity:** Channel can be one-way or bi-directional, contain an active or snooping sender, and involve information leakage between two masters, two slaves, or a master and a slave

4. **Data Width of Trojan Channel (k):** number of bits leaked during a Trojan transaction

5. **FIFO Depth (d):** FIFO used to buffer Trojan channel data if the receiver is busy accepting valid bus transactions

Bus topology and protocol are selected by the system designer, whereas Trojan channel connectivity is chosen by the attacker. Data width (k) and Trojan FIFO depth (d) are parameters selected by the attacker to trade-off performance and overhead of the Trojan channel. The black-colored components in Figure 4.2 are necessary to implement a Trojan communication channel for a shared bus topology, which is shown in Figure 4.3a. For this case, the Data and Control lines from the sender component are directly visible at the receiver. The red-colored components in Figure 4.2 show the extra circuitry

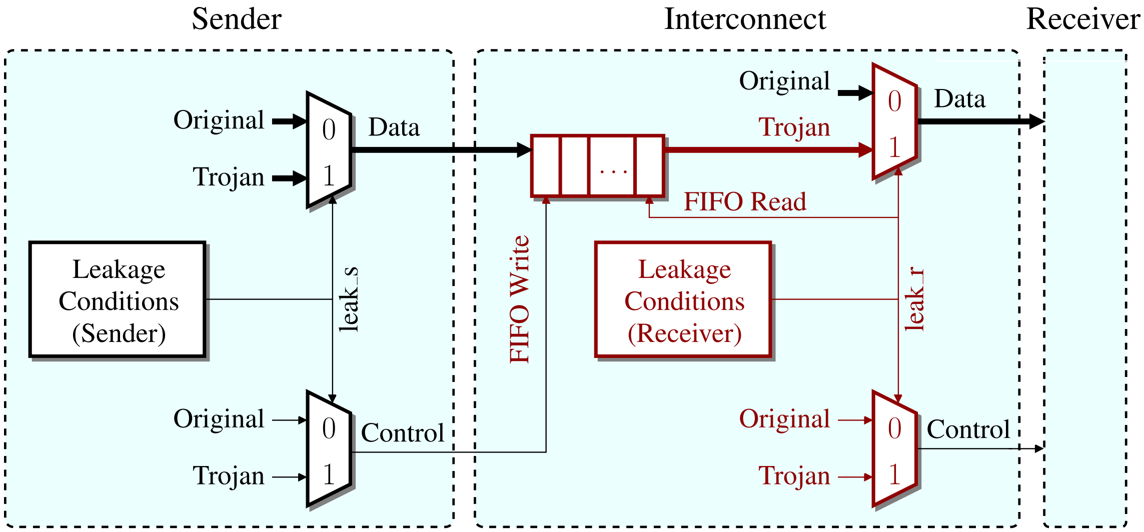
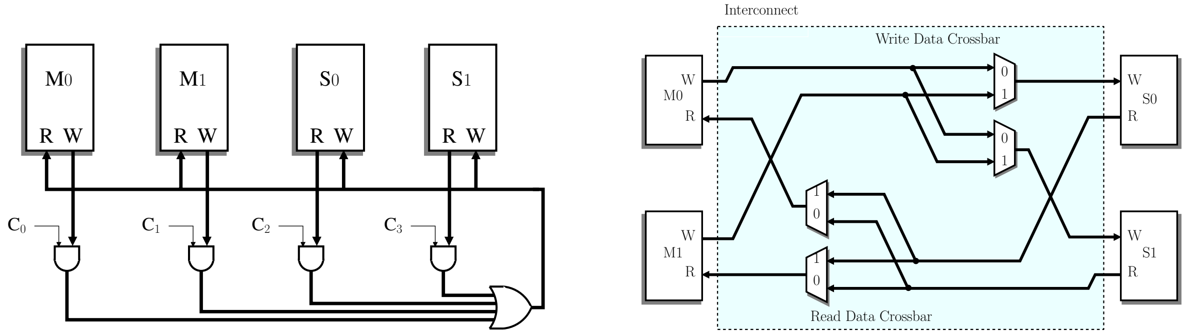


Figure 4.2: Trojan Channel Circuitry

required to implement the channel in an interconnect with a MUX based topology, which is shown in Figure 4.3b.



(a) Shared R/W Data Channels [57]

(b) Concurrent Data Channels [66]

Figure 4.3: Bus Topologies on Opposite Ends of the Area v. Throughput Spectrum

The sender and the receiver can be any master or slave component on the interconnect. The goal of the Trojan channel is to use *only pre-existing* interconnect interfaces to pass data from the sender to the receiver. For example, the line labeled Data in Figure 4.2 on the sender's side could be the write data or read/write address port if the sender is a bus master and the read data port if the sender is a bus slave and vice versa for the Data on the receiver's side.

Since the Trojan data is transmitted using the same lines as normal bus traffic, additional signaling must mark when valid Trojan data is being transmitted. These signals are labeled as Control in Figure 4.2, and like the Trojan data, are mapped to pre-existing data/address/control signals, meaning no additional interface ports are created. The Leakage Conditions Logic is protocol dependent and examines signals at the sender's interconnect interface to determine when it is "safe" to replace the original bus signal values with Trojan values.

4.3.3 Topology Dependent Trojan Channel Properties

All bus signals can be classified as address, data, or control signals, and additionally classified as belonging to read and/or write functionality. The interconnect topology specifies the degree of parallelism between the different categories of bus signals, and the connectivity between masters and slaves [57].

Figures 4.3a and 4.3b show the read and write data channels for topologies sitting at opposite ends of the area efficiency and channel throughput trade-off. Figure 4.3a is the most area efficient, but can only support a single transaction at a time, whereas Figure 4.3b contains significantly more circuitry, but can support multiple simultaneous transactions.

In Figure 4.3a, all read and write transactions are visible to all bus components, meaning no Trojan circuitry is required to simply snoop bus data. If a Trojan bus component wishes to send information, the black-colored circuitry inside the sender block of Figure 4.2 is required. In Figure 4.3b, data is not visible to a component uninvolved in the transaction. Unlike Figure 4.3a, forming a channel between two slaves or two masters requires extra circuitry inside the interconnect, shown in red in Figure 4.2. Because the signals at the sender's interconnect interface are not visible at the receiver's interface and vice versa, new leakage conditions are required, which monitor the receiver's interface and determine when it is safe to leak data without altering valid bus transactions. Signals available at the receiver's interface must also be selected to implement the Data and Control lines. The FIFO is necessary because leakage conditions at the sender and receiver may not occur simultaneously.

4.3.4 Protocol Dependent Trojan Channel Properties

The specifics of the Leakage Conditions Logic, which produces *leak_s* and *leak_r*, and the selection of Data and Control signals depend on the bus protocol used. Because of the similarities between various bus protocols, a general procedure for determining the Leakage Conditions Logic and the selection of Data and Control signals can be given.

Data Signal Selection

In order to remain stealthy, the Trojan cannot create additional signals to transmit data, and must send data via pre-existing signals in the bus protocol. Being that the primary purpose of a bus is to transmit data, all bus protocol/topology combinations have signals that are suitable for sending/receiving Trojan data.

In a protocol with separate read and write data signals, selection depends on if the Trojan Sender/Receiver resides in a master or slave component, since masters drive write data and observe read data signals, and vice versa for slave components. If the Trojan Sender resides in a master component, the read and write address signals can also be used to send Trojan data.

Leakage Conditions Logic

Since pre-existing bus signals are used to transmit Trojan data, logic ensuring that normal bus operation is not compromised by the Trojan is necessary. The Leakage Conditions Logic examines protocol control signals to identify when Trojan Data signals are not being used to transmit *valid* data, and have unspecified values.

Every bus protocol clearly defines the conditions for which data, address, and error reporting signals are valid. Some protocols, such as AXI4, designate a “valid” signal for each data channel, while others such as APB use the current state within the protocol to identify which signals are valid.

leak_s is set when the Trojan Sender has data to transmit and the Data signals are not involved in a valid transaction. If the Trojan Sender is leaking valid bus transactions instead of actively sending information, then *leak_s* is not needed. *leak_r* is set when there are items in the Trojan FIFO and the Data signals at the receiver interface are not currently involved in a valid transaction.

Control Signal Selection

When a Trojan Data signal is not being used in a valid bus transaction, its value is unspecified. During idle bus cycles, either Trojan data is being transmitted, or the bus is truly idle, and no data (Trojan or valid) is sent. To distinguish between these two cases, existing bus signals are selected to be Trojan Control signals, which mark when Trojan data is on the bus.

The criteria for selecting these signals and their corresponding values is that when *leak_s/leak_r* is asserted, the normal behavior of the signal is predictable, but also unspecified. For most protocols, control signals are good candidates because they often are unused during idle cycles, yet their values remain static when idle for a given implementation.

4.4 Protocol Specific Trojan Channel Definitions

Following the general Trojan channel procedure outlined in Section 4.3.4, we present the Leakage Conditions Logic and selection of Trojan Control and Data signals in detail for two commonly used bus protocols from ARM: AMBA AXI4/AXI4-Lite and AMBA APB in order to insert a Trojan in unspecified functionality.

AXI4 is a protocol designed for connecting high speed components such as processors, memory, and network controllers, and contains complex features to increase channel throughput. In contrast, APB is a simple protocol designed to connect low speed peripherals such as UART, keyboard, and timer modules. In a typical SoC, components on the APB bus are connected to the high speed bus via a bridging component [57].

4.4.1 AMBA AXI4

AXI4 defines 5 independent transaction channels seen at the interface of every master and slave: read address channel, read data channel, write address channel, write data channel, and write response channel [56]. Each channel uses the VALID/READY handshake signal pair shown in Figure 4.1 to indicate when the receiver is ready to process bus data, and to mark when valid data is on the bus. Typically, buses using AXI4 choose MUX-based configurations such as those shown in Figure 4.3b, meaning that the red-colored circuitry in Figure 4.2 is required to create the Trojan channel.

The specific AXI4 signals selected to use as Data or Control signals depends on if the sender and receiver are master or slave components.

Master Sender

Data Signal Selection: If the sender is a bus master, data can be leaked through any bus signals a master drives, mainly those on the read or write address channels, or the write data channel. The values of all master driven signals on these channels have no functional meaning when the channel VALID signal is low, hence:

$$\boxed{\textit{leak\_s} = \textit{troj\_data\_ready} \ \& \ {\sim}\text{VALID}}$$

Control Signal Selection: WSTRB is used in both AXI4 and AXI4-Lite, and quoting the specification, "A master must ensure that the write strobes are HIGH only for byte lanes that contain valid data. **When WVALID is LOW, the write strobes can take any value**..." If the application uses all byte lanes in every transfer, it is likely that all strobe bits would be kept HIGH, even when WVALID is LOW, so a good indicator of a valid Trojan transaction would be to set 1 or more bits LOW when *leak_s* is asserted. If the interconnect services peripherals with data widths of 1, 2, and 4 bytes, asserting exactly 3 out of 4 bits of WSTRB is a better option, since this set of values is unlikely to be assigned to WSTRB during normal operation. The following assignment of WSTRB (where WSTRB_ORIG is the Trojan-free value of WSTRB) would work in both cases:

$$\boxed{\text{WSTRB} = \textit{leak\_s} \ ? \ \text{4'b1011} : \text{WSTRB\_ORIG}}$$

The signal WLAST is used to indicate the last transfer in a write burst transaction. When WVALID is low, WLAST is not used, however almost certainly will be de-asserted, meaning that asserting this signal can also mark a valid Trojan transaction:

$$\boxed{\text{WLAST} = \textit{leak\_s} \ ? \ 1 : \text{WLAST\_ORIG}}$$

Slave Sender

Data can be leaked through any bus signals a slave can drive (those on the read data channel or write response channel). The logic for *leak_s* is identical to the logic presented in the previous section since both channels employ VALID signals. To mark when Trojan data is valid, RLAST can be used in a similar manner as WLAST.

RRESP and BRESP are 2-bit error reporting signals and are typically set to indicate "OKAY, normal access success" (all 0's) when not in use (channel VALID is LOW). Setting either RRESP or BRESP to a non-zero state when *leak_s* is asserted can indicate the presence of Trojan data on the bus, for example:

$$\boxed{\text{RRESP} = \textit{leak\_s} \text{ ? 2'b10 : RRESP\_ORIG}}$$

Trojan Receiver

A Trojan master/slave receives information on the same set of bus signals a Trojan slave/master sends. Because of this symmetry, the selection of Data and Control signals is identical to the previous sections. The only difference is that before leaking data to a receiver, the FIFO must not be empty, meaning:

$$\boxed{\textit{leak\_r} = \textit{fifo\_not\_empty} \text{ \& ~VALID}}$$

4.4.2 AMBA APB

The bridging component is the only bus master in APB. The slave components have their own slave select signal (PSELx), but typically share all read data (PRDATA) and control signals (PREADY and PSLVERR) in an AND-OR configuration like the one shown in Figure 4.3a.

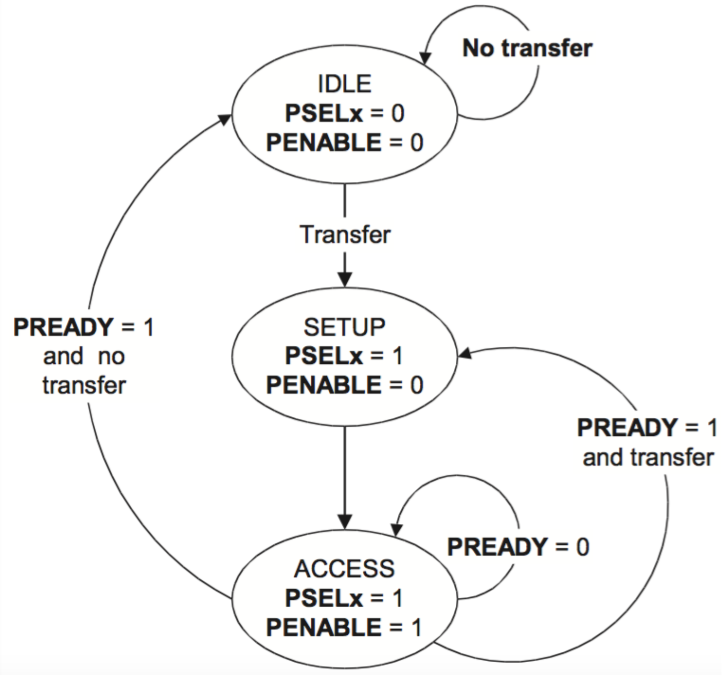


Figure 4.4: AMBA APB Transaction State Diagram [67]

Slave Sender

Since slaves can only drive PRDATA, PREADY, and PSLVERR, PRDATA is used for Trojan Data and PREADY and PSLVERR are selected as the Trojan Control signals. Since all 3 signals are visible to all bus components, the black-colored circuitry presented in Figure 4.2 is sufficient to implement the Trojan channel.

Figure 4.4 shows the state diagram for an APB transaction. PRDATA is only valid during the ACCESS state. The malicious slave leaks information by placing Trojan data on PRDATA as not to conflict with a valid transaction, but can only place data on PRDATA when PSELx is set, meaning information can only be leaked during the SETUP state:

$$\boxed{\mathit{leak\_s} = \mathit{troj\_data\_ready}\ \&\ \text{PSELx}\ \&\ \text{\textasciitilde PENABLE}}$$

Either PREADY or PSLVERR must be used to mark when valid data is on the

Trojan channel. As seen in Figure 4.4, PREADY can take on any value during the SETUP phase without affecting the behavior of a valid transaction. Similarly, quoting the specification, “PSLVERR is only considered valid during the last cycle of an APB transfer, when PSEL, PENABLE, and PREADY are all HIGH” [67]. The combination of setting PSLVERR and de-asserting PREADY during the SETUP phase can be used to signal valid Trojan data.

Master Sender

The APB bridge is the only bus master, and a malicious APB bridge component can be used to connect a Trojan component from the high-speed bus with an APB bus slave. The APB bridge can leak data over PWRITE during the IDLE state, and use the combination of de-asserting all PSEL lines while asserting PENABLE to signal the occurrence of a Trojan transaction.

4.5 AXI4-Lite Interconnect Trojan Example

The system shown in Figure 4.5 is created to verify the AXI4-Lite Interconnect Fabric through RTL simulation. The two slaves are simple 8-bit adder coprocessors which receive 3 operands to add via the interconnect from 3 processors. Since the specifics of the main processors are irrelevant, in the example infrastructure, they are replaced by AXI4-Lite bus functional models (BFMs) from [68]. Additionally, AXI4-Lite assertions packaged by ARM for protocol compliance checking [65] are active during system simulation.

The AXI4-Lite Interconnect Fabric IP block used is the LogiCORE IP AXI Interconnect (v1.02.a) from Xilinx [66] configured in Shared-Address Multiple-Data (SAMD) mode (the topology shown in Figure 4.3b).

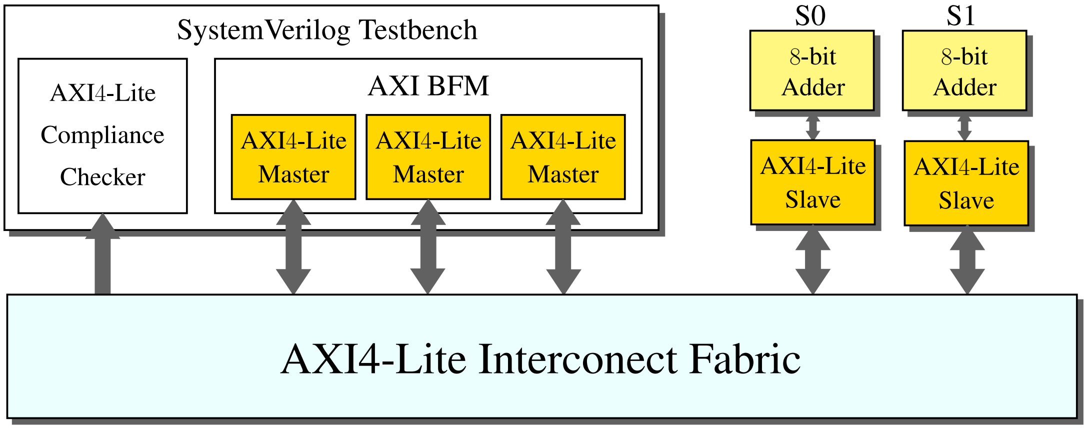


Figure 4.5: AXI4-Lite Example System Verification Infrastructure

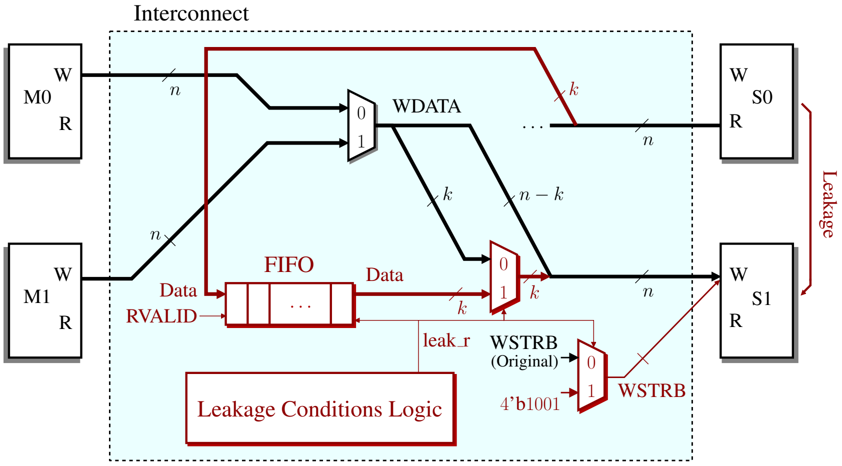


Figure 4.6: Trojan Channel Logic for AXI4-Lite Interconnect

4.5.1 Trojan Operation

The AXI4-Lite Interconnect IP in Figure 4.5 is infected with two copies of the circuitry shown in red in Figure 4.6 to allow S1 to snoop on read requests for S0 and vice versa. Without the Trojan, the read data channel for S0 is not visible to S1 and vice versa.

The waveform in Figure 4.7 first demonstrates how 3 read data responses (values 42, 15, then 14) from S1 are snooped and routed to S0's write channel, then shows a single read data response (value 96) from S0 routed to S1's write channel, and finally

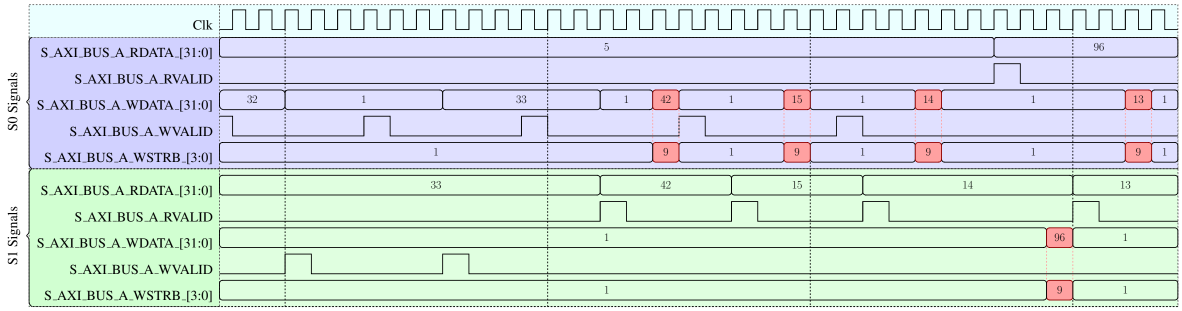


Figure 4.7: 2-way Information Leakage Waveform

Table 4.1: Trojan-Free Design Results (After Place and Route)

Configuration	**# FF**	**# LUT**	**# BRAM**	**Frequency [MHz]**
3 Masters 2 Slaves	1814	2474	2	250
4 Masters 6 Slaves	3071	4247	3	250

another read data response from S1 (value 13) leaked to S0. All Trojan transactions are highlighted in red in Figure 4.7. The WSTRB signal is used to indicate when leaked data is on the bus. Normally WSTRB == 1, but when information is leaked, WSTRB == 9.

For AXI4-Lite, there are over 50 assertions monitoring bus signals during simulation, and none of them are violated even when information is flowing through the Trojan channel!

4.5.2 Overhead

To determine the area and timing overhead of implementing a 2-way Trojan channel between S0 and S1, the SystemVerilog Testbench in Figure 4.5 is replaced by several simple bus masters. Table 4.1 shows results for the Trojan-free design, after placement and route, assuming 3 masters and 2 slaves (labeled as 3M2S) as well as 4 masters and 6 slaves (labeled as 4M6S) for a Virtex-7 FPGA (7vx330t-3).

Table 4.2 illustrates how the selection of Trojan channel parameters Data Width (k)

Table 4.2: Area Overhead of 2-way HW-Trojan Channel

Data Width	FIFO Depth	% Increase in FF		% Increase in LUT	
		3M2S	4M6S	3M2S	4M6S
2	2	0.8	0.5	0.9	0.4
	4	1.1	0.7	1.5	0.6
	8	1.4	0.8	1.8	1.1
4	2	1.0	0.6	1.4	0.7
	4	1.3	0.8	2.0	0.8
	8	1.7	1.0	2.0	1.5
8	2	1.4	0.8	1.8	1.0
	4	1.8	1.0	2.4	1.2
	8	2.1	1.2	3.0	1.7

and FIFO Depth (d) affect the results. The Trojan channel does not affect the operating frequency of the design, and stays within 3% of the original FF and LUT utilization. As the number of masters and slaves increases, the interconnect and overall design area increases, but the size of the Trojan circuitry does not change.

The Trojan channel is easier to hide as the complexity of the interconnect and the number of components connected increases. The master and slave components used to generate the results in Tables 4.1 and 4.2 are far simpler than those in a typical SoC, so the results in Table 4.2 give a loose upper bound on the expected percentage of area increase caused by the Trojan channel in a modern design.

4.6 Trojan Channel in SoC Implementation

To demonstrate how our proposed Trojan channel can give an attacker an extremely powerful foothold in a complex system, we infest a Xilinx Zynq ARM processor based SoC framework running a Linux OS with Trojan circuitry allowing an unprivileged user access to root-user memory transactions. In this section, we detail the Trojan channel operation, the interactions of users within the OS, and the area overhead of the Trojan.

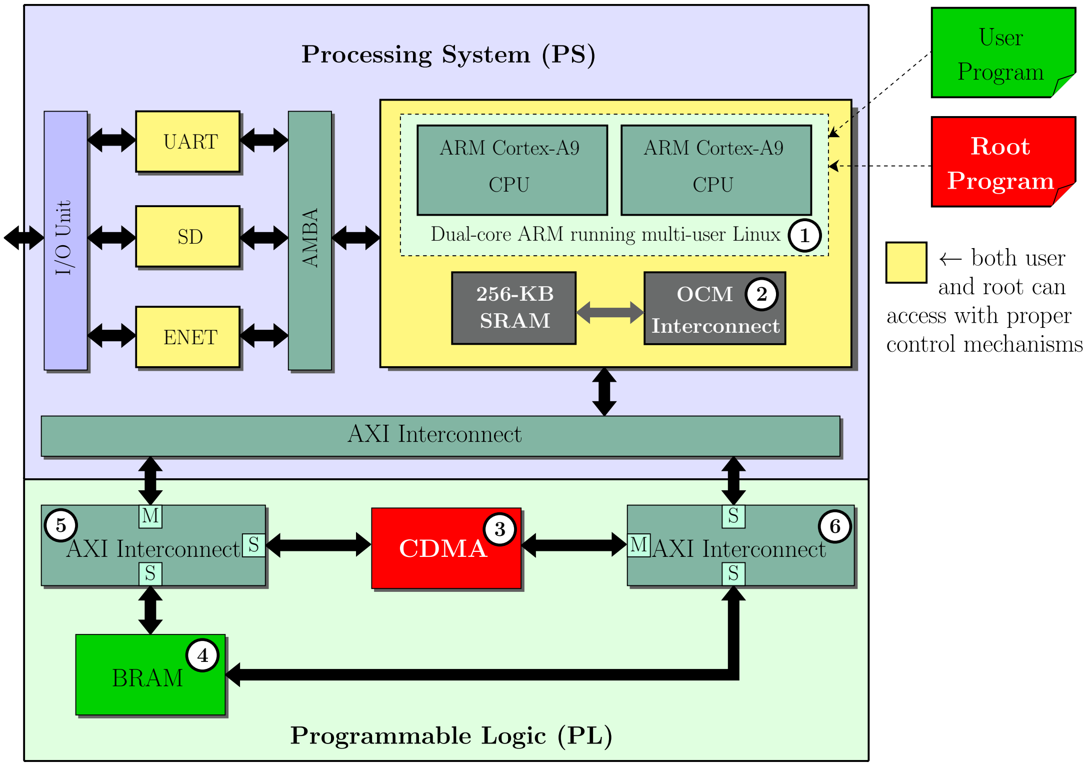


Figure 4.8: Demonstration Platform Block Diagram

4.6.1 Zynq-7000 Based SoC Platform Overview

We design and implement a Trojan infested SoC architecture based on the Zynq-7000 programmable SoC platform in order to demonstrate the operation of the proposed Trojan channel in a real-world application. A full SoC environment running multi-user Linux is created containing Trojan infected Interconnect and Block RAM (BRAM) Controller IP allowing an unprivileged user to observe any data transferred via the Central Direct Memory Access (CDMA) Controller.

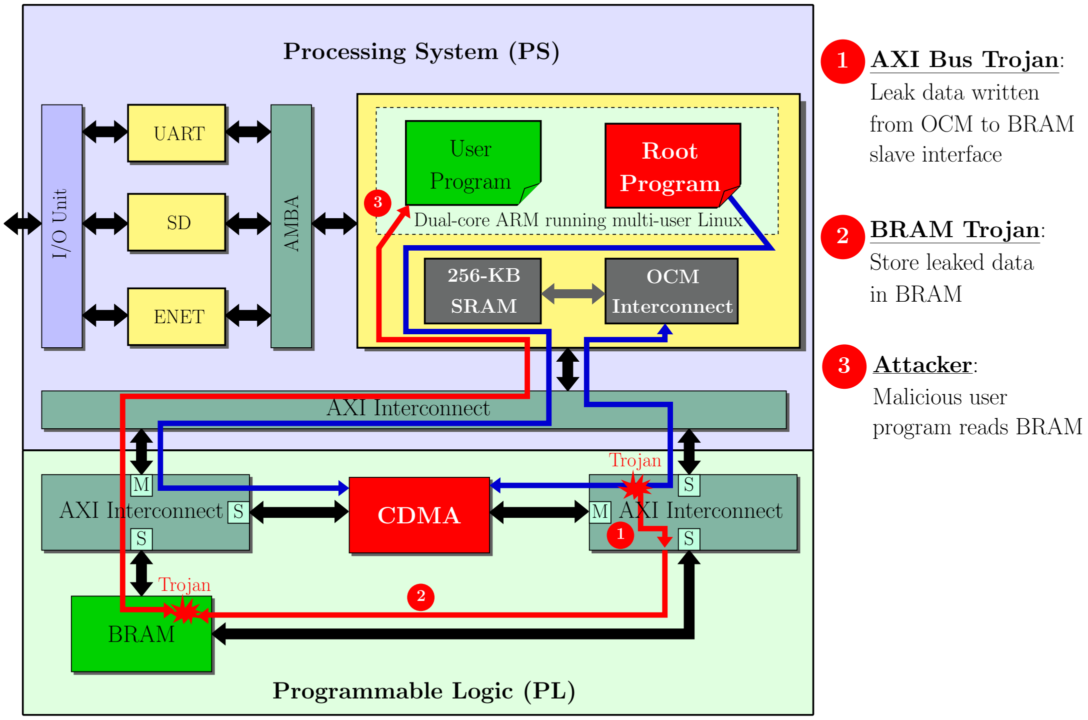
A block diagram of the SoC architecture is shown in Figure 4.8. The SoC architecture includes (1) ARM processors running a multi-user Linux OS, (2) an on-chip memory (OCM) available to all users, but managed by the kernel to ensure memory isolation and privacy, (3) a central direct memory access (CDMA) controller only accessible by a user with root privileges which performs direct memory transfers from a source address to a destination address and (4) a BRAM component which can be accessed directly by any user. Components communicate through several AXI Interconnect blocks, the most relevant labeled as (5) and (6) in Figure 4.8. The ARM cores access the CDMA and BRAM peripherals through (5), and in (6) the CDMA initiates read/write transactions to the BRAM and on-chip memory.

The system is created using Vivado 2015.1 [69] targeting the Zynq-7000 All-Programmable SoC found in the Zedboard platform [70]. The Zynq-7000 architecture integrates two ARM Cortex-A9 cores, on-chip memory, and other peripherals, designated as the Processing System (PS) with Xilinx Programmable Logic (PL) [71]. The Processing System provides the necessary resources to run Xillinux [72], a multi-user Linux distribution, while the flexibility of the Programmable Logic allows for Trojan insertion.

4.6.2 Hardware Trojan Operation

Figure 4.9 illustrates how Trojan circuitry inserted in the BRAM Controller and AXI Interconnect enables an unprivileged user program to observe memory transfers made by root. First, a root program must initiate a DMA transfer by writing to control registers in the CDMA. The most basic DMA transfer requires specifying the Source Address (SA), Destination Address (DA), and number of Bytes to Transfer (BTT) [73]. Once the BTT register is written, the DMA transfer is performed by issuing read and write transactions to the relevant peripheral (in Figure 4.9 the CDMA is transferring data between two locations in on-chip memory). This flow is illustrated by blue arrows in Figure 4.9. The following steps, shown using red arrows in Figure 4.9, illustrate Trojan operation:

1. AXI Bus Trojan leaks transactions visible only at the OCM slave interface to the BRAM slave interface



1 **AXI Bus Trojan**: Leak data written from OCM to BRAM slave interface

2 **BRAM Trojan**: Store leaked data in BRAM

3 **Attacker**: Malicious user program reads BRAM

Figure 4.9: Hardware Trojan Operation

2. BRAM Trojan captures leaked data at the AXI interface, stores at incrementing BRAM memory locations

3. Malicious unprivileged user program reads BRAM locations containing the leaked data

One should note that even if an attacker does not have the ability to run or infiltrate a software program running on the SoC, information from the Trojan channel can be captured and transmitted to the attacker using only hardware Trojans. For example, instead of leaking the DMA transfer data to BRAM, a Trojan infested Ethernet or UART Controller could be used to send data to an attacker.

4.7 Details of Trojan Insertion in Xilinx IP

Each block in the Programmable Logic portion of Figure 4.8 corresponds to a Verilog or VHDL module provided by Xilinx, with Vivado integrating the IP into a complete system. Trojans are inserted in the AXI4 Interconnect and AXI BRAM Controller IP.

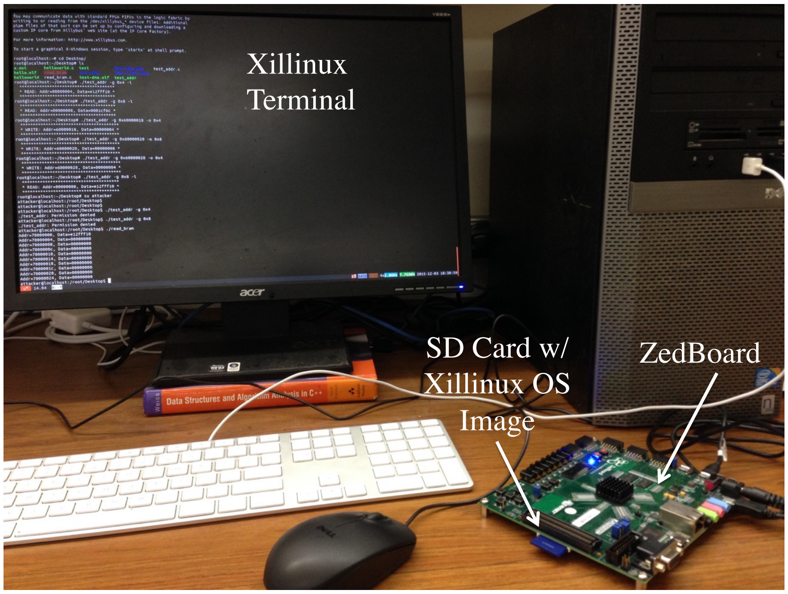
AXI4 Interconnect

The AXI Interconnect block labeled (6) in Figure 4.8 has a single bus master (the CDMA) and two slaves. The Verilog file, *axi_crossbar_v2_1_axi_crossbar.v*, from AXI Interconnect 2.1 (Rev. 5) [74] is modified to insert the Trojan into this block.

Because there is only a single bus master, the 32-bit write data is broadcast to both of the slaves. Even though the BRAM slave can observe write data destined for the processing system, WVALID signals are not broadcast, meaning only the processing system knows which cycle the data is valid. Trojan circuitry is needed to notify the BRAM slave when valid data is being sent to the processing system.

Similar to the example in Section 4.5, the 4-bit WSTRB signal seen at the BRAM slave interface is used to mark when valid data is being written to the processing system. Since the BRAM data width is 32 bits, WSTRB is always 4'b1111. The Trojan circuitry sets WSTRB to 4'b1110 to mark when data is being written to the processing system.

Since there is only one bus master, valid write data can never be sent to BRAM and the processing system simultaneously, guaranteeing that valid write transactions to BRAM are not disrupted when the Trojan alters WSTRB. This eliminates the need for Trojan FIFO or buffering circuitry.



```
You may communicate data with standard FPGA FIFOs in the logic fabric by
writing to or reading from the /dev/xillybus_* device files. Additional
pipe files of that sort can be set up by configuring and downloading a
custom IP core from Xillybus' web site (at the IP Core Factory).

For more information: http://www.xillybus.com.

To start a graphical X-Windows session, type "startx" at shell prompt.

root@localhost:~# cd Desktop/
root@localhost:~/Desktop# ls
a.out        helloworld.c  test          [illegible]   test_addr.c
hello.elf    read_bram     test-dma      [illegible]
helloworld   read_bram.c   test-dma.elf  test_addr
root@localhost:~/Desktop# ./test_addr -g 0x4 -i
 ****************************************
 * READ: Addr=00000004, Data=e12fff10 *
 ****************************************
root@localhost:~/Desktop# ./test_addr -g 0x8 -i
 ****************************************
 * READ: Addr=00000008, Data=0001cf6c *
 ****************************************
root@localhost:~/Desktop# ./test_addr -g 0x60000018 -o 0x4
 ****************************************
 * WRITE: Addr=60000018, Data=00000004 *
 ****************************************
root@localhost:~/Desktop# ./test_addr -g 0x60000020 -o 0x8
 ****************************************
 * WRITE: Addr=60000020, Data=00000008 *
 ****************************************
root@localhost:~/Desktop# ./test_addr -g 0x60000028 -o 0x4
 ****************************************
 * WRITE: Addr=60000028, Data=00000004 *
 ****************************************
root@localhost:~/Desktop# ./test_addr -g 0x8 -i
 ****************************************
 * READ: Addr=00000008, Data=e12fff10 *
 ****************************************
root@localhost:~/Desktop# su attacker
attacker@localhost:/root/Desktop$
attacker@localhost:/root/Desktop$
attacker@localhost:/root/Desktop$ ./test_addr -g 0x4
./test_addr: Permission denied
attacker@localhost:/root/Desktop$ ./test_addr -g 0x8
./test_addr: Permission denied
attacker@localhost:/root/Desktop$ ./read_bram
Addr=70000000, Data=e12fff10
Addr=70000004, Data=00000000
Addr=70000008, Data=00000000
Addr=7000000c, Data=00000000
Addr=70000010, Data=00000000
Addr=70000014, Data=00000000
Addr=70000018, Data=00000000
Addr=7000001c, Data=00000000
Addr=70000020, Data=00000000
Addr=70000024, Data=00000000
attacker@localhost:/root/Desktop$
```

Figure 4.10: Demonstration Environment

AXI BRAM Controller

The Trojan inserted in the AXI BRAM Controller, labeled (4) in Figure 4.8, captures WDATA (32-bits) when WSTRB is 4'b1110, then writes the data to Port B of the BRAM. In our example framework, the address the leaked data is written to starts at 0x70000000, then increases by 4 with every data word written.

The VHDL file, *full_axi.vhd*, from AXI BRAM Controller v4.0 [75] is modified by adding a counter to increment the BRAM address for the leaked data and logic to monitor the AXI write data channel and write the leaked data to the BRAM.

4.7.1 OS-Level Extraction of Trojan Channel Information

Figure 4.10 shows the demonstration environment. Xillinux runs on an SD card located on the Zedboard, and a USB/UART cable connects a desktop workstation to a Xillinux root terminal. The demo uses two Xillinux users: `root` and `attacker`. The privileged user `root` can read/write directly to physical addresses using a program called access_addr while attacker is unprivileged, and cannot use this program.

However, to allow non-privileged users access to the BRAM, the executable read_bram runs with root privileges, but can be executed by any user, and reads the first 10 locations in the BRAM. The read_bram program can be thought of as a very simple device driver since it provides an unprivileged user with controlled and limited access to a peripheral.

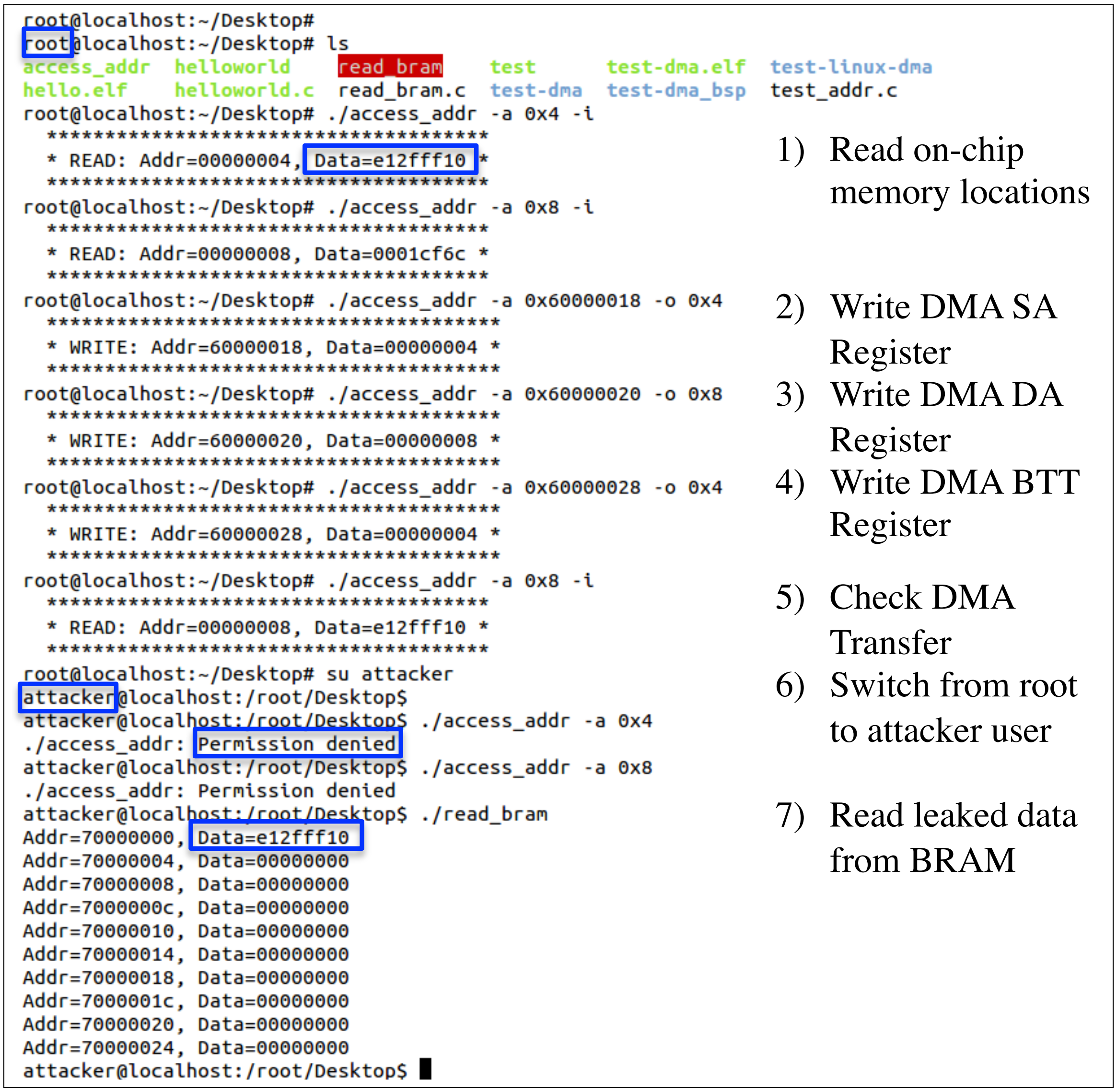
In the demo, root uses the DMA controller to transfer the contents at address 0x4 to address 0x8 (both on-chip memory locations). In the system memory map, on-chip memory addresses start at 0x0, the CDMA base address is 0x60000000, and the BRAM base addresses is 0x70000000.

Figure 4.11 shows Xillinux terminal output during the demonstration of Trojan functionality. Note that the commands at the beginning of the demo are executed as root.

(1) Data at addresses 0x4 and 0x8 are read using access_addr. (2-4) CDMA registers are written, instructing the CDMA to transfer 4 bytes of data from address 0x4 to address 0x8. (5) access_addr is used to confirm that the correct data from address 0x4 (0xe12fff10) is written to address 0x8. (6) The demo switches to the perspective of the attacker user. Notice that attacker tries to execute access_addr to learn the contents of addresses 0x4 and 0x8, but does not have sufficient privileges to do so. (7) Because of the hardware Trojan, the attacker is able to recover the data transferred by root using read_bram.

4.7.2 Overhead

Table 4.3 shows the overhead of inserting the Trojan circuitry in the AXI Interconnect and BRAM Controller IP. The Trojan channel data width is 32 bits, and the interconnect topology is such that no FIFO is necessary. The utilization results given are for the Programmable Logic portion of the platform, since the Processing System exists on the



```
root@localhost:~/Desktop#
root@localhost:~/Desktop# ls
access_addr  helloworld    read_bram    test      test-dma.elf  test-linux-dma
hello.elf    helloworld.c  read_bram.c  test-dma  test-dma_bsp  test_addr.c
root@localhost:~/Desktop# ./access_addr -a 0x4 -i
  **************************************
  * READ: Addr=00000004, Data=e12fff10 *
  **************************************
root@localhost:~/Desktop# ./access_addr -a 0x8 -i
  **************************************
  * READ: Addr=00000008, Data=0001cf6c *
  **************************************
root@localhost:~/Desktop# ./access_addr -a 0x60000018 -o 0x4
  ***************************************
  * WRITE: Addr=60000018, Data=00000004 *
  ***************************************
root@localhost:~/Desktop# ./access_addr -a 0x60000020 -o 0x8
  ***************************************
  * WRITE: Addr=60000020, Data=00000008 *
  ***************************************
root@localhost:~/Desktop# ./access_addr -a 0x60000028 -o 0x4
  ***************************************
  * WRITE: Addr=60000028, Data=00000004 *
  ***************************************
root@localhost:~/Desktop# ./access_addr -a 0x8 -i
  **************************************
  * READ: Addr=00000008, Data=e12fff10 *
  **************************************
root@localhost:~/Desktop# su attacker
attacker@localhost:/root/Desktop$
attacker@localhost:/root/Desktop$ ./access_addr -a 0x4
./access_addr: Permission denied
attacker@localhost:/root/Desktop$ ./access_addr -a 0x8
./access_addr: Permission denied
attacker@localhost:/root/Desktop$ ./read_bram
Addr=70000000, Data=e12fff10
Addr=70000004, Data=00000000
Addr=70000008, Data=00000000
Addr=7000000c, Data=00000000
Addr=70000010, Data=00000000
Addr=70000014, Data=00000000
Addr=70000018, Data=00000000
Addr=7000001c, Data=00000000
Addr=70000020, Data=00000000
Addr=70000024, Data=00000000
attacker@localhost:/root/Desktop$
```

1) Read on-chip memory locations

2) Write DMA SA Register

3) Write DMA DA Register

4) Write DMA BTT Register

5) Check DMA Transfer

6) Switch from root to attacker user

7) Read leaked data from BRAM

Figure 4.11: OS-Level Trojan Demonstration Shell Commands

Table 4.3: Overhead of Programmable Logic in SoC Platform (After Place-and-Route)

	# FF	# LUT	# Memory LUT	# Block RAMs	Freq. [MHz]
Trojan-Free	4766	4149	267	1	50
Trojan-Infested	4809	4201	267	1	50
% Increase	0.9	1.2	0	0	0

FPGA board as hard silicon, and cannot be modified or further optimized by Vivado. The presence of the Trojan circuitry did not affect the frequency of the design, and the FF and LUT utilization rose by approximately 1% making the Trojan circuitry unlikely to be detected due to anomalous area consumption.

4.8 Detection Strategies

To *guarantee* that no Trojan channel exists in the interconnect circuitry, one straightforward procedure is to:

1. Fully specify the behavior of every bus signal
2. Modify the bus implementation to comply with the fully refined specification
3. Formally prove the bus implementation conforms to the behavior specified in 1)

Even if the requirement for formal verification is replaced by assertions monitoring the interconnect during simulation, for complex protocols, the task of complete behavior specification without causing unacceptable overhead is formidable. For example, in AXI4, it is easy to require that if a channel VALID signal is LOW, all other channel signals must be driven LOW. However, given that data and address buses in AXI4 are typically 32 or 64 bits wide, an implementation adhering to this requirement must augment hundreds of bits with MUX circuitry to switch between LOW and the original signal value.

To overcome the large area and power overhead of zeroing circuitry for data and address signals, this circuitry can be implemented only for signals that have the potential to become Trojan channel Control signals (ex. WSTRB and WLAST). Preventing the ability to signal when Trojan transactions occur greatly decreases the usability of the Trojan channel.

If no zeroing circuitry can be afforded, the Trojan channel can be targeted by developing additional complex assertions, which define the behavior of bus signals during invalid cycles in a less straight forward, but more area efficient way. For example, instead of requiring WSTRB == 0 when VALID is LOW, a test bench monitor can record the value of WSTRB during the most recent valid write transaction and require that this value remain unchanged until the next valid transaction.

Developing complex assertions and test bench code to minimize the overhead of defining unspecified behavior still may not be feasible depending on the amount of effort budgeted for design verification, as the development of assertions and code must be done manually. To avoid both the area and timing overhead of defining unspecified functionality and the cost of increasing test bench complexity, we introduce an *automated* Trojan detection methodology in Chapter 5 based on formal methods which is capable of detecting the bus Trojans proposed in this chapter *without* requiring specification refinement. The detection methodology determines if the values of bus signals can propagate to design boundaries when they are unspecified without actually having to define desired values for the bus signals during idle cycles. If any bus component is receiving information from a Trojan channel, this method will flag that component as containing a Trojan.

4.9 Summary

In this chapter we present a new type of Hardware Trojan which creates a covert communication channel between components spread across an SoC using only existing on-chip bus signals without affecting normal bus functionality [25]. We illustrate how our Trojan channel model is applicable to any bus topology and protocol, and give details for two widely used protocols. Our Trojan channel circuitry is shown to avoid detection by a protocol compliance checking suite from the IP vendor, and confirmed to have manageable area overhead. We also illustrate how Trojan channel information can be extracted by malicious unprivileged software by creating a complete SoC platform infected with a bus Trojan. Additionally, several detection strategies are outlined.

Chapter 5

Detecting Hardware Trojans in Unspecified Functionality

5.1 Introduction

In this chapter we present a Trojan detection methodology for Trojans modifying only unspecified design functionality. Prior chapters have proposed different Trojans in this space such as Trojans modifying RTL don't cares to leak information (Chapter 2) and Trojans which create covert communication channels (Chapter 4), however we have still not provided a *detection* methodology, only prevention techniques which must be applied to a Trojan-free design (Chapter 2) and identification of unspecified design functionality which *could potentially* be modified by an attacker (Chapter 3). The method presented in this chapter provides the ability to actually classify unspecified functionality as being Trojan-infested or Trojan-free and can detect the Trojans proposed in prior chapters.

We formulate the detection problem in terms of information leakage by making the observation that if a signal, x, is unspecified under a condition $\mathcal{C}$, the value of x should not be able to propagate to important points in the design such as registers or primary outputs, and if it can, Trojan circuitry is present or a design bug exists. This observation can be concisely expressed as a satisfiability problem, allowing us to take advantage of the recent advances in both boolean and satisfiability modulo theory (SMT) solvers. The main contributions of this chapter are:

1. A general detection methodology for Trojans in unspecified functionality which can be followed using a wide variety of tools and techniques

2. A precise formulation for "dangerous" unspecified functionality expressed as a satisfiability problem

3. A method to detect the Trojan communication channels proposed in [25] which are formed by only modify signals in common on-chip bus protocols when they are unspecified

The rest of the chapter is organized as follows: Section 5.2 explores related work in information flow analysis, Section 5.3 gives the threat model, Section 5.4 precisely formulates the detection problem, Section 5.5 details our detection methodology, Section 5.6 demonstrates the effectiveness and quantifies the overhead of our method using several example designs, and we summarize the chapter in Section 5.7.

5.2 Related Work: Information Flow Analysis

Information flow analysis techniques verify security properties such as confidentiality, integrity, and availability. There exists a large body of work on how to specify and verify these properties for software (ex. [76, 77]), as well as several methods for analysis of information flow in hardware [78, 79, 80] and firmware [81].

Information flow properties specify the conditions under which information can safely flow between signals in a design. A whitelist of such properties describes the proper access and disclosure mechanisms for all important signals, and the design is analyzed to detect violations of the specified properties. Our analysis instead focuses on creating a list of signals and the conditions under which they are functionally meaningless, then making the observation that it is suspicious for information to flow from a signal *anywhere* in the design when it is unspecified. The threat model addressed is different, but our problem can be formulated in terms of information leakage properties.

For example, the Cadence JasperGold Security Path Verification App [82] is a commercial tool which formally proves the existence/lack of a path from a source to destination signal. We did not have access to this tool, but we suspect that our detection formulation can be transformed into a set of properties compatible with the Security Path Verification App. In this work we focus on providing a concise theoretical formulation for Trojan detection, and provide several (but certainly not all possible ways) to solve the problem.

5.3 Threat Model

Our method aims to detect Trojans using design inputs or internal signals when they are unspecified to modify other design signals in a malicious, but covert manner. These Trojans never violate the design specification, and hide completely within unspecified design functionality.

For example, consider a peripheral with registers visible to unprivileged software connected to the same on-chip bus as a memory controller. The Trojans proposed in Chapter 4 allow an attacker to leak information such as memory accesses from the root user to the bus interface of this easily accessible peripheral when signals in the interface are not being used for valid bus transactions. The peripheral could then transfer the leaked information to unused addresses in the register space or unused bit fields in existing registers allowing a malicious, but unprivileged, software program access to sensitive data.

Trojan Insertion Stage: It is assumed that no golden RTL model exists to aid in Trojan detection during later stages in the design cycle, meaning that it is possible for Trojans to be inserted in the RTL code or higher-level model. Hundreds of engineers are involved in the design and test of a silicon chip. A single malicious 3rd party IP provider, CAD tool vendor, or disgruntled engineer has the ability to insert a Trojan in the RTL code.

5.4 Problem Formulation

For a hardware design f, let x be a signal in f which is **unspecified** under condition $\mathcal{C}$. For a given hardware design, there will be many (x, $\mathcal{C}$) pairs. The targeted Trojans will insert malicious functionality by modifying x during $\mathcal{C}$.

Key insight: Instead of enumerating and targeting every possible Trojan in this space, we observe that any functionality the attacker inserts must eventually influence design outputs or modify design state otherwise it is redundant. While malicious functionality is not redundant, by definition unspecified functionality *should be*, and this allows us to formulate the problem without modeling Trojan functionality or defining "expected" behavior for the unspecified functionality. The only assumption made is that when x is unspecified, its value should never influence any key points in the design.

Determining if x influences circuit output under $\mathcal{C}$ can be formulated as a satisfiability problem. If Equation 5.1 is **satisfiable**, two different values of x (x_0 and x_1) result in differences in circuit output under $\mathcal{C}$, which is not consistent with properties of unspecified functionality meaning the existence of Trojan circuitry or a design bug is likely.

$$\mathcal{C} \wedge (f_{x \to x_0} \oplus f_{x \to x_1}) \tag{5.1}$$

A Motivating Example: We return again to the simple FIFO example in Figure 1.2 in Chapter 1. Because the Trojan circuitry never affects FIFO read functionality, it is unlikely to be detected by existing verification methodologies. The value of $read\_data$ when $read\_enable = 0$ is unspecified because it is assumed that any circuitry in the fan-out of $read\_data$ will only propagate or store its value when a valid FIFO read occurs. It would be a waste of precious verification resources to specify and verify the value of $read\_data$ when $read\_enable = 0$, however verifying that the FIFO is Trojan-free only involves defining the $(x, \mathcal{C})$ pairs $(read\_data, read\_enable = 0)$ and $(write\_data, write\_enable = 0)$ and determining the satisfiability of Equation 5.1 for each pair.

5.4.1 Identifying $(x, \mathcal{C})$ Pairs

There are several approaches, based on work presented in Chapters 3 and 4, that can be used to identify $(x, \mathcal{C})$ pairs for a design. An automated approach, applicable to any design type, uses the mutation testing based technique proposed in Chapter 3 in which **undetected** mutants affecting attacker observable signals are flagged as dangerous because an attacker can modify behavior related to the mutant to leak information. A signal differing under a dangerous mutant, and the corresponding condition under which the difference occurs form an $(x, \mathcal{C})$ pair.

Because mutation testing is expensive, another approach is to define $(x, \mathcal{C})$ pairs manually for specific classes of designs. Identifying a complete list of $(x, \mathcal{C})$ pairs for a complex design may not always be possible, however the more pairs included for analysis results in greater confidence in the security of the design. For designs with on-chip bus interfaces, it is straightforward to comprehensively list $(x, \mathcal{C})$ pairs involving bus signals for several well known standard bus protocols. The example designs used to demonstrate the effectiveness of our detection method in Section 5.6 give details for the AXI4-Lite and Wishbone protocols.

5.5 Detection Methodology

5.5.1 Overview

Our detection methodology is given in Figure 5.1. Before any (x, $\mathcal{C}$) pairs are analyzed, an SMT or boolean formula, o, is built for each primary output by traversing the data-flow graph or synthesizing the design.

Then for each (x, $\mathcal{C}$) pair, and primary output o in f:

1. Replace x with new variables x_0 and x_1 respectively in 2 identical copies of o
2. If $SAT(\mathcal{C} \wedge (o_{x\rightarrow x_0} \oplus o_{x\rightarrow x_1}))$, flag signal x as *dangerous* for involvement in Trojan circuitry
3. Inspect design behavior and code activated by the satisfying assignment

If f is an RTL design written in Verilog or VHDL, the analysis can be performed on 1) formulas containing only boolean variables and logical connectives, or 2) formulas containing symbols and operators whose semantic meaning is derived from a theory (ex. the $+$, $-$, $\times$, $<$, $\geq$ operators all have behavior described by the theory of arithmetic and commonly appear in Verilog/VHDL code).

Determining the satisfiability of formulas containing symbols and operators governed by the theory of integers, bit vectors, and arrays requires Satisfiability Modulo Theory (SMT) solvers [27]. In Section 5.5.2 we detail how to construct SMT formulas for each circuit output directly from Verilog code (meaning no synthesis tool is required) and use PySMT [83] to perform Step 2 in the detection procedure.

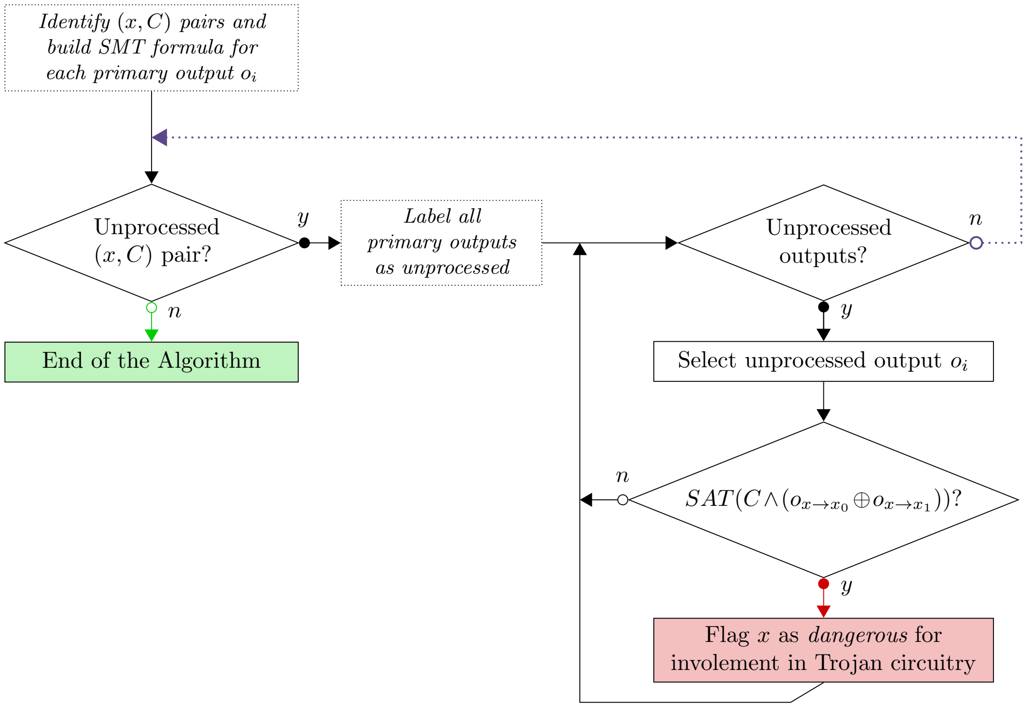

Identify (x, C) *pairs and build SMT formula for each primary output* o_i
Unprocessed (x, C) pair?
y
n
End of the Algorithm
Label all primary outputs as unprocessed
Unprocessed outputs?
n
y
Select unprocessed output o_i
$SAT(C \wedge (o_{x \to x_0} \oplus o_{x \to x_1}))?$
n
y
Flag x as *dangerous* for involement in Trojan circuitry

Figure 5.1: Detection Methodology

The satisfiability of boolean formulas can be determined using a SAT solver, but the approach we present in Section 5.5.3 leverages the robustness and scalability of commercial equivalence checking tools (ex. Cadence Conformal [36]) in which a gate-level model for f produced using logic synthesis is analyzed.

5.5.2 SMT Formulas from RTL Code

Pyverilog [84], an open-source Verilog code parser and static analysis tool written in Python is used to build a data-flow graph for each primary output. To construct the data-flow graph, Pyverilog first builds an abstract syntax tree (AST) representation from the Verilog code and creates a table with information about ports, signals, and constants found in each Verilog module. Then, the AST is traversed again to determine the scope of each signal and resolve all parameters and constants in the design hierarchy, and finally a

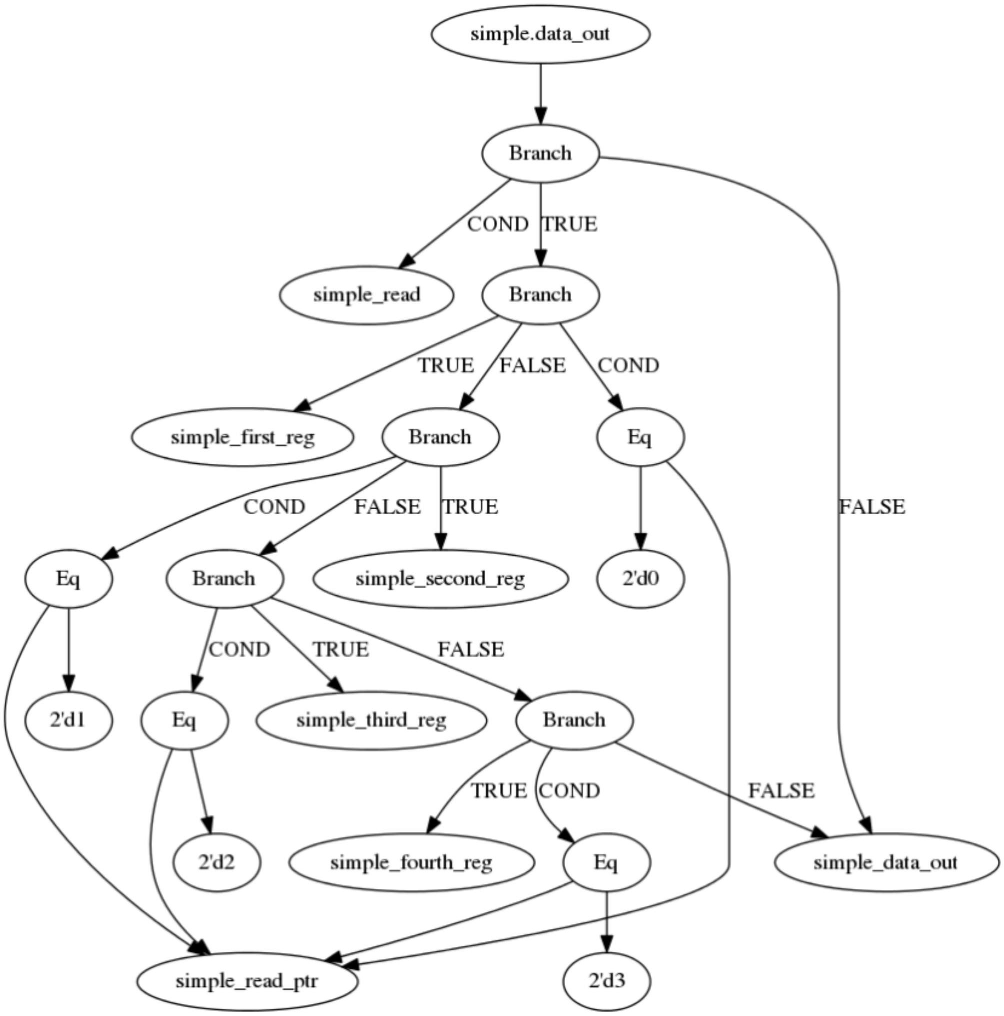


Figure 5.2: Data-flow Graph for *simple.data_out* Generated by Pyverilog

third pass of the AST creates an assignment tree for every signal describing the complete data-flow.

Figure 5.2 shows the data-flow graph generated by Pyverilog for the signal *data_out* (an output in the Verilog module *simple*). The Verilog code for *data_out* is given in Listing 5.1.

Listing 5.1: simple.v

```

module simple(clk, rst, data_in, read, write, data_out, out_valid);

  input clk, rst, read, write;
  input [7:0] data_in;
  output reg [7:0] data_out;
  output reg out_valid;

  wire [1:0] read_ptr, write_ptr;
  reg [7:0] first_reg, second_reg, third_reg, fourth_reg;
  ...
  //data_out logic
  always @(posedge clk) begin
    if (read)
      case(read_ptr)
        2'b00: data_out <= first_reg;
        2'b01: data_out <= second_reg;
        2'b10: data_out <= third_reg;
        2'b11: data_out <= fourth_reg;
      endcase
  end
  ...
endmodule
```

Data-flow graph nodes fall into the following categories: branches, operators (logic, arithmetic, etc.), bit vector slicing and concatenation, constants, and terminals. Our detection methodology builds the formula for each output, o, using Python functions provided by PySMT [83] to create symbols/variables, and describe bit vector, arithmetic, and boolean operations based on the nodes encountered in the data-flow graph.

PySMT provides functions to build formulas involving several theories such as Linear Real Arithmetic (LRA), Real Difference Logic (RDL), Equalities and Uninterpreted Functions (EUF), and Bit-Vectors (BV), then calls existing solvers such as MathSAT [85], Z3 [86], CVC4 [87], Yices 2 [88], CUDD [89], PicoSAT [90], and Boolector [91] to determine if the formula is satisfiable based on the theories present [83].

For each output, all nodes in the data-flow graph generated by Pyverilog are processed by a recursive traversal procedure which returns sub-formulas for each node. Branches, operators, constants, and bit vector operation nodes are straightforward, as PySMT has good support for constructing expressions with boolean and arithmetic bit-vector operations. New variables in the formula are created when terminal nodes are encountered.

Terminal nodes in the graph correspond to intermediate *reg* and *wire* Verilog variables in all modules, inputs and outputs for modules instantiated by the top module, and primary inputs for the top module. If the terminal is a primary input for the top module, a new formula variable is either created or retrieved from a table containing formula variables already encountered during processing. Otherwise, a formula for the *reg*/*wire* variable, input, or output is built by exploring its data-flow graph and then stored in a table to avoid repeated analysis.

The formula built from traversing the graph in Figure 5.2 would contain the $simple.read$ variable because it is a primary input, but the nodes corresponding to $simple.read\_ptr$, $simple.first\_reg$, $simple.second\_reg$, etc. have their own data-flow graphs which are traversed to produce formulas in terms of primary inputs.

One should note that signals corresponding to state elements in the design will reference themselves in the data-flow graph. If this situation is encountered, the variable will become a symbol in the formula to avoid an infinite processing loop. Eventually, the formulas for primary outputs will only reference primary input variables and state variables.

PySMT functions are used to construct Equation 5.2.

$$\mathcal{C} \wedge (o_{x \to x_0} \oplus o_{x \to x_1}) \tag{5.2}$$

The exclusive-or and conjunction operators have corresponding functions in PySMT, and the `substitute` function is used to create $o_{x \to x_0}$ and $o_{x \to x_1}$, where x_0 and x_1 are new independent variables. If Equation 5.2 is satisfiable, PySMT provides a *model*, or one set of possible satisfying assignments for all variables in Equation 5.2 (including the new variables x_0 and x_1) proving that x can influence o under $\mathcal{C}$.

5.5.3 Equivalence Checking

The applicability of the Trojan detection procedure described in Section 5.5.2 to commercial hardware designs is limited by the robustness of the Pyverilog parser and the efficiency and usability of the available open-source SMT solvers. While the available SMT solvers are able to handle incredibly large and complex formulas, reliable transformation of all possible constructs in Verilog/VHDL code to the input format required by

SMT solvers does not exist.

On the other hand, logic equivalence checking for hardware designs is a mature, scalable and robust technology, with several commercial tools available. We present a way to perform the detection procedure outlined in Section 5.5.1 using Cadence Conformal LEC [36], the advantage being employing a tool already part of the circuit design workflow.

One disadvantage of this approach is that if x is a multi-bit signal, the satisfiability of Equation 5.1 must be determined for each bit in x separately. Because equivalence checking compares two gate-level designs, it is impossible to symbolically replace x, a multi-bit signal, with x_0 in one version and x_1 in the other *unless* x is a single bit.

To determine the satisfiability of Equation 5.1 using Conformal, 1) two versions of the design must be created assigning a single bit in x to 0 in one version and 1 in the other, and 2) equivalence should be proven only under $\mathcal{C}$.

To form the two circuit versions, LEC has two commands: `add_primary_input` and `add_pin_constraints`, that force arbitrary signals in the design to 0 or 1 in the golden or revised versions of the circuit. Ignoring equivalence checking results under certain conditions is accomplished in LEC using the `$constraint` function. For example, if a and b are design signals, inserting `$constraint(a==1 && b==1)` in the design source code forces LEC to ignore counterexamples requiring a or b to be 0.

5.6 Case Studies

To validate our detection methodology, we infest two designs, 1) an adder coprocessor, and 2) a Universal Asynchronous Receiver/Transmitter (UART) communication controller, with Trojans modifying unspecified functionality to leak information. The inserted Trojans covertly receive information from the bus interface during idle cycles, and then modify unspecified design functionality either to store the information for later

Table 5.1: Results Summary: Size of Example Designs and Total Analysis Time For All $(x, \mathcal{C})$ Pairs

Design	**Lines of Code**		**# 2NAND**		**Time (sec.)**	
	Orig.	Trj.	Orig.	Trj.	Orig.	Trj.
Adder	614	616	839	877	0.61	0.69
UART	2269	2273	5829	5836	8.59	8.63

retrieval by the attacker or transmit the information outside the chip. The Trojans created for the case studies are non-trivial and representative of malicious functionality that can be inserted into any bus peripheral, which in a typical SoC includes most IP blocks. The on-chip bus is a critical component, and a high value target for an attacker.

We apply both versions of the detection methodology (SMT solving and equivalence checking) to the designs in order to demonstrate the strengths and weaknesses of both approaches. The Trojans inserted in both designs are successfully identified by the method employing an SMT solver, and in Section 5.6.3 we discuss the limitations of using equivalence checking tools for Trojan detection.

Table 5.1 summarizes the experimental results, giving the size of the two original and Trojan-infested designs in lines of code and 2-input NAND gates. The gate count was determined by synthesizing the design using Synopsys Design Compiler (ver I-2013.12-SP2) with a freely available 45nm technology library from NanGate [92]. Table 5.1 also lists the total time in seconds spent on Trojan detection using the SMT solving approach. This includes parsing all design files and building the data-flow graph for each signal in addition to determining the satisfiability of Equation 5.2 for each $(x, \mathcal{C})$ pair and primary output. The experiments were run on a Dell Optiplex 960 computer with 8GB RAM.

5.6.1 Adder Coprocessor

A simple coprocessor takes input from an AMBA AXI4-Lite [56] bus interface. This coprocessor adds 3 8-bit values: a, b, and c. The programmer communicates with the coprocessor by reading and writing to registers in the peripheral.

Trojan Description

Although the registers in the adder coprocessor are allocated 32-bits in the addressing scheme, many only use a single bit. For example, the user writes to the least significant bit of the *b_ap_vld* register to indicate that the b operand is valid, and the rest of the bits go unused. The inserted Trojan circuitry takes advantage of this fact by storing 4 bits of data leaked over the bus interface in *b_ap_vld[5:1]* for a malicious software program to read. Unused register bits are an example of unspecified functionality common in many designs, making the inserted Trojan an ideal example to test our detection strategy.

It is assumed that the leaked data is otherwise not accessible by the adversary and they have access to registers in the coprocessor. The data is leaked using the Trojan communication channel proposed in Chapter 4, which only alters bus signals at the coprocessor's interface when they are not in use. While the inserted Trojan modifies unspecified functionality present in the AXI4-Lite protocol, insertion of Trojan communication channels generalizes to all bus protocols and interconnect topologies.

4-bit Trojan data is delivered on the AXI-Lite write data channel signal *WDATA[3:0]* when $\neg WVALID \wedge WSTRB == 4'b1111$. The Trojan residing in the coprocessor recognizes that when $WSTRB == 4'b1111$, the data present on the $WDATA$ signal is from the Trojan communication channel and then stores *WDATA[3:0]* in *b_ap_vld[5:1]*.

Table 5.2: Trojan Detection Results for Adder Coprocessor: Analysis Time in Seconds, and Outputs SAT for Each (x, $\mathcal{C}$) Pair

$\boldsymbol{x}$	$\boldsymbol{\mathcal{C}}$	**Time, Outputs SAT**	
		Orig.	Trj.
AWADDR[31:0]	¬AWVALID	0.13, *None*	0.13, *None*
WDATA[31:0]	¬WVALID	0.13, *None*	0.17, RDATA
WSTRB[3:0]	¬WVALID	0.13, *None*	0.17, RDATA
ARADDR[31:0]	¬ARVALID	0.10, *None*	0.10, *None*

Description of (x, $\mathcal{C}$) Pairs

Input to the adder coprocessor module comes from the AXI4-Lite bus interface. AXI4 defines 5 independent transaction channels seen at the interface of every master and slave, with each channel employing a $VALID/READY$ handshake signal pair to indicate when the receiver is ready to process bus data, and to mark when valid data is on the bus. We refer the reader to [56] for further detail on the AXI4-Lite protocol.

From the perspective of Trojan detection, we wish to determine if any information received on the bus interface on any channel while the channel $VALID$ signal is 0 can influence any data read out from the registers of the adder coprocessor. Since the adder coprocessor is a bus slave, the inputs (x signals) are those shown in the first column of Table 5.2, with $\mathcal{C}$ being the condition that the corresponding channel $VALID$ signal is 0.

The signals listed in Table 5.2 cover all the inputs to the adder coprocessor excluding the clock and reset signals, meaning that this list of (x, $\mathcal{C}$) pairs is complete for guaranteeing that the adder is not processing leaked information.

Detection Results

The primary outputs for the adder coprocessor are shown in Table 5.3 along with the number of nodes in the data-flow graph for each output in the Trojan-free and Trojan-

Table 5.3: Number of Nodes in the Data-Flow Graph for Each Primary Output in the Adder Coprocessor Design

Output Name	**# Nodes**	
	Orig.	Trj.
AWREADY	45	45
WREADY	45	45
BRESP[1:0]	2	2
BVALID	45	45
ARREADY	34	34
RDATA[31:0]	12507	16505
RRESP[1:0]	2	2
RVALID	34	34
interrupt	3294	4326

infested versions of the design. The satisfiability of Equation 5.2 is determined for every output in Table 5.3 for every (x, $\mathcal{C}$) pair in Table 5.2. Each row in Table 5.2 gives the time taken (in seconds) to build and analyze Equation 5.2 for all outputs using a specific (x, $\mathcal{C}$) pair and lists any outputs for which Equation 5.2 was satisfiable (if any) for both the Trojan-free and Trojan-infested versions of the adder.

For the Trojan-free design, none of the (x, $\mathcal{C}$) pairs were able to influence primary outputs, which is consistent with our assumption that unspecified functionality should not be capable of significantly affecting design behavior. For the Trojan-infested design, our analysis was able to highlight the signals involved in the Trojan circuitry. *WDATA* and *WSTRB* are shown to influence *RDATA*, which is expected, because the Trojan alters the contents of the *b_ap_vld* register, which can be read by an AXI4-Lite read request. The read and write address signals are not involved in the Trojan, and are not flagged by our detection methodology.

The adder coprocessor design is also analyzed using the detection methodology version requiring Conformal, presented in Section 5.5.3, and produces results identical to those shown in Table 5.2.

5.6.2 UART Communication Controller

We also validate our detection methodology on a UART circuit from OpenCores [54] designed to interface with a host processor through a Wishbone Bus Interface [55].

Trojan Description

To transmit data serially on the UART output *stx_pad_o*, 8 bits are written to the UART transmit data register through a Wishbone write transaction. Normally only a bus master is able to issue write requests and cause the UART to transmit data, however if a Trojan communication channel is inserted inside the Wishbone bus, it is possible for another slave to covertly signal the Trojan inserted in the UART controller, and cause data to be serially transmitted outside the chip.

The Trojan inserted in the UART allows writes to UART registers when the signal *wb_sel_i* == 4′*b*1001. The Wishbone bus uses 32-bit data signals, but the UART registers are only 8-bits wide. *wb_sel_i* marks which byte lanes contain valid data. For the UART controller, any values of *wb_sel_i* with Hamming Weight > 1 are "don't care."

In addition to *wb_sel_i*, there are several additional control signals seen at the UART bus interface driven by the bus master. *wb_cyc_i* indicates if a valid bus transaction is in progress, *wb_stb_i* selects the slave, and *wb_we_i* indicates if the transaction is read or write since the address signal is shared by read and write transactions.

A valid write transaction is marked by *we_o*, whose assignment is given by the following code:

```
assign we_o =  (wb_we_is & wb_stb_i & wb_cyc_i & wbstate==2'b10);
```

The Trojan modifies this assignment to the following:

```
assign we_o =  (wb_we_is & wb_stb_i & wb_cyc_i & wbstate==2'b10)
     | (wb_sel_is == 4'b1001 & ~re_o);
```

Table 5.4: Trojan Detection Results for UART Core: Analysis Time in Seconds, and Outputs SAT for Each (x, $\mathcal{C}$) Pair

x	$\mathcal{C}$	Time, Outputs SAT	
		Orig.	Trj.
wb_adr_i[4:0]	¬wb_stb_i ∨ ¬wb_cyc_i	1.54, *None*	1.63, (int_o, dtr_pad_o, stx_pad_o, rts_pad_o, baud_o)
wb_dat_i[31:0]	¬wb_stb_i ∨ ¬wb_we_i ∨ ¬wb_cyc_i	3.00, *None*	2.82, (int_o, dtr_pad_o, stx_pad_o, rts_pad_o, baud_o)
wb_sel_i[3:0]	¬wb_stb_i ∨ ¬wb_we_i ∨ ¬wb_cyc_i	2.95, *None*	3.07, (int_o, dtr_pad_o, wb_ack_o, stx_pad_o, rts_pad_o, baud_o)

This modification allows registers to be written to even when the slave is not selected or a valid transaction is not occurring, provided a read transaction is not taking place.

Description of (x, $\mathcal{C}$) Pairs

Table 5.4 gives the (x, $\mathcal{C}$) pairs analyzed. Similar to the adder coprocessor, the goal is to determine if bus signals can significantly affect the design during conditions in which they are functionally irrelevant. Since the UART is a bus slave, the inputs are the Wishbone address, data, and select signals, which are unspecified when the slave is not selected ($\neg wb\_stb\_i$), or a valid bus cycle is not in progress ($\neg wb\_cyc\_i$). The data and select signals are also unspecified when a read transaction is taking place ($\neg wb\_we\_i$).

Detection Results

Table 5.5 lists all the primary outputs in the UART design along with the number of nodes in the data-flow graph for each output in the Trojan-free and Trojan-infested versions. The outputs *wb_dat_o*, *int_o*, and *stx_pad_o* all have a large number of nodes due to the presence of variables assigned within case statements with many branches and

Table 5.5: Number of Nodes in the Data-Flow Graph for Each Primary Output in the UART Design

Output Name	**# Nodes**	
	Orig.	Trj.
wb_dat_o[31:0]	1.48×10^{17}	3.79×10^{17}
wb_ack_o	109	159
int_o	8.45×10^{16}	21.6×10^{16}
stx_pad_o	1.35×10^{9}	3.45×10^{9}
rts_pad_o	586	1426
dtr_pad_o	586	1426
baud_o	5492	13732

several FIFO memories.

Memory representation could be simplified by using the theory of arrays instead of transforming all instances of Verilog arrays into a case statement in which a separate *reg* variable for each memory word is assigned based on the address signal, however PySMT currently does not provide any functions for constructing formulas with arrays.

Despite the large graph sizes, solving Equation 5.2 for all primary outputs for each (x, $\mathcal{C}$) pair takes only a few seconds, as seen in Table 5.4. For the Trojan-free design, there were no false positives, again consistent with our assumption that bus signals should not influence the rest of the peripheral when they are not being used in a valid transaction. Table 5.4 also shows that the Trojan circuitry uses *wb_adr_i* and *wb_dat_i* in addition to *wb_sel_i*. This matches the behavior of the Trojan, which allows any data to be written to any register as long as $wb\_sel\_i = 4'b1001$.

5.6.3 SMT Solving v. Equivalence Checking

We attempted to use the method presented Section 5.5.3 to analyze the UART design using equivalence checking, however ran into an issue resulting from the fact that most of the bus signals, (*wb_dat_i*, *wb_sel_i*, etc.) are latched before use in certain portions of the code, whereas the bus signals in the adder coprocessor were never stored.

Conformal is a *combinational* equivalence checking tool, meaning any inputs to storage elements are considered outputs of the combinational logic (pseudo-primary outputs) and all outputs from storage elements are treated as circuit inputs (pseudo-primary inputs). If x is a direct flip-flop input (ex. in data path pipelining), x is a pseudo-primary output during combinational equivalence checking, meaning $f_{x=0}$ and $f_{x=1}$ are trivially non-equivalent because a pseudo-primary output is tied to different static values in both versions. If non-equivalence of pseudo-primary outputs is ignored, and only actual primary outputs are compared, any Trojans using different values of x (and by extension, different values of the stored version of x, x_q) to affect design outputs will not be detected using the equivalence checking version of our detection strategy.

The fact that combinational equivalence checking cannot be used to verify sequential circuit behavior is well known, and techniques such as bounded sequential equivalence checking and bounded model checking exist to address this limitation. These techniques use time-frame expansion to capture k cycles of sequential behavior in a purely combinational circuit by copying the combinational circuit k times and connecting pseudo-primary outputs in one time-frame to pseudo-primary inputs in the following time-frame (ex. [93]).

If the sequential depth, k, of x is known, the method proposed in Section 5.5.3 can analyze a version of the design expanded k time-frames to detect Trojans such as the ones inserted in the UART design. The version of our detection strategy based on SMT solving does not suffer from the same problem because a latch output variable, x_q will always have the corresponding latch input variable, x, in its data-flow graph, meaning any formula for a primary output containing x_q will also contain x.

5.7 Summary

In this chapter we propose a detection methodology for Trojans in unspecified functionality by formulating detection as a satisfiability problem based on the assumption that unspecified functionality should not influence design outputs. Our detection procedure can be followed using a wide variety of tools and techniques, and we give specifics for Trojan detection using 1) SMT solving and 2) equivalence checking, meaning our method is applicable to both RT and gate-level designs. We apply our detection methodology to an adder coprocessor and UART communication controller and successfully and efficiently detect Trojan-infested versions of both designs. The inserted Trojans process information leaked through the bus interface, exploiting the fact that bus protocols only partially specify signal behavior. Our methodology is the first to provide a detection mechanism for this Trojan type.

Chapter 6

Trojan Detection Using Exhaustive Testing of k-bit Subspaces

6.1 Introduction

This chapter presents a *post-silicon* Trojan detection methodology which generates test vectors targeting Trojans with rare triggering conditions. Our method overcomes the shortcomings of existing post-silicon detection methodologies, which focus on controllability and observability metrics meaning they cannot be applied to cryptographic hardware where all plaintext bits have equal controllability.

Existing techniques used to detect hardware Trojans in a large chip population fall into two main categories:

1. Techniques identifying anomalies in chip side-channel characteristics such as power consumption and delay

2. Techniques identifying Trojans in the functional domain by improving circuit observability and activation likelihood

Techniques which identify chips containing Trojans by comparing side-channel characteristics such as power [94, 17] and delay [95] with Trojan-free chips or models derived from the Trojan-free netlist, unlike functional testing, have the ability to identify malicious behavior which does not affect any values of *known* circuit nodes. However, side-channel detection methods face ever-increasing process variation, which can overshadow the influence a Trojan has on the chip signature, especially for large complex designs such as SoCs. Beyond this, these focus mainly on the detection mechanism, and still rely on the ability of the test vectors or placement of scan flip-flops to partially or fully activate the Trojan. Determining which design states should be explored during testing to activate Trojan circuitry, and how to best propagate Trojan behavior to observable points is an important task, on which the effectiveness of both side-channel and functional detection methods rely on.

This chapter presents a post-silicon test vector generation strategy, especially applicable to cryptographic hardware, that detects Trojans with triggers based on patterns and sequences of digital signals. A stealthy Trojan has a very small probability of being activated during both the verification effort and during normal operation, but is relatively easy for the adversary to force.

Many existing methods for post-silicon test vector generation use node controllability and observability to bias the test set [96, 97, 98, 99]. The assumption is that in order to decrease Trojan activation probability, the adversary will select signals that have very low 0 or 1-controllability, making the combination of these rare values unlikely to occur during testing. These strategies first identify random-pattern resistant nodes in the circuit, and their corresponding rare values, then derive an optimal test set to trigger low probability node values multiple (N) times. The usability of a Trojan from the attacker's point of view is severely diminished if the attacker cannot reliably control the triggering signals. Existing methods assume all inputs are attacker controllable, hence every single node is

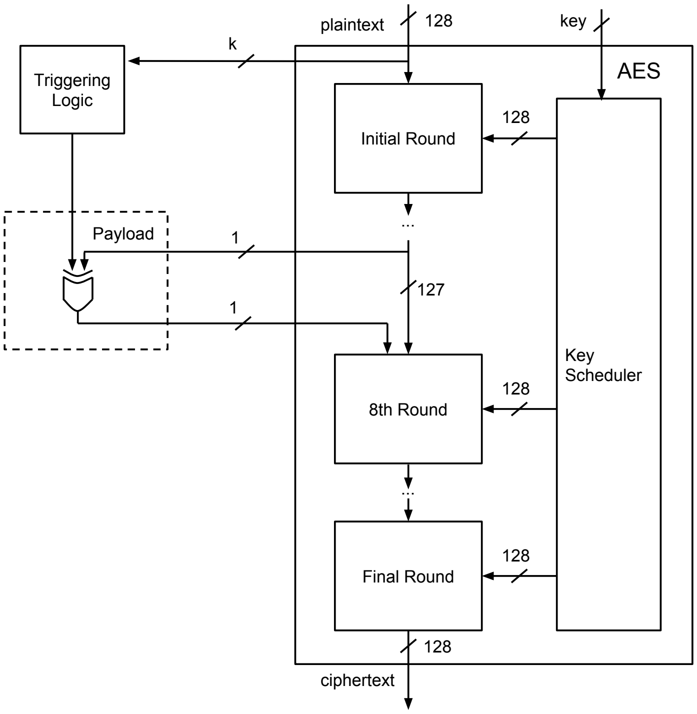


Figure 6.1: AES Fault Attack Trojan

a candidate triggering signal for an attacker with complete knowledge of the design.

In cryptographic hardware, the key bits are unknown to the attacker, therefore any internal circuit nodes influenced by key bits will be uncontrollable, hence cannot be reliably used as triggering signals. For example, in AES, the first step is to XOR the plaintext with key bits [100], leaving the plaintext bits as the only viable triggering signals. For many Trojans proposed for AES [15, 16] and RSA [17], this is indeed the case, and testing strategies based on rare circuit values are not applicable since all plaintext bits have identical controllability and observability.

Challenge-response protocols implemented on both servers and embedded systems such as smart cards, allow the challenger to select the plaintext. In the case of AES, this gives the attacker 128 bits to choose from. Figure 6.1 illustrates the AES Trojan implemented in [15] and [16]. Both works use a subset, k, of n bits (where $n = 128$), and the Trojan payload implements Piret's differential fault attack [101], allowing recovery of all secret key bits after observing as few as 2 faulty ciphertexts. Even if the attacker utilizes only a small subset of the 128-bit plaintext, unless the verification team can discover which subset the attacker will use, they are left facing the impossible task of verifying all 2^{128} plaintext values.

Since exhaustive testing is infeasible, our solution makes the following observation: **An attacker can realistically only afford to use a small subset, $\boldsymbol{k}$, of all $\boldsymbol{n}$ possible controllable signals for triggering.** Our Trojan detection strategy uses this observation, instead of controllability and observability metrics, to reduce the state space targeted by the test vectors. To our knowledge, we are the first to address scenarios where controllability and observability metrics do not provide a foothold for biasing testing. Our approach is exhaustive, but with respect to k instead of n, meaning that our test vectors are *guaranteed* to activate a Trojan if the k-value chosen is realistic.

Section 6.4.2 discusses the factors involved in determining k, but intuitively a hardware Trojan containing a 128-bit comparator or large counter will have a noticeable area and power footprint. The feasibility of inserting such circuits is far less during fabrication, but Trojans inserted pre-silicon have the opportunity to be detected by formal methods, simulation, and analysis of the RTL code. This chapter also provides additional strategies for the case where one cannot afford exhaustive testing with respect to the estimated k-value.

The rest of the chapter is organized as follows: Section 6.2 specifies the class of Trojans our solution detects and relates their activation to the concept of k-subspace coverage, Section 6.3 details how to generate test vectors providing exhaustive k-subspace coverage, Section 6.4 presents a case study where different Trojan triggers are inserted in a 128-bit AES circuit and we discuss how area overhead metrics can influence the selection of k, and Section 6.5 summarizes our contributions.

6.2 Problem Definition and Formulation

6.2.1 Interaction with Existing Test and Detection Methods

Traditional manufacturing tests target stuck-at or delay fault models, based on circuit structure. Trojans inserted during or after fabrication are not present in the gate-level model therefore are not targeted by test pattern generation tools or candidates for observation points.

Because of the confusion and diffusion properties of cryptographic algorithms, the difficultly in Trojan detection lies in triggering the Trojan, not propagating faulty behavior cause by the Trojan to observable points. For example, a Trojan payload may shorten the number of rounds in a cipher or create a fault during encryption. In both cases, the resulting cipher text will differ from the Trojan free version, and is easily detectable.

Therefore, for the remainder of the paper we focus on test generation strategies for trigger activation, and do not address the class of Trojans which leak circuit information through side channels. However, our method can be used in conjunction with existing side channel detection methodologies to magnify the difference between Trojan and Trojan-free side channel fingerprints in the case where information leakage only occurs after a triggering condition is met.

6.2.2 Trojan Trigger Models

Our work aims to detect Trojans whose digital triggers take as input k design signals, where $0 < k \leq n$, and n is the total number of attacker controllable signals.

For the attacker, increasing k *decreases* the probability that the Trojan is triggered during testing, however an increase in k leads to a larger Trojan area and power footprint, making the circuitry more visible.

Table 6.1: Activation Probabilities during a sequence of t n-bit uniform random test vectors for each trigger model

Combinational	**Partial Ordering**
$1-\left(1-\frac{1}{2^k}\right)^t$	$1-\left(1-\frac{1}{2^{km}}\right)^{\binom{t}{m}}$

Contiguous Ordering
$1-\left(1-\frac{1}{2^{km}}\right)^{t-m+1}$

We consider 3 classes of k-bit triggers:

1. **Combinational:** activation occurs immediately upon recognition of a k-bit pattern
2. **Partially Ordered Sequence:** activation occurs upon recognition of a partially ordered sequence of m k-bit patterns
3. **Contiguously Ordered Sequence:** activation occurs upon recognition of a contiguous sequence of m k-bit patterns

The Trojan activation probabilities during a sequence of t n-bit uniform random test vectors for trigger classes 1 - 3 are given by the equations in Table 6.1. Depending on the desired Trojan area overhead and activation probability, the attacker can implement any of the 3 trigger types inside the Triggering Logic block in Figure 6.1.

It should be noted that for the partial and contiguous orderings, the m patterns need not be unique. However, implementing even a few different k-bit pattern recognizers leads to an significant increase in Trojan area, as seen in Table 6.7 in Section 6.4.1, without decreasing activation probability. Therefore, it is very reasonable to assume that only 1 k-bit pattern is used in conjunction with a counter.

Counting m patterns before triggering greatly reduces activation probability. A special case of trigger classes 2 and 3 is a large counter that counts clock cycles instead of patterns. [14] refers to this type of trigger as a "time bomb", and proposes periodically

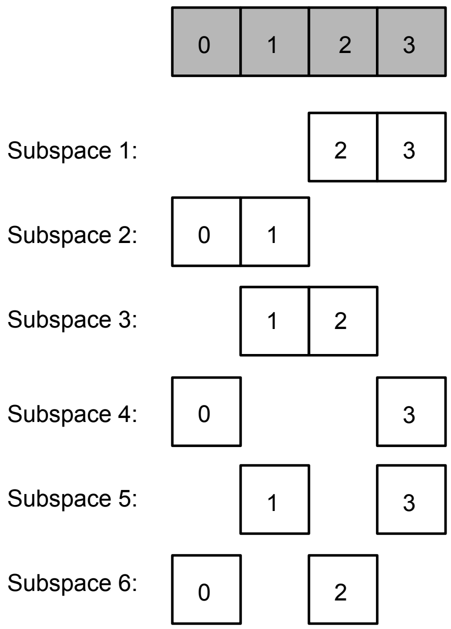


Figure 6.2: All 6 2-bit Subspaces in a 4-bit Vector

performing power resets during circuit operation to limit the maximum counter value, forcing the attacker to use a smaller counter if the Trojan is ever to trigger in the field. If circuit validation is run for a time period exceeding the power reset period, the Trojan is guaranteed to be triggered. For the more general class of sequential triggers that we are considering, power resets effectively limit the value of m, but the test vectors must still ensure the appearance of the magic k-bit pattern m times before activation.

6.2.3 Subspace Coverage

Let n be the number of possible attacker-influenced circuit nodes. Some examples are the plaintext bits in cryptographic hardware or bus data bits on a processor running untrusted software or firmware. If a Trojan can incorporate a maximum of k bits into its triggering mechanism, the goal of the detection effort is to apply the smallest number of n-bit test vectors, $|T_{min}(n, k)|$, which exhaustively cover all 2^k possible values that can occur on all $\binom{n}{k}$ possible sets of k-bit signals (k-subspaces), guaranteeing Trojan activation.

Figure 6.2 shows all 6 possible 2-subspaces when $n = 4$. One simple method of generating the test vectors for this set is to target each subspace individually, resulting in $\binom{n}{k} \times 2^k = \binom{4}{2} \times 2^2 = 24$ test vectors. However, since only 16 test vectors are needed to exhaustively test 4-bits, it is obvious that this method does not generate $T_{min}(n,k)$. An example exhaustive 2-subspace test set generated by trial and error contains only 5 vectors: $\{0000, 0111, 1110, 1101, 1011\}$. These 5 vectors guarantee activation of a trigger using *any* 2 out of 4 controllable bits matching *any* 2-bit pattern. Clearly, $2^k \leq |T_{min}(n,k)| \leq 2^n$, but it is not obvious how to generate $T_{min}(n,k)$ systematically, or determine $|T_{min}(n,k)|$.

6.3 Our Solution

6.3.1 Test Generation for Exhaustive k-subspace Coverage

A method for generating several sets of n-bit test vectors which exhaustively cover all k-subspaces is given in [102]. Each test set is composed of 1 or more sets of *constant weight vectors*. A set of constant weight vectors is the set of *all* n-bit vectors with a given Hamming weight w. There are $\binom{n}{w}$ vectors in a constant weight set.

There are $n-k+1$ test sets to choose from, and each is described by a set of weights, which are found by solving Equation 6.1 with $n-k+1$ different values for c.

$$w \equiv c \bmod (n-k+1),\ 0 \leq c \leq n-k \tag{6.1}$$

Table 6.2: Test Sets for $n = 8$, $k = 3$

$\boldsymbol{c}$	**Weight Set**	**Test Length, $\lvert\boldsymbol{T(8,3)}\rvert$**
0	$\{0, 6\}$	29
1	$\{1, 7\}$	16
2	$\{2, 8\}$	29
3	$\{3\}$	56
4	$\{4\}$	70
5	$\{5\}$	56

The number of test vectors in each test set is

$$\sum_{all\ weights} \binom{n}{w_i} \tag{6.2}$$

For example, let $n = 8$ and $k = 3$. Equation 6.1 becomes

$$w \equiv c \bmod (6),\ 0 \leq c \leq 5 \tag{6.3}$$

There are 6 different test sets which can exhaustively cover all 3-subspaces. The weights and test lengths are given in Table 6.2. Clearly, not all generated test sets are optimal. The test set composed of all vectors with Hamming weights 1 and 7 is the smallest. The weights for the smallest test set are given by Equation 6.4. The size of the minimal test set, $|T_{min}(n, k)|$, is given by Equation 6.5 [102].

$$w_0 = \left\lfloor \frac{k}{2} \right\rfloor,\ w_1 = \left\lfloor \frac{k}{2} \right\rfloor + (n - k + 1) \tag{6.4}$$

$$|T_{min}(n, k)| = \binom{n}{\lfloor \frac{k}{2} \rfloor} + \binom{n}{k - \lfloor \frac{k}{2} \rfloor - 1} \tag{6.5}$$

Table 6.3: Test Set Length for Exhaustive k-subspace Coverage

$\boldsymbol{n}$	$\boldsymbol{m}$	$\boldsymbol{k}$	**Test Set Length**
128	1	2	2^{7}
128	1	4	2^{13}
128	1	8	2^{23}
128	1	16	2^{40}
128	1	32	2^{67}
128	4	8	2^{25}
128	8	8	2^{26}
128	10000	8	2^{37}
256	1	8	2^{27}
256	10000	8	2^{41}
2048	1	8	2^{39}
2048	10000	8	2^{53}

6.3.2 Sequential Triggers

If the Trojan is triggered by partially or continuously ordered sequences of m k-bit patterns, the minimal k-subspace exhaustive test set, $T_{min}(n, k)$, provided in the previous section does not guarantee activation.

Partial Ordering: All *partially* ordered sequences of m k-bit patterns can be exhaustively tested using $|T_{min}(n, k)| \times m$ vectors by repeating $T_{min}(n, k)$ m times.

Contiguous Ordering: If the m patterns can be distinct, then $|T_{min}(n, k)|^m$ test vectors are needed to exhaustively cover this scenario! If the same k-bit pattern, occurring m times in a row, triggers the Trojan, we can repeat each vector in $T_{min}(n, k)$ m times to exhaustively cover this case using only $|T_{min}(n, k)| \times m$ test vectors.

6.3.3 Example Test Set Sizes

Table 6.3 illustrates how n, m, and k affect the test set size for exhaustive k-subspace testing. For Trojans with $m > 1$, Table 6.3 shows the test length assuming either partial ordering of m possibly distinct patterns, or contiguous ordering of m identical patterns. Increasing k and n results in exponential growth in test size. Increasing m causes linear growth in test size, and can be limited by using the power reset technique [14].

6.3.4 When Exhaustive k-subspace Testing is Too Expensive

Although exhaustive k-subspace coverage requires fewer than 2^n vectors, test size grows exponentially with increases in n and k, as seen in Table 6.3, in some cases making exhaustive testing infeasible. Let k_{max} be the maximum number of bits a Trojan trigger can utilize, and T_{max} be the number of vectors budgeted for testing. When $|T(n, k_{max})| \leq T_{max}$, the exhaustive k_{max}-subspace test set is both guaranteed to activate the Trojan and within T_{max}. However, when $|T(n, k_{max})| > T_{max}$, other testing strategies must be considered.

Since our method aims to detect Trojans in designs where signal controllability and observability cannot guide test vector selection, the only alternative testing strategy is the application of uniform random vectors to the attacker controllable bits in the design. Depending on T_{max}, n, k, and k_{max}, one can consider the following test sets:

- **Strategy 1:** T_{max} uniform random vectors
- **Strategy 2:** The complete exhaustive k-subspace test set where $k < k_{max}$, and $|T(n, k)| \leq T_{max}$
- **Strategy 3:** The complete exhaustive k-subspace test set where $|T(n, k)| \approx \frac{T_{max}}{2}$ in addition to $\frac{T_{max}}{2}$ random vectors (excluding those already in $T(n, k)$)

The Trojan activation probability, p_a, for Strategy 1 is given in Table 6.1, p_a for Strategy 3 is computed using simulation, while the derivation of p_a for Strategy 2 is given below.

Strategy 2 p_a: The probability of observing a random k_{max}-bit pattern in the k-subspace exhaustive test set $T(n,k)$, where $k < k_{max}$, can be derived by considering the possible Hamming weights, w_{troj}, for the k_{max}-bit triggering pattern, where $0 \leq w_{troj} \leq k_{max}$. $T(n,k)$ contains *all* possible n-bit vectors with Hamming weights $\{w_0, w_1\}$ given by Equation 6.1. $T(n,k)$ is guaranteed to activate a Trojan with weight w_{troj} if the $k_{free} = n - k_{max}$ bits unused by the Trojan can be assigned a Hamming weight x such that Equation 6.6 holds.

$$w_{troj} + x = w_0 \text{ or } w_1 \tag{6.6}$$

$$f(T, w_{troj}, k_{max}) = \begin{cases} 0 & \nexists \text{ a solution to Eq. 6.6} \\ 1 & \exists \text{ a solution to Eq. 6.6} \end{cases} \tag{6.7}$$

By enumerating all trigger pattern Hamming weights and considering how many such patterns exist for a given n, the number of patterns detectable by any given $T(n,k)$ can be computed, leading to the formula for activation probability given in Equation 6.8.

$$p_a = \frac{\sum_{w_{troj}=0}^{k_{max}} \binom{k_{max}}{w_{troj}} \times f(T, w_{troj}, k_{max})}{2^{k_{max}}} \tag{6.8}$$

Comparisons and Discussion: Table 6.4 compares the activation probability given for Strategy 2 with the activation probability for T_{max} uniform random vectors when $n = 128$. Table 6.5 shows how the mixed test set (Strategy 3), compares with the uniform random vectors for $n = 128$ and various values of k and k_{max}. $|T(n,k)|$ is the number of test vectors in the exhaustive k-subspace test set, $|T_{rnd}|$ is the number of weighted random vectors used, and $|T_{total}| = |T(n,k)| + |T_{rnd}|$.

Exhaustive k-subspace test sets always have lower activation probabilities than the same number of uniform random vectors when $k_{max} > k$. This is because uniform random vectors sample from the entire space of 2^n possible vectors while $T(n,k)$ is restricted to

Table 6.4: Strategy 2 v. Strategy 1 – Activation Probabilities for $n = 128$

$\boldsymbol{k_{max}}$	k	$\boldsymbol{\lvert T(n,k)\rvert}$	$\boldsymbol{p_a}$ $\boldsymbol{(T(n,k))}$	$\boldsymbol{p_a}$ **(rand)**
16	4	2^{13}	0.00235	0.1184
16	8	2^{23}	0.04904	0.9999
32	4	2^{13}	1.309e-07	1.922e-06
32	8	2^{23}	1.093e-05	0.00256
32	16	2^{40}	0.004551	0.9999
64	4	2^{13}	1.163e-16	4.476e-16
64	8	2^{23}	3.919e-14	5.968e-13
64	16	2^{40}	3.163e-10	8.263e-08

Table 6.5: Strategy 3 v. Strategy 1 – Activation Probabilities for $n = 128$

$\boldsymbol{k_{max}}$	$\boldsymbol{k}$	$\boldsymbol{\lvert T(n,k)\rvert, \lvert T_{rnd}\rvert,}$ $\boldsymbol{\lvert T_{total}\rvert}$	$\boldsymbol{p_a}$ **(mixed)**	$\boldsymbol{p_a}$ **(rand)**
8	3	$256, 256, 512$	0.6574	0.8652
16	3	$256, 256, 512$	0.0036	0.007782
20	3	$256, 256, 512$	0.0003	0.0004882
10	5	$2^{14}, 2^{14}, 2^{15}$	0.9999	0.9999
16	5	$2^{14}, 2^{14}, 2^{15}$	0.213	0.3911
20	5	$2^{14}, 2^{14}, 2^{15}$	0.017	0.03053

vectors of particular Hamming weights. As k_{max} becomes larger compared to k, there are more Trojan trigger Hamming weights not targeted by exhaustive k-subspace vectors that uniform random vectors still sample from.

The advantage of using exhaustive k-subspace test vectors for a feasible k is that **activation for all subspaces smaller than $\boldsymbol{k}$ is guaranteed.** Because hardware Trojan insertion is challenging, especially during fabrication, k values smaller than k_{max} are more likely. If random vectors can be used in combination (Strategy 3) to target the less likely larger k values, the activation probability is closer to that of the uniform random vectors. Strategy 3 provides a balance between guaranteed activation for smaller k and optimal sampling of the remaining state space with random vectors.

Table 6.6: % Area Increase and **G:** Gate Count (Equivalent 2-input NAND Gates) v. k and m (identical patterns) for $n = 128$

	m							
	1		128		1024		8192	
k	%	**G**	%	**G**	%	**G**	%	**G**
4	0.11	25	0.77	176	1.00	230	1.23	282
8	0.14	32	0.83	190	1.06	243	1.29	295
32	0.32	72	1.18	270	1.41	323	1.64	376
64	0.55	125	1.65	377	1.88	430	2.11	482
128	1.01	232	2.58	590	2.82	644	3.04	695

6.4 AES Trojan Case Study

6.4.1 Area Overhead

We have implemented the Trojan in Figure 6.1 for each of the 3 Trojan trigger types shown in Section 6.2.2 in an AES encryption IP from OpenCores [103], where $n = 128$. The infected designs were synthesized in 45nm technology using the NanGate Open Cell Library [92] with Synopsys Design Compiler (ver I-2013.12-SP2) and routed using Cadence Encounter (v09.14) to quantify the overhead due to the trojan logic. The percentage increase in area for the infected design and equivalent 2-input NAND gate count of the Trojan is shown in Table 6.6 for various m and k values.

In Table 6.6, the m patterns are identical, not distinct. It can be seen in Table 6.7 that if the trigger is designed with multiple distinct patterns, the area overhead increases significantly. For example, when $k = 4$ and the number of distinct patterns is 4, the area overhead is already greater than 1% of the original design. The synthesis area and NAND gate-count increase significantly as k and m increase, more sharply with an increase in k than that of m. This is because the value of m doubles with addition of only a single counter bit, while increasing k requires an increase in the comparison logic of the trigger.

Table 6.7: % Area Increase and **G:** Gate Count (Equivalent 2-input NAND Gates) v. k and # of Distinct Patterns for $n = 128$ and $m = 8192$

	# distinct patterns					
	4		8		16	
k	%	**G**	%	**G**	%	**G**
4	1.36	310	1.82	417	2.72	621
8	1.48	338	2.05	470	3.18	728
32	2.17	497	3.46	790	5.98	1368

A limit on m can be enforced by using the power reset strategy outlined in [14].

6.4.2 Factors Influencing k_{max}

Determining the feasibility of different k_{max} values requires formulating a realistic threat model for each design and testing scenario. Our method can target Trojans inserted both pre-silicon and during fabrication, but the ease of Trojan insertion and the variety of detection methodologies available at both stages differs greatly.

Trojan Insertion Pre-silicon: On one hand, Trojans inserted in 3rd party IP have practically no limits on Trojan size, since the customer often only has access to the net list or a pre-routed block, making it difficult to reverse engineer the design and identify Trojans or detect increases in area due to Trojan circuitry. Also, post-silicon side-channel detection methods will fail due to the lack of Trojan-free gate-level models, as well as the lack of golden reference chips.

On the other hand, complete observability during simulation and the availability of formal methods such as equivalence checking provide powerful opportunities for detection strategies such as [104]. If a Trojan is inserted at gate-level (post-synthesis), and attempts to hide within minor changes made during the effort to meet timing and power requirements, the presence of several hundred extra gates will surely be noticed, as is the case for when $k > 32$, and $m > 128$ as seen in Table 6.6. How reverse engineering, code and circuit analysis, and formal methods can be used to either prove $k_{max} = 0$ or determine a reasonable k_{max} to target using our post-silicon exhaustive k-subspace approach is a topic for further research.

Trojan Insertion During Fabrication: Modifying the optical mask to insert Trojans is extremely difficult. Only a few works have actually fabricated circuits containing hardware Trojans and analyzed the complexity of insertion at mask level. In [16], Trojans instrumenting Piret's fault attack on an AES circuit are inserted into the layout using a commercial Engineering Change Order (ECO) placement tool. They vary the number of plaintext bits used in the trigger until the software is no longer able to place the ECO without completely re-routing the entire design.

With a Core Utilization Rate of 99%, the tool cannot place an ECO for a Trojan composed of as few as 16 AND gates. While further research is required to validate this approach for estimating an upper bound on k_{max}, it is clear that k is severely restricted for mask level Trojan insertion, making exhaustive k-subspace testing a feasible and complete method for guaranteeing Trojan activation.

6.5 Summary

Our AES circuit case study shows that realistically, an attacker can only incorporate k out of all n possible controllable signals into a Trojan triggering mechanism, where $k << n$. In this chapter we use this observation instead of the controllability and observability metrics widely employed in existing methods to guarantee detection of Trojans in cryptographic circuits using up to k triggering signals [28]. We also present additional strategies when the size of k requires a prohibitively large exhaustive test set to guarantee detection.

Chapter 7

Conclusions

In this document we have addressed the threat of Hardware Trojans in unspecified design functionality. Due to the complexity of modern chips, a design specification usually only defines a small fraction of behavior. Traditional verification techniques only focus on ensuring the correctness of specified behavior, meaning any modifications or bugs (malicious or accidental) only affecting unspecified functionality will likely go undetected.

Several chapters of this dissertation are dedicated to illustrating how this verification hole allows an attacker with the ability to modify the design to stealthily undermine the security of a system. We have shown that all secret key bits in an Elliptic Curve Processor can be leaked by only modifying RTL don't cares, and that it is possible to create a covert Trojan communication channel on top of existing on-chip bus infrastructure for several common bus protocols by only modifying the on-chip bus interface signals when the channel is idle. This channel is shown to allow an attacker running as an unprivileged software program access to root-user data.

By viewing security as an extension of the verification problem we develop several analysis methodologies based on existing techniques such as equivalence checking and mutation testing which both detect Trojans and increase confidence in the correctness of specified design functionality. These techniques include a Trojan prevention methodology based on equivalence checking that classifies all don't care bits in a design as dangerous or safe, a mutation testing based methodology capable of identifying dangerous unspecified functionality regardless of the abstraction level or class of design analyzed, and a methodology which inspects this dangerous unspecified functionality for the existence of Trojans by formulating Trojan detection in terms of satisfiability. Once Trojan detection is expressed as a satisfiability problem, there is a wide variety of existing tools and techniques which can be employed to detect Trojans. In this document we detail Trojan detection using SMT solvers and data-flow graph analysis for RT-level designs, and combinational equivalence checking for gate-level designs. We also present a post-silicon Trojan detection methodology for Trojans with rare triggering conditions.

Because unspecified functionality is by nature unknown, there is still much work to be done in fully exploring the scope of the Trojan threat in this space. Future work includes exploration of unspecified functionality at higher levels of abstraction such as TLM and SystemC, and analyzing how unused instruction fields (common in almost every instruction set architecture) can be used to encode Trojan operations. The Trojan threat at the hardware/software boundary is another direction for future work as device driver and operating system code interacts closely with hardware, and many of the techniques developed in this thesis may be applicable to detecting software Trojans.

Bibliography

[1] I. Duncan and A. K. McDaniels, "Medstar hack shows risks that come with electronic health records," The Baltimore Sun, April 2016. [Online]. Available: http://www.baltimoresun.com/health/bs-md-medstar-healthcare-hack-20160402-story.html

[2] K. Zetter, "Inside the cunning, unprecedented hack of ukraine's power grid," Wired, March 2016. [Online]. Available: http://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/

[3] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, pp. 461–491, Aug. 2004.

[4] M. Dale, "Verification crisis: Managing complexity in SoC designs," *EE Times*, 2001. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1215507

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

[6] S. Mitra, H.-S. P. Wong, and S. Wong, "Stopping hardware trojans in their tracks," *IEEE Spectrum*, Jan. 2015.

[7] Y. Shiyanovskii, F. Wolff, A. Rajendran, C. Papachristou, D. Weyer, and W. Clay, "Process reliability based trojans through NBTI and HCI effects," in *Adaptive Hardware and Systems (AHS), 2010 NASA/ESA Conference on.* IEEE, 2010, pp. 215–222.

[8] L.-W. Kim, J. D. Villasenor, and Ç. K. Koç, "A trojan-resistant system-on-chip bus architecture," in *Proceedings of the 28th IEEE Conference on Military Communications*, ser. MILCOM'09, 2009, pp. 2452–2457.

[9] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET).* Berkeley, CA, USA: USENIX Association, 2008, pp. 5:1–5:8.

[10] L. Lin, W. Burleson, and C. Paar, "Moles: Malicious off-chip leakage enabled by side-channels," in *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*, Nov 2009, pp. 117–122.

[11] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson, "Stealthy dopant-level hardware trojans," in *Cryptographic Hardware and Embedded Systems (CHES)*, ser. Lecture Notes in Computer Science, G. Bertoni and J.-S. Coron, Eds. Springer Berlin Heidelberg, 2013, vol. 8086, pp. 197–214.

[12] 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.

[13] C. Krieg, A. Dabrowski, H. Hobel, K. Krombholz, and E. Weippl, "Hardware malware," *Synthesis Lectures on Information Security, Privacy, & Trust*, vol. 4, no. 2, pp. 1–115, 2013.

[14] A. Waksman and S. Sethumadhavan, "Silencing hardware backdoors," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP'11, 2011, pp. 49–63.

[15] S. S. Ali, R. S. Chakraborty, D. Mukhopadhyay, and S. Bhunia, "Multi-level attacks: An emerging security concern for cryptographic hardware," in *2011 Design, Automation Test in Europe*, March 2011, pp. 1–4.

[16] S. Bhasin, J. L. Danger, S. Guilley, X. T. Ngo, and L. Sauvage, "Hardware trojan horses in cryptographic ip cores," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, Aug 2013, pp. 15–29.

[17] D. Agrawal *et al.*, "Trojan detection using ic fingerprinting," in *Security and Privacy, IEEE Symposium on*, 2007.

[18] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of stealthy malicious logic using boolean functional analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS'13.* ACM, 2013, pp. 697–708.

[19] D. Sullivan, J. Biggers, G. Zhu, S. Zhang, and Y. Jin, "FIGHT-Metric: Functional identification of gate-level hardware trustworthiness," in *Proceedings of the 51st Annual Design Automation Conference, DAC'14.* ACM, 2014, pp. 173:1–173:4.

[20] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, "VeriTrust: Verification for hardware trust," in *Proceedings of the 50th Annual Design Automation Conference, DAC'13.* ACM, 2013, pp. 61:1–61:8.

[21] M. Hicks *et al.*, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP'10.* IEEE Computer Society, 2010, pp. 159–172.

[22] N. Fern, S. Kulkarni, and K.-T. Cheng, "Hardware Trojans hidden in RTL don't cares - Automated insertion and prevention methodologies," in *Test Conference (ITC), IEEE International*, Oct 2015, pp. 1–8.

[23] N. Fern and K.-T. Cheng, "Detecting hardware trojans in unspecified functionality using mutation testing," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD'15*. IEEE Press, 2015, pp. 560–566.

[24] P. Lisherness, N. Lesperance, and K. T. Cheng, "Mutation analysis with coverage discounting," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 31–34.

[25] N. Fern, I. San, Ç. K. Koç, and K. T. Cheng, "Hardware trojans in incompletely specified on-chip bus systems," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 527–530.

[26] S.-Y. Huang and K.-T. Cheng, *Formal Equivalence Checking and Design Debugging.* Kluwer Academic Publishers, 1998.

[27] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011.

[28] N. Lesperance, S. Kulkarni, and K.-T. Cheng, "Hardware trojan detection using exhaustive testing of k-bit subspaces," in *The 20th Asia and South Pacific Design Automation Conference*, Jan 2015, pp. 755–760.

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

[30] G. Qu and L. Yuan, "Secure hardware ips by digital watermark," in *Introduction to Hardware Security and Trust.* Springer New York, 2012, pp. 123–141.

[31] C. Dunbar and G. Qu, "Designing trusted embedded systems from finite state machines," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 5s, pp. 153:1–153:20, Oct. 2014.

[32] R. A. Bergamaschi, D. Brand, L. Stok, M. Berkelaar, and S. Prakash, "Efficient use of large don't cares in high-level and logic synthesis," in *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, Nov 1995, pp. 272–278.

[33] M. Turpin, "The dangers of living with an x (bugs hidden in your verilog)," in *Boston Synopsys Users Group (SNUG)*, October 2003.

[34] L. Piper and V. Vimjam, “X-propagation woes: Masking bugs at rtl and unnecessary debug at the netlist,” in *Design and Verification Conference and Exhibition (DVCon)*, 2012.

[35] H. Z. Chou, H. Yu, K. H. Chang, D. Dobbyn, and S. Y. Kuo, “Finding reset nondeterminism in rtl designs - scalable x-analysis methodology and case study,” in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 1494–1499.

[36] “Cadence conformal equivalence checker.” [Online]. Available: http://www.cadence.com/products/ld/equivalence_checker

[37] M. Turpin, “Solving verilog x-issues by sequentially comparing a design with itself. you’ll never trust unix diff again!” in *Boston Synopsys Users Group (SNUG)*, 2005.

[38] A. R. Bradley, “SAT-based model checking without unrolling,” in *Verification, Model Checking, and Abstract Interpretation.* Springer, 2011, pp. 70–87.

[39] G. Cabodi, S. Nocco, and S. Quer, “Improving sat-based bounded model checking by means of bdd-based approximate traversals,” in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 898–903.

[40] C. Wolf, “Yosys Open SYnthesis Suite.” [Online]. Available: http://www.clifford.at/yosys/

[41] “ABC.” [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/

[42] “Jasper x-prop app.” [Online]. Available: http://www.jasper-da.com/products/jaspergold-apps/x-propagation-verification-app

[43] “Atrenta spyglass lint tool.” [Online]. Available: http://www.atrenta.com/pg/2/

[44] “Cadence incisive.” [Online]. Available: http://www.cadence.com/rl/Resources/articles/10Ways_Incisive_13_2.pdf

[45] “Synopsys magellan.” [Online]. Available: http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/Magellan.aspx

[46] J. W. Bos, J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, and E. Wustrow, “Elliptic curve cryptography in practice,” in *Financial Cryptography and Data Security.* Springer, 2014, pp. 157–175.

[47] C. Rebeiro and D. Mukhopadhyay, “High performance elliptic curve cryptoprocessor for FPGA platforms,” in *12th IEEE VLSI Design And Test Symposium*, 2008.

[48] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649–678, 2011.

[49] B. Breech, M. Tegtmeyer, and L. Pollock, “An attack simulator for systematically testing program-based security mechanisms,” in *2006 17th International Symposium on Software Reliability Engineering*, Nov 2006, pp. 136–145.

[50] N. Bombieri, F. Fummi, G. Pravadelli, M. Hampton, and F. Letombe, “Functional qualification of tlm verification,” in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 190–195.

[51] P. Lisherness and K. T. Cheng, “Scemit: A systemc error and mutation injection tool,” in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, June 2010, pp. 228–233.

[52] N. Bombieri, F. Fummi, and G. Pravadelli, “A mutation model for the systemc tlm 2.0 communication interfaces,” in *2008 Design, Automation and Test in Europe*, March 2008, pp. 396–401.

[53] “Synopsys certitude.” [Online]. Available: https://www.synopsys.com/TOOLS/VERIFICATION/FUNCTIONALVERIFICATION/Pages/certitude-ds.aspx

[54] “UART 16550 core.” [Online]. Available: http://opencores.org/project,uart16550

[55] “Wishbone bus.” [Online]. Available: http://opencores.org/opencores,wishbone

[56] *AMBA AXI and ACE Protocol Specification, Issue E*, ARM, 2013.

[57] S. Pasricha and N. Dutt, *On-Chip Communication Architectures: System on Chip Interconnect*. Morgan Kaufmann Publishers Inc., 2008.

[58] L.-W. Kim and J. D. Villasenor, “A system-on-chip bus architecture for thwarting integrated circuit trojan horses,” *VLSI Systems, IEEE Transactions on*, vol. 19, no. 10, pp. 1921–1926, 2011.

[59] M. Henson and S. Taylor, “Memory encryption: A survey of existing techniques,” *ACM Computing Surveys*, vol. 46, no. 4, pp. 53:1–53:26, 2014.

[60] “Arm trustzone controllers.” [Online]. Available: http://www.arm.com/markets/trustzone-controllers.php

[61] D. Wang, “Formal verification of the PCI local bus: A step towards ip core based system-on-chip design verification,” Master’s thesis, Carnegie Mellon University, May 1999.

[62] A. Roychoudhury, T. Mitra, and S. R. Karri, “Using formal techniques to debug the AMBA system-on-chip bus protocol,” in *Design, Automation and Test in Europe Conference and Exhibition, DATE’03*, 2003, pp. 828–833.

[63] R. Luo and H. Tan, “Formal modeling and model checking analysis of the wishbone system-on-chip bus protocol,” in *Proceedings of the Third International Conference on Information Computing and Applications, ICICA’12.* Springer-Verlag, 2012, pp. 211–220.

[64] “Synopsys vip for arm amba.” [Online]. Available: http://www.synopsys.com/Tools/Verification/FunctionalVerification/VerificationIP/amba/Pages/default.aspx

[65] “Amba 4 axi4, axi4-lite and axi4-stream protocol assertions bp063 release note (r0p1-00rel0),” ARM. [Online]. Available: https://silver.arm.com/browse/BP063

[66] *DS768: LogiCORE IP AXI Interconnect (v1.02.a)*, Xilinx Inc., March 2011.

[67] *AMBA 3 APB Protocol v1.0 Specification, Issue B*, ARM, 2004.

[68] “Axi4 bfm.” [Online]. Available: https://github.com/sjaeckel/axi-bfm

[69] “Vivado design suite, 2015.1.” [Online]. Available: http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2015-1.html

[70] *ZedBoard Hardware User’s Guide (v2.2)*, Avnet Inc., 2014. [Online]. Available: http://zedboard.org/

[71] *UG585: Zynq-7000 All Programmable SoC Technical Reference Manual (v1.10)*, Xilinx Inc., February 2015.

[72] “Xillinux.” [Online]. Available: http://xillybus.com/xillinux

[73] *PG034: LogiCORE IP AXI Central Direct Memory Access (v4.1)*, Xilinx Inc., November 2015.

[74] *PG059: LogiCORE IP AXI Interconnect (v2.1)*, Xilinx Inc., April 2016.

[75] *PG078: LogiCORE IP AXI BRAM Controller (v4.0)*, Xilinx Inc., April 2016.

[76] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *Selected Areas in Communications, IEEE Journal on*, vol. 21, no. 1, pp. 5–19, 2003.

[77] J. Newsome and D. Song, “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,” 2005.

[78] D. W. Palmer and P. K. Manna, "An efficient algorithm for identifying security relevant logic and vulnerabilities in rtl designs," in *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on.* IEEE, 2013, pp. 61–66.

[79] T. McComb and L. Wildman, "SIFA: A tool for evaluation of high-grade security devices," in *Information Security and Privacy.* Springer, 2005, pp. 230–241.

[80] X. Li, M. Tiwari, B. Hardekopf, T. Sherwood, and F. T. Chong, "Secure information flow analysis for hardware design: Using the right abstraction for the job," in *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ser. PLAS '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:7.

[81] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, "Verifying information flow properties of firmware using symbolic execution," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 337–342.

[82] "Cadence jaspergold security path verification app." [Online]. Available: http://www.cadence.com/products/fv/jaspergold_security/pages/default.aspx

[83] M. Gario and A. Micheli, "PySMT: a solver-agnostic library for fast prototyping of smt-based algorithms," 2015.

[84] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, 2015, pp. 451–460.

[85] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 93–107.

[86] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2008, pp. 337–340.

[87] C. Barrett *et al.*, "Cvc4," in *Computer Aided Verification.* Springer, 2011, pp. 171–177.

[88] B. Dutertre, "Yices 2.2," in *Computer-Aided Verification*, vol. 8559. Springer, July 2014, pp. 737–744.

[89] F. Somenzi, "Efficient manipulation of decision diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 171–181, 2001.

[90] A. Biere, "Picosat essentials," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, pp. 75–97, 2008.

[91] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2009, pp. 174–177.

[92] "Nangate 45nm open cell library," 2011. [Online]. Available: https://www.si2.org/openeda.si2.org/projects/nangatelib

[93] F. Lu and K.-T. Cheng, "SEChecker: A sequential equivalence checking framework based on k-th invariants," *VLSI, IEEE Transactions on*, vol. 17, no. 6, pp. 733–746, 2009.

[94] Y. Liu, K. Huang, and Y. Makris, "Hardware trojan detection through golden chip-free statistical side-channel fingerprinting," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 155:1–155:6.

[95] K. Xiao, X. Zhang, and M. Tehranipoor, "A clock sweeping technique for detecting hardware trojans impacting circuits delay," *IEEE Design & Test of Computers*, vol. 30, no. 2, pp. 26–34, 2013.

[96] F. Wolff, C. Papachristou, S. Bhunia, and R. S. Chakraborty, "Towards trojan-free trusted ics: Problem analysis and detection scheme," in *2008 Design, Automation and Test in Europe*, March 2008, pp. 1362–1365.

[97] R. S. Chakraborty *et al.*, "MERO: A statistical approach for hardware trojan detection," in *CHES 2009*, ser. LNCS. Springer Berlin Heidelberg, 2009, vol. 5747, pp. 396–410.

[98] S. Narasimhan, D. Du, R. S. Chakraborty, S. Paul, F. Wolff1, C. Papachristou, K. Roy, and S. Bhunia, "Multiple-parameter side-channel analysis: A non-invasive hardware trojan detection approach," in *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*, June 2010, pp. 13–18.

[99] A. Sreedhar, S. Kundu, and I. Koren, "On reliability trojan injection and detection," *J. Low Power Electronics*, vol. 8, no. 5, pp. 674–683, 2012.

[100] "Advanced encryption standard (aes)." [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[101] G. Piret and J.-J. Quisquater, "A differential fault attack technique against spn structures, with application to the aes and khazad," in *Cryptographic Hardware and Embedded Systems (CHES), Sep. 2003*, ser. LNCS, vol. 2779. Springer, 2003, pp. 77–88.

[102] D. Tang and L. S. Woo, "Exhaustive test pattern generation with constant weight vectors," *Computers, IEEE Transactions on*, Dec 1983.

[103] "Aes (rijndael) ip core," 2002. [Online]. Available: http://opencores.org/project, aes_core

[104] M. Banga and M. Hsiao, "Trusted rtl: Trojan detection methodology in pre-silicon designs," in *Hardware Oriented Security and Trust (HOST)*, June 2010, pp. 56–59.