UNIVERSITY OF CALIFORNIA, SAN DIEGO

Testing Hardware Security Properties and Identifying Timing Channels

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

in

Computer Science

by

Jason Kaipo Oberg

Committee in charge:

Ryan Kastner, Co-Chair
Timothy Sherwood, Co-Chair
Ali Irturk
Andrew Kahng
Stefan Savage

2014

The Dissertation of Jason Kaipo Oberg is approved and is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California, San Diego

2014

DEDICATION

To all the family, friends, and teachers who believed and supported me from the little island of Kauai.

EPIGRAPH

Learning does not make one learned: there are those who have knowledge and those who have understanding. The first requires memory and the second philosophy.

Alexandre Dumas

TABLE OF CONTENTS

Signature Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures viii

List of Tables x

Acknowledgements xi

Vita xiv

Abstract of the Dissertation xvii

Chapter 1 Introduction 1
1.1 Overview 1
1.2 Thesis Outline 5
1.2.1 Background and Related Work 5
1.2.2 Deriving Equations for GLIFT Logic 5
1.2.3 Analyzing Timing Channels in Bus Protocols Using GLIFT . . . 6
1.2.4 Testing Timing Channels in a System-on-Chip 6
1.2.5 Formal Analysis of Timing Channels and GLIFT 7

Chapter 2 Background and Related Work 8
2.1 The Cost of Embedded Security 8
2.2 Information Flow Tracking and GLIFT 10
2.2.1 Using GLIFT as a Static Testing Technique 17
2.2.2 Example: Using GLIFT on AES 20
2.3 Timing-Channel Attacks 22

Chapter 3 Fundamental GLIFT Logic Equations and Overheads 28
3.1 Taint and Tracking (GLIFT) Logic 28
3.2 GLIFT Logic for Basic Boolean Gates 29
3.3 GLIFT Logic Overheads 31

Chapter 4 Exploring Information Leaks in Bus Protocols 35
4.1 Information Flows in I^2C 36
4.1.1 Enforcing Non-interference in I^2C 37
4.1.2 I^2C Non-interference Overheads 38

4.2 Information Flows in USB 40
4.2.1 Enforcing Non-interference in USB 43
4.2.2 USB Non-interference Overheads 43

Chapter 5 Testing Timing Information Flows in Larger Systems 46
5.1 Designing a Secure Crossbar in Wishbone 48
5.1.1 Mix-Trusted System with Hardware Accelerator 50
5.1.2 Building a Secure Cross-Bar for WISHBONE 52
5.1.3 Secure Cross-Bar Evaluation 54

Chapter 6 Formalizing Timing Channels at the Gate-level 59
6.1 Threat Model 60
6.2 Preliminary Definitions 60
6.3 Recap–Information Flow Tracking and GLIFT 64
6.3.1 Formal definitions for GLIFT 65
6.3.2 GLIFT and Timing Channels 68
6.4 Isolating Timing Channels 70
6.4.1 Finding Functional Flows 71
6.4.2 A sample usage: fast/slow multiplier 75
6.5 The Bus Covert Channel 76
6.5.1 Identifying Timing Flows in I^2C 78
6.5.2 Overheads 84
6.6 Cache Timing Channel 85
6.6.1 Overview of Access-Driven Timing Attacks 87
6.6.2 Identifying the Cache Attack as a Timing Channel 88
6.6.3 Overheads 91
6.7 Timing Channels in RSA Encryption Core 92
6.7.1 Detecting Leak as Timing Channel 93

Chapter 7 Conclusion 95

Chapter 8 Future Research in Hardware Security 98

Bibliography 101

LIST OF FIGURES

Figure 2.1. Examples of different security lattices. 11

Figure 2.2. Simple code snippet showing information flow. 12

Figure 2.3. GLIFT for a NAND gate. 15

Figure 2.4. GLIFT logic on a 1-bit counter. 16

Figure 2.5. Using GLIFT as a testing technique. 18

Figure 2.6. Using GLIFT to test information flows in AES. 21

Figure 3.1. The number of minterms in the GLIFT logic function of gate primitives. 32

Figure 3.2. The number of minterms in the GLIFT logic function of several benchmarks. 33

Figure 4.1. I^2C Bus configuration. I^2C can support several devices on a single global bus. 36

Figure 4.2. I^2C configured with an additional adapter to enforce TDMA. This enforces non-interference between devices under the presented test conditions. 38

Figure 4.3. Packets sent from the host are broadcast onto the bus to all connected devices. The topology is a tiered star structure. 41

Figure 4.4. How USB leaks information during its broadcasting. 42

Figure 5.1. The WISBHONE bus architecture 49

Figure 5.2. WISHBONE system used to test for information flows. 51

Figure 5.3. Waveform showing tainted information flow in WISHBONE. 54

Figure 6.1. GLIFT tracking logic of an AND gate 67

Figure 6.2. The classes of information flows in hardware. 71

Figure 6.3. How the functional flow testing method can be used to isolate timing channels. 72

Figure 6.4. Simple multiplier example demonstrating how to use our model to isolate timing channels. 75

Figure 6.5. The I^2C bus architecture. 79

Figure 6.6. Adding strict time-partitioning to the I^2C bus. 81

Figure 6.7. How timing attacks are performed on a processor cache. 88

Figure 6.8. A block diagram of a simple MIPS-based CPU. 90

LIST OF TABLES

Table 3.1. GLIFT logic area and overhead for logic benchmarks. 33

Table 4.1. Time spent simulating our particular test scenario for both the original I^2C design and the one with GLIFT. 39

Table 4.2. Area for I^2C components in non-interference compliant design. This is the final system after testing and does not contain GLIFT logic. . 40

Table 4.3. Time spent simulating the mentioned test scenario on USB with and without GLIFT. 44

Table 4.4. Area Overhead for Replicating State Machines 45

Table 5.1. Information flows when different applications on run on WISHBONE system. 53

Table 5.2. Simulation times for various stages of our design. The base RTL refers to what is shown in Figure 5.2 (a). The GLIFT logic incurs at most a 2X overhead in simulation time in this system. 57

Table 6.1. Simulation times in milliseconds associated with the three presented cases for I^2C, and for a single trace. GLIFT imposes a small overhead in the simulation time for these test cases. 84

Table 6.2. Simulation times in seconds for AES running with different encryption keys, with and without GLIFT tracking logic. In general, simulating a design with GLIFT logic causes large slow-downs. . . . 92

ACKNOWLEDGEMENTS

I would first like to give the warmest aloha to everyone back on my home island of Kauai who believed in me. All the family, friends, and high-school teachers who gave motivation and encouragement in an environment which rarely promotes academics.

Although perhaps cliché, I want to acknowledge my advisor Dr. Ryan Kastner who allowed me to do research as a Sophomore undergraduate and really helped spark my interest. I would have never pursued a doctorate degree without his encouragement and support. I also want to thank Dr. Timothy Sherwood who also provided me with some of my intial research advising also as an undergraduate. Dr. Sherwood's former student, Dr. Mohit Tiwari helped me tremendously as both an undergraduate and junior graduate student. I want to acknowledge that his assistance was invaluable in developing the fundamentals presented in this dissertation.

I would also like to thank my committee members. Dr. Ali Irturk provided some crucial assistance in the early development of the work in this dissertation while I was a junior graduate student. Dr. Andrew Kahng provided me with valuable feedback about how this work applies to the hardware design space and how to effectively convey these key points. Dr. Stefan Savage motivated me to focus more attention on security beyond what my background already had. His questions and feedback made me a stronger security researcher and motivated me to think about the practical security applications of research.

None of this would have been possible without my many collaborators. I would like to thank Dr. Cynthia Irvine, Dr. Ted Huffmire, and Dr.Timothy Levin from the Naval Post Graduate School (NPS) who consitently provided me with valuable feedback and constructive criticisms about my research. I would also like to thank Dr. Irvine for the many contributions as a valuable reference. My colleagues at UC Santa Barbara also deserve the greatest thanks and acknowledgement. Dr. Xun Li, Dr. Jonathan Valamehr,

Dr. Mohit Tiwari, Dr. Fredrick Chong, Dr. Bita Mazloom, Dr. Timothy Sherwood, and Dr. Ben Hardekopf have all contributed substantially to my broader research interests and completion of my doctorate.

My many funding sources have been key to my success in my degree. I want to acknowledge the National Science Foundation for their graduate research fellowship and my participation in the Innovation Corps Program where I learned invaluable knowledge about the hardware security industry. I would also like to thank the generous women of the Achievement Rewards for College Scientists (ARCS) scholarship. Their generosity helped relieve much of the financial stress that many graduate students experience.

On a more personal note, I want to thank Dr. Ravi Chugh for the many stress relieving and glorious golf rounds and competitve late night Mario Kart. Dr. Zach Tatlock for answering my questions about programming languages, compilers, and formal verification. Wei Hu for his hard work and helpful support during our close collaboration when he was visiting UCSD between 2010-2012. Our relationship on these projects helped develop the technology in a short amount of time. Last, but certainly not least, Dr. Sarah Meiklejohn for her help with cryptography and mathematical models.

Majority of the work herein appears in both academic journals and conferences. Chapter 3, in small part, is a reprint of the material as it appears in the Design Automation Conference 2010. Oberg, Jason; Hu, Wei; Irturk, Ali; Tiwari, Mohit; Sherwood, Timothy; Kastner, Ryan; The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a reprint of the material as it appears in the Design Automation Conference 2011. Oberg, Jason; Hu, Wei; Irturk, Ali; Tiwari, Mohit; Sherwood, Timothy; Kastner, Ryan; This dissertation author was the primary investigator and author of this paper.

Chapter 5, in full, is a reprint of the material as it appears in the IEEE journal on

Design and Test of Computers 2013. Oberg, Jason; Sherwood, Timothy; Kastner, Ryan. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in full, is a reprint of the material as it appears in the conference on Design Automation and Test in Europe 2013 and also in submission at the IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems 2014. Oberg, Jason; Meiklejohn, Sarah; Sherwood, Timothy; Kastner, Ryan. The dissertation author was the primary investigator and author on both of these papers.

VITA

2009	Bachelor of Science, Computer Engineering University of California, Santa Barbara
2009	Teaching Assistant, Department of Computer Science and Engineering University of California, San Diego
2010	Systems Engineering Intern Qualcomm, San Diego CA
2011	Research Intern Microsoft Research, Redmond WA
2009–2014	Research Assistant, Department of Computer Science and Engineering University of California, San Diego
2012	Master of Science, Computer Science and Engineering University of California, San Diego
2014	Doctor of Philosophy, Computer Science University of California, San Diego

PUBLICATIONS

Sapper: A Language for Hardware-Level Security Policy Enforcement. Xun Li, Vineeth Kashyap, Jason Oberg, Mohit Tiwari, Vasanth Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf and Frederic T. Chong, **International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)**

SurfNoC: A Low Latency and Provably Non-Interfering approach to Secure Networks-On-Chip. Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, Timothy Sherwood, **International Symposium on Computer Architecture (ISCA 2013)**

Expanding Gate Level Information Flow Tracking for Multi-level Security. Wei Hu, Jason Oberg, Janet Barrientos, Dejun Mu, and Ryan Kastner, **IEEE Embedded System Letters, vol. 5, no. 2, May 2013**

Eliminating Timing Information Flows in a Mix-trusted System-on-Chip. Jason Oberg, Timothy Sherwood, and Ryan Kastner, **IEEE Design and Test of Computers, vol. 30, no. 2, March/April 2013**

A Software-based Dynamic-warp Scheduling Approach for Load-Balancing the Viola-Jones Face Detection Algorithm on GPUs. Tan Nguyen, Daniel Hefenbrock, Jason Oberg, Ryan Kastner, and Scott Baden, **Journal of Parallel and Distributed Computing, January 2013**

Sapper: A Language for Provable Hardware Policy Enforcement. Xun Li, Vineeth Kashyap, Jason Oberg, Mohit Tiwari, Vasanth Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf and Frederic T. Chong, **Workshop on Programming Languages and Analysis for Security (PLAS 2013).**

A Practical Testing Framework for Isolating Hardware Timing Channels. Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner, **The conference on Design Automation and Test in Europe (DATE 2013)**

On the Complexity of Gate Level Information Flow Tracking Logic. Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner, **IEEE Transactions on Information Forensics and Security (TIFS), vol. 7, no. 3, June 2012**

Simultaneous Information Flow Security and Circuit Redundancy in Boolean Gates. Wei Hu, Jason Oberg, Dejun Mu, and Ryan Kastner, **The international conference on Computer-Aided Design (ICCAD 2012)**

Random Decision Tree Body Part Recognition Using FPGAs. Jason Oberg, Ken Eguro, Ray Bittner, and Alessandro Forin, **The International conference on Field Programmable Logic and Applications (FPL 2012)**

Trimmed VLIW: Moving Application Specific Processors Towards High Level Synthesis. Janarbek Matai, Jason Oberg, Ali Irturk, Taemin Kim, and Ryan Kastner, **The Electronic System Level Synthesis Conference (ESLsyn 2012)**

Theoretical Fundamentals of Gate Level Information Flow Tracking. Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner, **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, issue 8, August 2011**

Simulate and Eliminate: A Top-to-Bottom Design Methodology for Automatic Generation of Application Specific Architectures. Ali Irturk, Janarbek Matai, Jason Oberg, Jeffrey Su, and Ryan Kastner, **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, issue 8, August 2011**

Enforcing Information Flow Guarantees in Reconfigurable Systems with Mix-trusted IP.

Ryan Kastner, Jason Oberg, Wei Hu, and Ali Irturk, **The conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2011), invited paper**

An Improved Encoding Technique for Gate Level Information Flow Tracking. Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner, **The International Workshop on Logic and Synthesis (IWLS 2011)**

Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. Mohit Tiwari, Jason Oberg, Xun Li, Jonathan K Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood, **In Proceedings of the International Symposium on Computer Architecture (ISCA 2011)**

Information Flow Isolation in I2C and USB. Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner, **In Proceedings of the Design Automation Conference (DAC 2011)**

Caisson: A Hardware Description Language for Secure Information Flow. Xun Li, Mohit Tiwari, Jason Oberg, Frederic T. Chong, Tim Sherwood, and Ben Hardekopf, **In Proceedings of the conference on Programming Language Design and Implementation (PLDI 2011)**

Minimal Multi-Threading: Finding and Removing Redundant Instructions in Multi-Threaded Processors. Guoping Long, Diana Franklin, Susmit Biswas, Pablo Ortiz, Jason Oberg, Dongrui Fan, and Frederic T. Chong, **In Proceedings of the International Symposium on Microarchitecture (MICRO 2010).**

Accelerating Viola-Jones Face Detection to FPGA-Level using GPUs. Daniel Hefenbrock, Jason Oberg, Nhat Tan Nguyen Thanh, Ryan Kastner, and Scott B. Baden, **In Proceedings of the conference on Field-Programmable Custom Computing Machines (FCCM 2010).**

Theoretical Analysis of Gate Level Information Flow Tracking. Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner, **In the Proceedings of the Design Automation Conference (DAC 2010).**

FPGA-Based Face Detection System Using Haar Classifiers. Jung Uk Cho, Shahnam Mirzaei, Jason Oberg, and Ryan Kastner, **In Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA 2009).**

ABSTRACT OF THE DISSERTATION

Testing Hardware Security Properties and Identifying Timing Channels

by

Jason Kaipo Oberg

Doctor of Philosophy in Computer Science

University of California, San Diego, 2014

Professor Ryan Kastner, Co-Chair
Professor Timothy Sherwood, Co-Chair

Computers are being placed in charge of the systems and devices we trust with our safety and security. These embedded systems control our automobiles, commercial airlines, medical devices, mobile phones, and many other aspects that we hope will behave in a secure and reliable manner. In addition, the hardware in these systems are becoming increasingly complex; making security testing and evaluation a very difficult problem. Unfortunately, we have already seen many attacks performed on many of these systems including automobiles and medical devices. Many of these issues could have been prevented had there been better methods for security assessment. Specifically,

hardware and embedded system designers are lacking the tools and methods for testing various security properties of their designs.

Recently, a method known as gate-level information flow tracking (GLIFT) was introduced to dynamically monitor information flows in hardware for security. This dissertaion shows that this same technique can be very effectively applied statically to hardware designs to systematically test various different hardware security properties (e.g. to ensure that secret encryption keys are not leaking). Even further, this thesis demonstrates that GLIFT can effectively capture timing-channels (where information leaks in the amount of *time* a computation takes). These timing channels have been exploited in many past works to extract secret keys from different stateful hardware resources such as caches and branch predictors. This thesis presents some very fundamental background of GLIFT, shows how it can be used statically using several application examples, and formalizes how it can be used to detect timing channels. These contributions ultimately provide a method to do hardware security testing and verification for our future computing systems.

Chapter 1

Introduction

1.1 Overview

Computing systems are at the heart of many of the systems we rely on for our personal safety and security. Automobiles, medical devices, commercial airlines, and mobile phones are becoming inreasingly complicated and we trust that their systems will not behave unexpectedly or leak our secret information. With exploits being exposed in many of these systems including pace-makers [41], insulin pumps [75] and automobiles [57], hardware security is becoming sought after to provide a root-of-trust. By building security in hardware, hardware specific vulnerabilites (which are very difficult to mitigate by software alone) are accounted for as well.

Security mechanisms themselves have existed in hardware for quite some time. They have existed in the form of security rings to limit the resources that processes can access and help enforce isolation between different domains (such as kernel and user space) [35, 95]. Hardware mechanisms have also long existed in the form of privileged instructions where a subset of instructions can only be executed by higher security processes (such as the kernel) [88, 37]. These instructions typically restrict operations such as I/O access and accessing protected memory regions from less privileged processes. Hardware implementations of crypto systems have also been used for quite some time

both in the form of data encryption [96, 106] and hash functions [110, 29]. In all cases, one of the primary reasons these features are designed in hardware is to increase system performance by taking some of the security burden of the software and boosting it with highly parallel and efficient implementations.

As we are seeing hardware becoming more complex, the demand for these hardware security mechanisms is increasing. The hardware itself is becoming an attack vector with fixes that have to be added in the hardware itself. As a result, efforts to combat the need for new hardware security mechanisms has seen a lot of momentum in the recent years. For example, Intel is focusing a substantial amount of attention on security with the continuous growth of its Security Center of Excellence (SeCoE). One of their most recent projects is focused on providing secure enclaves [68] for more secure execution. Specifically, they provide hardware mechanisms for isolation by allowing untrusted applications to execute in special "enclaves" that they build into their instruction set architecture (ISA). In addition, we are seeing an increasing number of hardware security modules (HSMs) being developed by both large and small companies. One example is the company ESCRYPT, which was recently acquired by Bosch [32], who's primary product is HSMs for next generation automobiles. As the hardware itself continues to rapidly progress towards mechanisms for better security, the methods for evaluating the security of these designs is unfortunately far behind. Although the methods for testing and verfication for correct functionality are quite rich in the electronic design and automation (EDA) space, the methods specific for security are extremely limited with most solutions being centered around "best-practices" and security auditing teams.

The complexity of modern hardware makes the intricate security properties that need to be tested and verified difficult. For example, a common property that often needs to be guaranteed in hardware is non-interference [38], where certain parts of the system should never interfere with other parts. An example where non-interference is

important is the Boeing 787 aircraft which has connectivity between the user and flight control networks [33]. In this type of scenario ensuring that there are no unintended information flows between the two networks is critical for the correct operation of the aircraft. This property of non-interference is also what Intel would like to demonstrate with their enclaves, by ensuring that you can have isolation between processes in the hardware. With the further development of intricate system-on-chips interacting via complex protocols, guaranteeing non-interference is a hard problem since information can flow through difficult to detect side channels.

A side-channel is defined as an entity which leaks information but was not intended for communication. The two most common side channels found in hardware are from timing (the data-dependent latency of a computation) and power (the data-dependent power consumed during a computation). In this work we address only logical side-channels (timing), physical ones (power) are out of the scope of this work. As an example, in order to show that two devices on a bus are non-intefering, it is required that the devices not intefere directly (i.e., by corrupt transmitted data) or through timing (i.e., by delays in response time). At first glance, these timing variations might seem benign, but these side-channels have been recently exploited by many to extract secret encryption keys from miss/hit delays in processor caches [10, 18, 86, 40]. These attacks rely on a exploiting information leaking through a *timing channel*, where an attacker is able to deduce information by simply measuring execution time. Since modern hardware is now increasingly coupled with non-determinism and hidden state, methods for detecting and even reasoning about these types of information leaks is an increasingly complex problem. Recently, some work that performs information flow tracking at the level of logic gates started getting traction as a potential way for detecting these timing channels.

Information flow tracking, in general, is a way of label information (e.g. label a key as *secret*) and then tracking where this key “flows”. It is a powerful mechanism

for detecting a potential security issue so that the system can take preventative actions. Information flow tracking has seen lots of attention at many levels of the computing hierarchy. It has been used in high-level languages [94, 89, 42, 16] and compilers [60, 112], binary analysis tools [113, 8, 25, 34], operating systems [114, 58, 31], and in many hardware assisted techniques [28, 27, 98, 104, 105, 93, 23]. However, performing gate-level information flow tracking (GLIFT) was only very recently explored [103]. The properties found at the gate-level (the lowest digital abstraction), specifically using GLIFT, provide a powerful mechanism for testing and proving the absence of harmful information flows in a hardware design; including those from timing channels.

GLIFT associates a label (for a 2-level security lattice, this is a single-bit) with information and then tracks this bit as it flows through boolean gates. In the past, GLIFT has been used to build a non-interfering processor [101], analyze the USB and I2C bus protocols [78], build larger systems with I/O and a microkernel [102], and analyze the security of system-on-chips [80]. In many of these applications, GLIFT was used to show non-interference. However, the specifics of how GLIFT can be used to detect and eliminate timing channels has not been thoroughly explored. **This thesis demonstrates how gate-level information flow tracking (GLIFT) can be used as a static hardware security testing technique and to identify hardware timing channels.** The approach taken is to first understand how information flows at the gate level and extend this understanding to make it possible to test and debug larger hardware security issues at a larger scale. It explores how GLIFT can be used on real designs and also presents a formal model for using GLIFT to detect timing channels. Specifically, This thesis is the first to formalize timing channels in hardware and demonstrate how to separate out these timing flows from other functional ones. To assist with reading this thesis, a detailed organization of each chapter are described next.

1.2 Thesis Outline

In this section, I give a brief description of each chapter in this this thesis, starting with some background and related work.

1.2.1 Background and Related Work

Before I present details of how I have used gate level information flow tracking to identify and eliminate timing channels, I will discuss some crucial background information. I will first present a discussion of the cost associated with security in Chapter 2.1. Next, I will discuss the necessary background and related work on information flow tracking in Chapter 2.2 and how our gate-level analysis can be used as a static testing technique. Lastly, I will provide a detailed discussion of timing channels and how they have been exploited in the past in Chapter 2.3. Following this section, I will discuss the details some theoretical fundamentals of GLIFT.

1.2.2 Deriving Equations for GLIFT Logic

As a first step, this thesis presents some theortical basics for GLIFT. This thesis expands on previous work by providing a theoretical foundation for generating and understanding Shadow (or GLIFT) Logic. Specifically, it 1) precisely defines taint and GLIFT logic; 2) derives logic equations for some gate-level primitives; and 3) discusses the overheads related to the GLIFT logic in terms of the number of minterms and area using several IWLS and ISCAS benchmarks. These three essential points provide a necessary theoretical foundation for GLIFT and are necessary for understanding the other concepts and analyses presented in this thesis. I address these points in detail in Chapter 3.

1.2.3 Analyzing Timing Channels in Bus Protocols Using GLIFT

Beyond the formalisms of GLIFT, this thesis demonstrates how this technique can be used to analyze information flows in some common bus protocols. Since GLIFT provides a solution for monitoring information flows in hardware and targets boolean gates it is general enough to be applied to any digital hardware. Furthermore it can precisely detect all explicit information flows as well as timing channels since it monitors the change of every bit cycle-by- cycle. As a result, this thesis will show that it is very effective for proving information flow policies about common bus protocols such as the Inter-Integrated Circuit protocol (I2C) and the Universal Serial Bus (USB). This thesis discusses how GLIFT can be used to analyze and remove unintended information flows in bus protocols using I2C and USB as examples but presents a general framework for doing so that fits in seamlessly into existing hardware design flows. The general testing framework is presented in Chapter 2 and how this framework can be applied to I2C and USB is discussed in Chapter 4. Although this thesis demonstrates a necessary first step in how GLIFT can be used to test for timing information flows in common bus protocols, it is not clear how it can be applied to larger hardware systems.

1.2.4 Testing Timing Channels in a System-on-Chip

Chapter 5 presents an explanation and scenario in which our GLIFT testing method can be used the test security properties of larger systems.

Specifically, Chapter 5 presents an analysis using two MIPS processors sharing a cryptographic core (the advanced encryption standard) over the WISHBONE [84] system-on-chip (SoC) bus. This chapter demonstrates how GLIFT can be used to test for timing-based information leaks in this larger SoC. This chapter describes the methodology of how it is used and the timing-based integrity properties that need to be upheld. This usage of GLIFT is the first of its kind to demonstrate how it can be used to abstract

away certain details of the design (the processors) and analyze simply the interconnect (the WISBHONE architecture) in order to demonstrate timing isolation between two processors even though they are sharing a common resource.

1.2.5 Formal Analysis of Timing Channels and GLIFT

An additional contribution of this thesis is a formalization of how GLIFT detects timing channels. As briefly mentioned, information leakage through time has been exploited by attackers to extract secret encryption keys in both branch predictors [11] and processor caches [86, 18, 10]. GLIFT has shown to be promising in detecting these timing channels but this claim has not been formally explored in prior work. The goal of Chapter 6 is to make this claim more apparent. We present a framework for identifying timing channels and separate them from other types of information flows.

To show the practicality of our framework, we explore in Chapters 6.5 and 6.6 two common shared resources which are at the heart of interference in modern computing systems: the shared bus and CPU cache. The shared bus in modern systems has been the source of the so called *bus-contention* channel [44] in which information can be covertly communicated through the traffic on a global bus. Previous work has explored how to identify information flows in global buses using GLIFT [78] but has fallen short of classifying these flows as functional or timing. Beyond the bus, we examine in Chapter 6.6 the CPU cache; as previously mentioned, the cache is a common vulnerability in modern systems, as it is typically susceptible to leaking secret information through timing channels. For both examples, we do not make claims about complete information security, but rather increased confidence by identifying the presence of functional information and separating it from timing channels. Before we present our formal model and its use, we outline some essential preliminary definitions in Chapter 6.2 and formal definitions of GLIFT and information flow tracking using GLIFT in Chapter 6.3.

Chapter 2

Background and Related Work

2.1 The Cost of Embedded Security

Hardware and embedded systems are seeing an increasing number of exploits that are costing companies millions of dollars [100]. This is primarily due to the lack of, systematic, formal methods for identifying hardware-specific security vulnerabilities (including timing channels) are non-existent. With the increasing complexity of our hardware, these vulnerabilities are becoming increasingly more of an issue.

For example, Intel states about the core i7-900 that "in some rare cases improper TLB invalidation may result in unpredictable system behavior and can hang your OS or result with incorrect data" [50]. While these bugs may not occur in common place execution, a motivated attacker may find unique ways to exploit these undocumented behaviors to leak information and/or circumvent policy. Furthermore, such "specification updates" are not at all uncommon. The same core i7-900 documentation states 152 separate issues, including 6 issues that cause "unpredictable behavior" and 11 capable of spawning "unexpected results". Assuming conservatively that this represents *all* of the issues, and that the issues are spread evenly over the 1 year lifetime during which the core is produced, this is still nearly 2 issues per week. Systematic ways for detecting these potential security vulnerabilities are necessary for more reliable and secure hardware.

Another example of a bug in an embedded system costing millions of dollars is in the Toyota unintended acceleration issue [100]. In this example, the accelator in some Camry's activated against user control and they were unable to control their automobile. Although these were unlikely to be caused by malicious intent, the security and saftely assessment of these systems was lacking. For example, the "embedded systems experts who reviewed Toyota's electronic throttle source code testified that they found it defective. They said it contains bugs – including some that can cause unintended acceleration." [100]. Toyota was forced to pay $1.5M to the driver of the car and $1.5M to a victim's family in a resulting crash. The jury in this case commented that "Toyota acted with 'reckless disregard' for the rights of others." This type of mishap not only costed Toyota $3M to the plaintiffs in this case, but unmeasurable amounts in terms of PR and related market effects. Toyota may have been able to circumvent some of these issues if they had more systematic security testing to ensure that these sort of safety critical systems would not malfunction inadvertently.

Some methods for security assessment do exist but are typically done by mandated security standards set forth by various standards bodies. Some companies have their own internal security auditing teams, such as Intel's Security Center of Excellence (SeCoE), but many others follow more gonvernment mandated standards. These standards provide a safety assessment mechanisms to help remedy some of these issues in the high-assurance markets. For example, there is the FIPS 140-2 standard for cryptographic devices [81] put on by the National Institute of Standards and Technology (NIST) to provide a standard assessment mechanism for designing cryptographic modules. Much emphasis is put on tamper-resistance and TEMPEST attacks (similar to side-channel attacks) and a level from 1 to 4 is assigned based on how well the module is tested. Other standards exist to measure how well systems have been evaluated. For example, the Common Criteria [1] standard specifies a set of rules for testing and evaluation. Specifically, an

Evaluation Assurance Level (EAL) is awarded to systems based on how thoroughly they have been evaluated (assigned a number from 1 to 7). Not surprisingly, achieving a high-assurance level is not only time consuming but extremely expensive. Such assurance often requires detailed verification including strict theorem proving and third party analysis [1]. This complicated process not only takes a tremendous amount of time estimated at 10 years [97] but also costs thousands of dollars per line of code [6]. Reducing this overhead is needed to keep these operation critical systems up-to-date with current technology at a reasonable cost. Tools for making the security assessment of hardware is greatly needed to both reduce this evaluation cost and make security evaluation a primary constraint in the hardware design flow.

A powerful new technology presented in past work [103] and more thoroughly evaluated in this thesis, called gate-level information flow tracking (GLIFT), provides a mechanism for helping with hardware security assessment. It provides a method for testing various security properties about the underlying hardware in a similar flavor to functional testing and verification. This thesis is primarily focused on how GLIFT can be used to specifically detect timing channels. However, in the rest of this chapter, I will first disuss some necessary background about information flow tracking, GLIFT, and timing channels.

2.2 Information Flow Tracking and GLIFT

A large amount of previous work has been done in the area of information flow security for complete systems. Numerous works have been done on information flow tracking specifically in hardware because monitoring information flows at this level allows for unintended flows to be identified without significantly affecting system performance. This section discusses the previous research in information flow security for complete sytems and specifically focuses on GLIFT since it is the primary technique we used in

previous and continuing analyses.

Information flow security focuses on monitoring the movement of data among different trust levels. Traditionally information flow security is guaranteed by ensuring that a particular information flow policy, such as integrity or confidentiality, is upheld. These policies can be modeled using a lattice $(L, \sqsubseteq)$ [30], where L is the set of security labels and $\sqsubseteq$ is a partial order between these security labels that specifies the permittable information flows. For example, consider the example lattices shown in Figure 2.1. Figure 2.1 (a) shows a typical binary security lattice. For such a binary lattice, the term *taint* is often used for the higher label on the lattice. For integrity, untrusted information is considered *tainted* in order to monitor if this taint violates a trusted (untainted) location. For confidentiality, taint is defined differently. The policy taints secret information to verify whether this leaks to an unclassified domain. Figure 2.1 (b) shows a security lattice with multiple trust levels. In both cases, integrity states that information may flow down in the lattice, but not up ("no write-up, no read-down" [19]). Confidentiality states that information can flow up the lattice but not down ("no read-up, no write-down" [17]). For simplicity, our analysis focuses on the binary lattice `low` $\sqsubset$ `high` in Figure 2.1 (a).

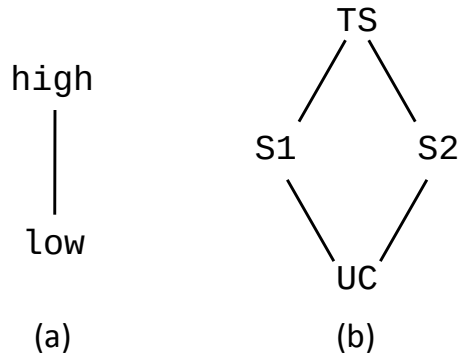

Figure 2.1. Examples of different security lattices. (a) is a typical binary security lattice showing `low` $\sqsubset$ `high`. (b) is a more complicated lattice with multiple labels. Here TS is top-secret, S1 and S2 are two security levels which are less secure than TS but more private than UC (unclassified).

To be concrete, consider the code snippet shown in Figure 2.2 and let $L(\mathtt{w})$ be the

label assigned to some variable w. First, to understand explicit information flows and how they can be tracked, consider the assignment shown $y = z$. For integrity, using the lattice $\texttt{low} \sqsubset \texttt{high}$, this code is secure if $L(y) \sqsubseteq L(z)$. In other words, this code is information flow secure only if the label assigned to z allows for information to flow into y without violating the integrity of y. Integrity in this situation is not violated as long as both y and z have the same label or y is low and z is high. Confidentiality does essentially the opposite as integrity. Using the binary lattice, the assignment is information flow secure if $L(z) \sqsubseteq L(y)$. Meaning that information can only flow into y from z if y is at higher or equal security level to that of z. Implicit flows in this example follow a similar strategy except that they flow information indirectly to the variable. This particular code shows an implicit channel in the form of a branch. Here x leaks information to y because, depending on x, y will be assigned the value of z. For both confidentiality and integrity, implicit flows need to also be eliminated. In this particular example, to enforce integrity, the labels must adhere to $L(y) \sqsubseteq L(x)$ in a similar manner as the explicit flow. For confidentiality, a similar relation holds for the explicit flow, namely $L(x) \sqsubseteq L(y)$. Note that if integrity or confidentiality is to hold for the entire code, both the information flow constraints for the explicit *and* implicit flows must be enforced. Using this common model of information flow security, many implementations have been made to enforce this at all layers of the system design.

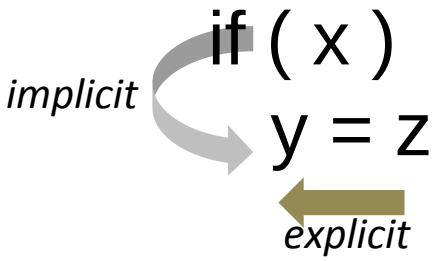

Figure 2.2. Simple code snippet showing explicit information flow from z to y and implicit information flow from x to y.

The most common techniques for information flow security are implemented in programming languages using type based systems and in operating systems. Sabelfield and Myers [94] present a survey on the different programming language based techniques. Most work has been done in static compile based techniques which build off of the typing system of a language in order to enforce information flow security. These methods have shown to be effective and can even eliminate implicit channels due to conditional branches in execution. Jif [74, 72] is a good example of such a type based system. Similar programming language based techniques have been applied to JavaScript to help alleviate security issues such as history sniffing or executing malicious code [51, 24]. Flume [58] has been shown to enforce information flow security using abstractions for operating system primitives such as processes, pipes, and the file system. It works by providing a user-level reference monitor which allows for decentralized information flow control (DIFC) [73]. Some other operating system DIFC mechanisms [31, 114] perform similar techniques but are incorporated into an entirely new operating system. These schemes are often effective but are forced to abstract away the potential implicit flows that occur in hardware. Further, these schemes force the designer to comply with a new typing system (in programming language techniques) or reduce the overall system performance (in operating system abstractions).

To maintain system performance, information flow tracking has been proposed in hardware. The most common hardware information flow tracking strategies focus on the Instruction Set Architecture (ISA) and microarchitecture. One such technique called Dynamic information flow tracking (DIFT), proposed by Suh et al. [98], tags information from untrusted channels such as network interfaces and tracks it throughout a processor. They label certain inputs to the processor as “spurious” (tainted) and check whether or not this input causes a branch to potentially untrusted code. This technique has been shown to successfully prevent buffer-overflow [26] and format string attacks [76]. Raksha [28]

is a DIFT style processor that allows the security policies to be reconfigured. Minos [27] uses information flow tracking to dynamically monitor the propagation of integrity bits to ensure that potentially harmful branches in execution are prevented in a manner similar to Suh el al. [98]. RIFLE [104] is an information flow secure system which uses binary re-writing. Since security annotations are done on the binary itself, applications can be executed on their system without requiring additional security type annotations to programs.

These previous techniques are effective at ensuring that potentially harmful branches in control flow are prevented or guaranteeing the integrity of critical memory regions. However, these methods target a higher level of abstraction and cannot be used to monitor the information flows in general digital hardware. For this reason, these methods also fail to detect hardware specific side channels in the form of timing. GLIFT provides a solution for tracking information flows, including those through timing channels, in general digital hardware. GLIFT works by tracking each individual bit in a system as they propagate through Boolean gates. This is done using an additional tag bit commonly referred to as *taint* and tracking logic which specifies how taint propagates. Information is said to flow through a logic gate if particular *tainted* inputs have a chance to *affect* the output.

Taint is a label associated with each data bit in the system which indicates whether or not this particular data bit should be tracked. If integrity is a concern, untrusted information is tainted to ensure that this tainted information does not flow to a trusted location. In the case of confidentiality, secret information is tainted to monitor whether it leaks to a public domain. Taint is propagated whenever a particular tainted data bit can affect the output. In other words, if the output of a function is dependent on changes to tainted inputs, then the output is marked as tainted.

For example, consider a simple 2-input NAND gate as seen in Figure 2.3 (a) and

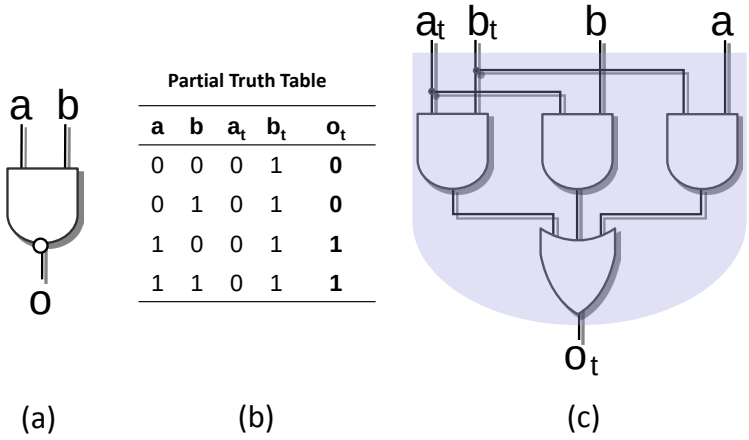

Figure 2.3. (a) A two-input NAND gate. (b) Truth table of two-input NAND gate with taint information (not all the combinations are shown). (c) The corresponding tracking logic of two-input NAND gate is $ab_t + ba_t + a_t b_t$. Every change at the input of the gate is precisely tracked at the output.

its corresponding tracking logic as shown in Figure 2.3 (c). For a NAND gate, only particular input changes will result in a change at the output. Specifically, consider the case in which $a = 0$ and $b = 1$. Here changing the value of b will cause no change at o since $a = 0$, meaning that there is no information flowing from b to o. If b were to be tainted ($b_t = 1$) and a untainted ($a_t = 0$) in this case, o would be untainted ($o_t = 0$) since the tainted input does not affect the output. A subset of all such combinations can be seen in Figure 2.3 (b). Using the full truth table, a function can be derived for all similar input combinations into a tracking logic function as shown in Figure 2.3 (c). Since NAND is functionally complete, the tracking logic for any digital circuit can be derived by constructively generating the tracking logic for each gate. In other words, given a circuit represented as NAND gates, the circuit can have complete information flow tracking by interconnecting the tracking logic for each individual NAND gate. This results in a design that precisely tracks the information flow of each individual bit. As mentioned, GLIFT is a useful tool for analyzing any digital hardware because it exposes all information flows explicitly.

All prior work in the area has assumed that if a function is computed the output should be tagged as tainted if *any* of the inputs are tainted. This assumption is certainly sound (i.e., it should never lead to a case wherein the output which should be tainted is marked as untainted) but it is overly conservative in many important cases, in particular if something is known about the inputs at runtime. To understand the importance of precisely tracking information flow, consider a simple 1 bit counter that increments (or toggles in this case) every cycle or gets cleared back to zero from a reset. If a counter is implemented as depicted in Figure 2.4 a), and uses a conservative information flow tracking scheme as mentioned, there is no way for the counter to ever return to an untainted state once it has been marked tainted since the value of the counter will be tainted. Yet it is obvious that the system should return to an untainted state on an untainted reset. However, if GLIFT is used as described, the counter would in fact reset back to an untainted state as seen in Figure 2.4 b) and c). This is because GLIFT considers the values of the inputs to determine if it is possible for information to flow from the inputs to the outputs.

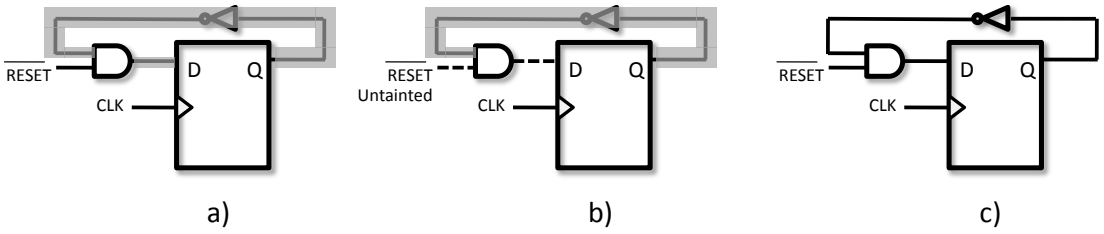

Figure 2.4. A 1-bit counter with reset. Our tracking logic is more precise and recognizes that an untainted reset guarantees an untainted 0 in the counter value. The gray area indicates the value here is tainted. a) Tainted counter. b) An untainted reset puts the counter back to an untainted state. c) Untainted counter

Now that the details behind how GLIFT works has been presented, it helps to see how it can be used in practice for testing for information flows.

2.2.1 Using GLIFT as a Static Testing Technique

Our GLIFT analysis can be accomplished *statically* (at system design time) meaning that additional gates due to our analysis logic are never actually fabricated and are used only at design time. Most of our customers have expressed interest in using our gate-level technology for static analysis so we focus on how our technology can be used in this manner.

In a static scenario, the GLIFT analysis is used to test or verify if the system complies with designer supplied security policies. This is done completely at design time, i.e., there is no need to physically instantiate the analysis logic after testing or verification is complete. In the past, we have successfully employed this in both testing [78, 53] and verification [102] using GLIFT.

Figure 2.5 provides a possible design flow for static analysis. The system is described in a hardware description language (HDL) such as *VHDL* or *Verilog* [49]. Once the digital circuit under design has passed functional verification, it is synthesized into a gate-level netlist using standard hardware synthesis tools such as Synopsys Design Compiler. Next, each gate in the netlist is augmented with the appropriate analysis logic. There are many methods for generating this tracking logic for a given circuit all with varying computational complexity and trade-offs with preciseness. In other words, GLIFT can be imprecise since sometimes it will conservatively say there is an information flow when there is in fact not one. We have proven that generating precise GLIFT logic for a general function is *NP*-complete in past work. This thesis will not explore these topics further, but the interested reader should see our work on preciseness [46] and on the complexity of the various different GLIFT logic generation methods [47].

The most common method used to generate GLIFT logic is a constructive one, which creates GLIFT logic for each logic gate in the gate-level library individually and

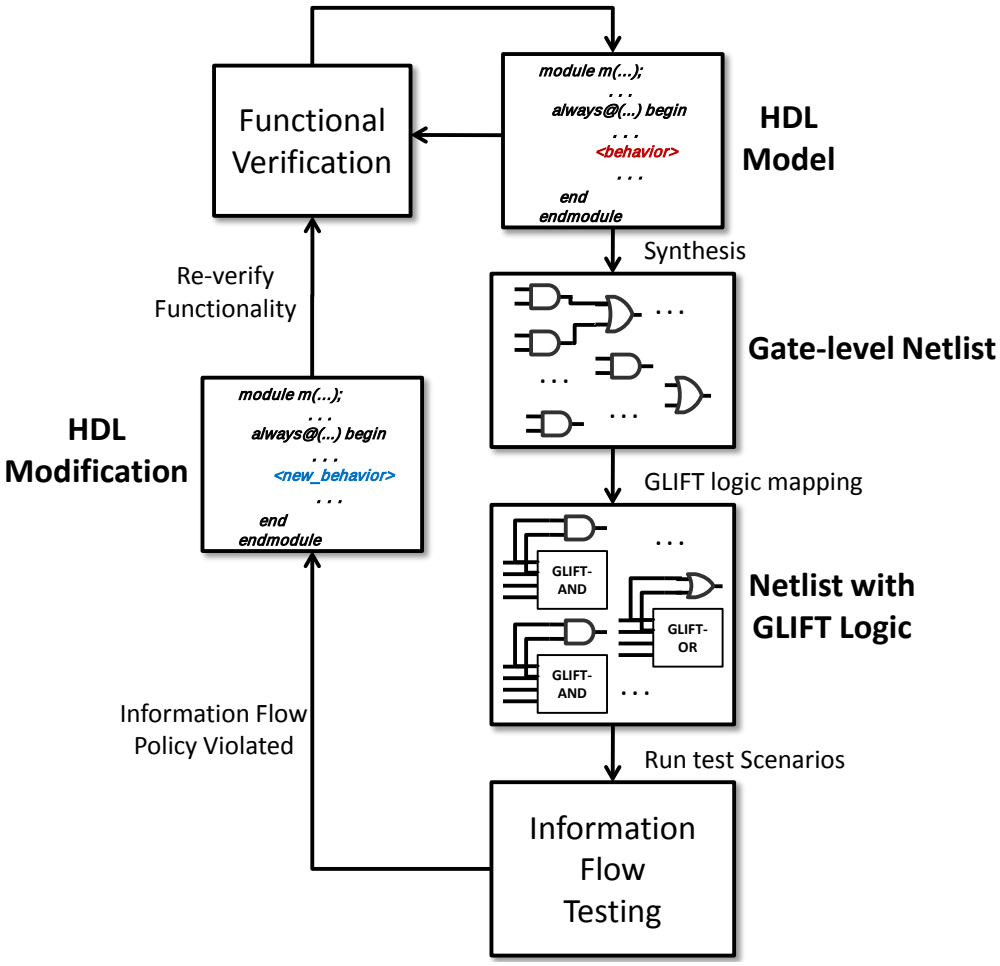

Figure 2.5. Using gate level analysis logic for static assesment of security properties in digital system design.

then appends these new "GLIFT" gates together. In other words, every gate in the system has logic attached to it individually in a linear fashion. This design equipped with GLIFT logic is now capable of being analyzed for unintended information flows. At this stage, test vectors are run on the hardware to see if it violates the information flow policy. If the policy is violated, the description of the hardware is modified and once again undergoes functional verification and proceeds through the testing flow again. Making modifications to the hardware model is not obvious and generally requires a form of time-multiplexing to prove the absence of timing channels from the design. This has shown to be effective when analyzing processor cores and bus controllers as subsequent sections discuss in more detail.

Once the appropriate GLIFT logic is added for the design, testing and/or verification scenarios are run to check if the design potentially violates any pre-defined security policies (e.g. "does my secret key leak?"). If some policy is violated, the design is modified and re-verified until all properties of interest are pass. In the static testing or verification application scenario, the tester is responsible for specifying what properties to analyze. For example, the tester may wish to ensure that certain critical portions of the system are never affected by untrusted components or that a secret encryption key is never flowing to somewhere other than the ciphertext.

In our past work we have demonstrated the use of gate property analysis in secure hardware design as a static analysis technique. We slightly modified the I2C and USB protocols to eliminate timing channels and utilized the testing methodology to show that these new protocols adhered to strict timing behaviors [78]. This is important in many scenarios, e.g., to ensure that real-time systems do not miss their deadlines. Other recent work addresses trust issues in hardware designs that use mix-trusted IP cores from different vendors [53]. Here we use gate-level analysis to verify that there is no unintended interaction between modules that may violate security properties such as

non-interference. In [102], a static analysis technique is used to verify the concrete hardware implementation with partial software specification to be free of unintended information flows. The information flow properties of the entire design are statically verified using gate-level analysis. To present a better understanding of how this method can be used in practice, I will describe an example use case on an AES core.

2.2.2 Example: Using GLIFT on AES

To understand how one might use this testing flow and how information flows at the gate-level, it is useful to think about concrete example. Let us consider a hardware encryption core. For the purposes of this discussion, I will use the advanced encryption standard (AES). I choose to use an encryption core as an example since they are very commonly built in hardware and their security is often at the core of the security of the entire system.

Figure 2.6a shows the overview of AES as a hardware encryption core. It is comprised of several different submodules which each perform a specific function such as *MixColumns*, *SubBytes*, etc. AES itself executes these different functions in many iterations (called rounds) using a portion of the expanded *Key* in each round to produce a ciphertext. AES itself has stood the test of time against cryptanalysis attacks and is accepted to be secure. In other words, the cryptographic community has accepted it to be mathematically infeasible to extract either the key or message from the ciphtertext. When this algorithm is built in hardware, however, there are various different control outputs that are required for ease of integration into the system. These outputs could be things such as a way for a designer to read out the encryption key when in a special debug mode or something as simple as saying when the ciphertext is ready.

In certain circumstances, flows from the secret key to these control outputs might be acceptable and in others could be disastrous to the security of the system. For example,

it might be acceptable to read the key out only when in a special "debug" mode. In addition, it might be unacceptable for the key to flow to a "ready" control output since this could be a potentially exploitable timing channel (i.e. the key has the ability to affect the execution time of the core).

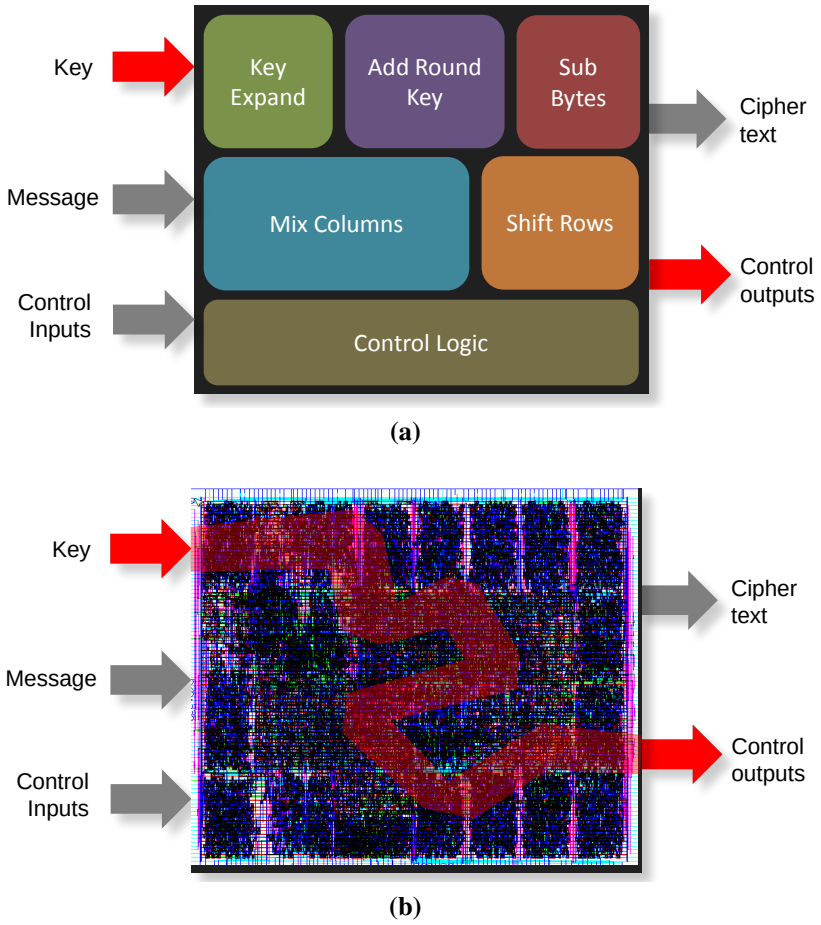

Figure 2.6. (a) An overview of the interface and internal modules of a hardware AES encryption block. (b) The layout for an AES block. Proving that the Key does not flow to a control output is non-trivial.

Unfortunately verifying a property like this is extremely non-trival. Figure 2.6b shows the layout of an implementation of AES. An intuitive way of figuring out whether or not the key flows to a control output becomes extremely difficult in the mess of thousands of gates. However, by using our gate-level technology, we can provide designers with

an easy tool to mark their key as "secret" and discover if it is actually ever leaking to an "unclassified" control output. In doing so, hardware designers can systematically evaluate the security of their design by enumerating the different properties they would like to check.

GLIFT in and of itself does not ensure that secret data does not spread across the whole machine, rather it is a static analysis technique that ensures that as secret data spreads across the machine it will always be properly detected. Tracking properties at the gate level at design time makes the most sense when coupled with the hardware and software designs that attempt to *enforce* that the desired properties hold.

2.3 Timing-Channel Attacks

Timing-channels are a form of side channel in which secret information leaks through the amount of time a computation takes to execute. Other forms of side channels include physical ones such as the amount of power a cryptograhic operation consumes [54, 66, 70, 71, 85], the amount of electromagnetic radiation it produces [13, 65], the sounds keys make on a computer keyboard [15], the difference in faulty behavior when a cryptographic hardware module ages [20, 21, 87], and even the minute sounds computers make (likely from the fluctuations in capacitors) when they perform a computation [36]. All of these side channels have been shown to leak secret encryption keys and other critical pieces of data. Although physical side channels are quite powerful in nature, this section focuses primarily on timing based side channels. A timing based channel when secret data affects how long a computation takes the execute and thus the secret information can be inferred from time. In addition, on the availability side, a timing-channel can be used by an attacker to provide variants of denial of service by affecting *when* a critical system component can respond. Timing-channels have seen a lot of attention in recent years with many demonstratable attacks performed on the underlying hardware. These

attacks typically exploit hardware specific features that are very frequently impossible to solve using software mechanisms without destroying system performance.

There have been many published confidentiality-based timing attacks that target cryptographic algorithms such as the advanced encryption standard (AES), the Rivest-Shamir-Adlemann (RSA) public-key crypto system, Diffie-Hellman, and others. These timing attacks exploit hardware performance mechanisms such as caches and branch-predictors. In the processor cache space many attacks of been performed in various fashions on data-caches [22, 10, 86, 18, 40] and even instruction caches in the work done by Aciiçmez et al. [12, 9]. All of these cache-based attacks rely on the non-deterministic latencies of performing a memory read. For example, cryptographic algorithms will fill up the cache with data and often times the cache line in which data is mapped leaks some critical information. As been demonstrated in this past work, an attacker with the ability to measure time can extract complete encryption keys solely from the variations in time from caches. Even further, Acciiçmez et al. also performed timing-based attacks on branch predictors [11] where they exploited the varying latencies of correctly and incorrectly predicting whether or not a program will take a branch. There has also been much recent research to help thwart these attacks by creating new cache designs, as done by Wang et al. [107, 108], and modifying the way in which attackers can measure time, as done by Martin et al. [67].

To give a simplified example of how a timing channel attack works, I will summarize a simplified version of the attack performed by Paul Kocher in his timing channel attack paper on RSA [55]. The Rivest-Shamir-Adleman (RSA) assymetric, public-key cryptographic system is one of the most widely used systems in the SSL/TLS standard for both symmetric key-exchange and digital-signatures. I will only briefly discuss the portions of the system that have been vulnerable to timing attacks so that the exploitability of this algorithm can be thoroughly understood. For details about the entire

algorithm, the reader should refer to the original RSA paper [92].

The security of RSA relies on the difficulting of computing discrete logarithms. Stated differently, data is encrypted and decrypted by performing modular exponentiation, which can be done efficiently. However, given a cipher text, it is computationally difficult to compute the inverse (discrete log) to extract the message or key. To be concrete, a message M is encrypted using a public key e and the ciphertext C is decrypted using a private key d as follows:

$$\begin{aligned} C &\equiv M^e \pmod{n} \\ M &\equiv C^d \pmod{n} \end{aligned} \tag{2.1}$$

Where e, d, and n are derived during the key-generation process (not discussed here) using two large and distinct prime numbers. Given that an attacker has access to M, ciphertext C, public key e, and modulus n, it is accepted to be computationally infeasible to compute the discret logarithm $\log_C(M) \pmod{n}$ to extract the secret key d.

Now, to perform modular exponentiation to take C to the d, one could naively perform: $\underbrace{C \times C \times ... \times C}_{d}$. This would be very inefficient since there are certain cases where a multiply is not necessary depending on the value of the exponent. To optimize this, a square-and-multiply algorithm is often used. One implementation of this algorithm can be seen in Algorithm 1.

As stated, this optimization is performed to reduce the number of multiplies the system needs to perform because they typically require more *time*. Since the decision to perform a multiply is made on the value of a key-bit, an attacker with the ability to measure the execution time can, with surprisingly high probability, extract the entire encryption key with a sufficient number of timing measurements (an interested reader

Algorithm 1. Basic algorithm for square-and-multiply to compute modular exponentiation. It computes $C^d \pmod{n}$.

```
R = 1;
temp = C;
for i = 0 to |d| − 1 do
  if bit d[i] = 1 then
    R = R · temp (mod n)
  end if
  temp = temp^2 (mod n)
end for
return R
```

should refer to Paul Kocher's paper [55] for specific details of the attack).

These timing-based attacks are not new and have been discussed early days of system literature (most notably by Wei Hu and other works related to the VAX Virtual Machine Monitor [52, 44, 111]). The most common solution proposed to solve this problem was to add randomness to the system to make the timing measurements more "noisy." Thus attemping to make the signal-to-noise ratio high enough that it makes stochastically difficult for an attacker to find any useful information. TimeWarp by Martin et al. [67] is a good recent example of this type of technique.

There has also been much work that is typically based on information theory to quantify how much information is leaking through a timing channel. Askarov et al. [14] present a software-based approach that black-boxes modules and adds randomness to when output events of this black box can be observed. They use information theory to bound the amount of information leaking through the timing channel. Giles et al. present "jammers" for timing channels that add randomness and then quantify the difficulty of exploting the channel using information theory. Kocher [55] presents cryptographic blinding in order to prevent timing channels. Cryptographic blinding is a way of adding randomness directly to the computation rather than adding randomness into the system itself. For example, for computing the modular exponentiation in RSA, one can multiply

the input message by a random number and then reverse the operation before decrypting. An attacker measuring time in this case would have a much more difficult time extracting the secrte information because it would be blinded by this random number. In a similar manner, Köpf et al. [56] analyze how much information is leaked once cryptographic blinding is applied. Their approach is also information theoretic.

The root cause of these timing channels are typically hardware specific. For example, the *time* penalty for missing a cache is higher than hitting it, the latency for a mis-predicted branch is higher than one that is correct, or more time is spent on certain branches of computation. In general, a multi-process system will take longer to finish its work when processes contend for hardware resources. All of these hardware-specific timing vulnerabilities can, and have been exploited to extract secret information. Further, hardware-based timing channels can even violate the availability where a process or subsystem cannot meet its deadline because a less trusted component has violated its response time.

Detecting these sort of vulnerabilities, in addition to other hardware security bugs, is becoming an increasingly difficult and costly problem. Currently, there are limited (if any) methods for assisting designers in finding potential problems in their hardware. There are many methods for helping prevent these sort of attacks, but systematic methods for *detecting* them is an extremely difficult problem. As mentioned, there are many other examples in which the lack of security assessment have costed companies money and time. If these companies had a mechanism for testing the security properties of their underlying hardware, many of the bugs that are found are likely to be circumvented. This thesis provides such a tool and demonstrates how it can be used against hardware timing channels. The remainder of this thesis will explore how gate-level information flow tracking (GLIFT) can be used to detect and identify hardware timing channels. The next Chapter will first provide some essential definitions for GLIFT in order to build a

solid theoretical foundation.

Chapter 3

Fundamental GLIFT Logic Equations and Overheads

In order to have a good understand of the details behind GLIFT, it is helpful to have an understanding of some essential definitions and associated overheads. In this chapter I will present some of these definitions and give some discussions into the overheads that the logic creates.

GLIFT logic is the co-existing logic that propagates information throughout a system. This logic performs no operations that affect the way that the system operates. The GLIFT logic is used to understand where information is propagating to point out where potential security vulnerabilities are. This is done by "tainting" information and monitoring it as it flows throughout the system. Before we can discuss the necessary formal equations for GLIFT, I will first define taint. Throughout this discussion, the shadow logic for basic gate primitives (AND, OR, NAND, etc.) will be discussed.

3.1 Taint and Tracking (GLIFT) Logic

Taint is simply a label associated with a piece of information that renders it to be tracked. For example, if one wishes to track the movement of a secret key in a piece of hardware, then they would taint the key to indicate that it is secret. In GLIFT logic, as

we will soon discuss, every bit has an associated taint bit (for a 2-level security lattice). To this end, if a bit's associated taint bit is asserted, we say that that bit is tainted. To be more precise, a more formal definition is given in Definition 1. I will later repeat this definition for completeness in Chapter 6.

Definition 1 (Taint)**.** *For a set of wires (inputs, outputs, or internals) X, the corresponding* taint set *is X_t. A wire x_i for $x = (x_1, \ldots, x_i, \ldots, x_n) \in X$ is* tainted *by setting $x_{it} = 1$ for $x_t \in X_t$ and $x_t = (x_{1t}, \ldots x_{it}, \ldots x_{nt})$.*

Now that taint has been introduced, I can now present some formal definitions of GLIFT for some common logic gates and some associated proofs. These definitions are with respect to a combinational (stateless) logic function. Later on in Chapter 6, these definitions will be expanded to include the notion of time and state. First I will define the tracking (GLIFT) logic for a combinational logic function.

Definition 2 (Tracking logic)**.** *For a combinational logic function $f : X \rightarrow Y$, the respective* tracking logic function *is $f_t : X_t \times X \rightarrow Y_t$, where X_t is the taint set of X and Y_t the taint set of Y. If $f(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$, then $f_t(x_1, \ldots, x_n, x_{1t}, \ldots, x_{nt}) = (y_{1t}, \ldots, y_{mt})$, where $y_{it} = 1$ indicates that some tainted input x_j (i.e., an input x_j such that $x_{jt} = 1$) can affect the value of y_i.*

From this definition, it can be seen that the tracking logic will assert logic 1 whenever a tainted input can affect the output of the original logic function. With this definition in place, I will now present some basic logic functions for some gate primitives and later present the overheads associated with GLIFT logic on larger benchmarks.

3.2 GLIFT Logic for Basic Boolean Gates

There are many ways in which to derivce the GLIFT logic for a logic function. The easiest method is to derive logic for gate primitives and then replace the gates in the

combinational logic function with the GLIFT logic for each primitive. This method was discussed prior in Chapter 2. There are other methods to generate this logic as well, the interested reader should consult our paper on generating GLIFT logic [47].

The purposes of this section are to present and discuss the intuition behind the tracking logic for several gate primitives. First, I will discuss the logic AND.

Given logic AND, $f = g \cdot h$, its associated tracking logic f_t is

$$f_t = g \cdot h_t + g_t \cdot h + g_t \cdot h_t \tag{3.1}$$

where $\cdot$ and $+$ represent the logical AND and OR operators respectively. To break down the intuition here, the output of f can be affected in several ways by a tainted input as each term in f_t conveys. The first term, $g \cdot h_t$, shows that f can be affected when $g = 1$ and h is tainted. In this case, changing the value of h will cause a change at the output of f, this f_t should be 1 on this case. A similar case is made for the term $g_t \cdot h$. The last term, $g_t \cdot h_t$ covers the case when both inputs are tainted.

Now, lets look at the tracking logic for a logical OR ($f = g + h$):

$$f_t = \bar{g} \cdot h_t + g_t \cdot \bar{h} + g_t \cdot h_t \tag{3.2}$$

The structure of this equation is the same as that of logic AND in that there are 3 terms. However, in the OR case, each input is negated. To understand why, consider the first term: $\bar{g} \cdot h_t$. Here, the output of f will be affected by the tainted input h only when $g = 0$. Only if $g = 0$, does h have the ability to influence the output.

The last gate-example I would like to show is that for an XOR ($f = g \oplus h$):

$$f_t = g_t + h_t \tag{3.3}$$

The intuition here is that, for an XOR, any input changing will cause a change at the output. As an example, if g is tainted ($g_t = 1$), then the output of the GLIFT logic (f_t) will be 1. This is because a change of a tainted input (g) is able to influence the output of f.

This analysis can be broadened to derive the tracking logic for more complex functions by simplying analyzing whether or not a tainted input can influence the output. In general, we have proven that derive completely precise GLIFT logic is *NP*-complete [47]. The best method to create this logic is to generate the logic for some complex gates, as we have done simply here, and then those gates can be swapped out with the GLIFT logic as needed. This reduces the amount of overhead required when generating the GLIFT logic for a new (and potentially complex) design. I encourage the interested reader to consult some of our other work on this topic [46, 47, 48, 45] but most of it is out of the scope of this thesis.

3.3 GLIFT Logic Overheads

To understand how much the GLIFT logic causes in terms of overheads, it helps to look at a set of standard benchmarks. First, I will present some results in terms of number of minterms, which are the number of terms in the logic function which make the output 1. I first use this metric because it provides a sense of the logic complexity independent of optimizations. Afterward, I will present overheads in terms of area for a set of IWLS and ISCAS benchmarks. It should be explicitly clear that the techniques presented in this thesis use GLIFT primarily for testing. That is, GLIFT logic is never physically instantiated on the design. The overheads presented here are to give a sense of how the GLIFT logic will affect the time spent performing testing/verification.

Figure 3.1 shows the number of minterms for several input AND and XOR gates. It should be noted here that the number of minterms for an AND-gate, OR-gate, NAND-

gate, and NOR-gate are all the same. Intuitively, this can be seen in Equations 3.1 and 3.2. Further, since the input of an inverter always affects the output, the GLIFT logic of an inverter is essentially a wire. Thus the GLIFT logic for a NAND gate is the same as an AND. The same rule applies to an OR gate. Figure 3.1 thus demonstrates that as as the number of inputs grows linearly, the number of minterms grows exponentially. The bounds 2^N and 2^{2N} represent the total number of minterms for the orginal logic function (2^N) and the GLIFT logic function (2^{2N}).

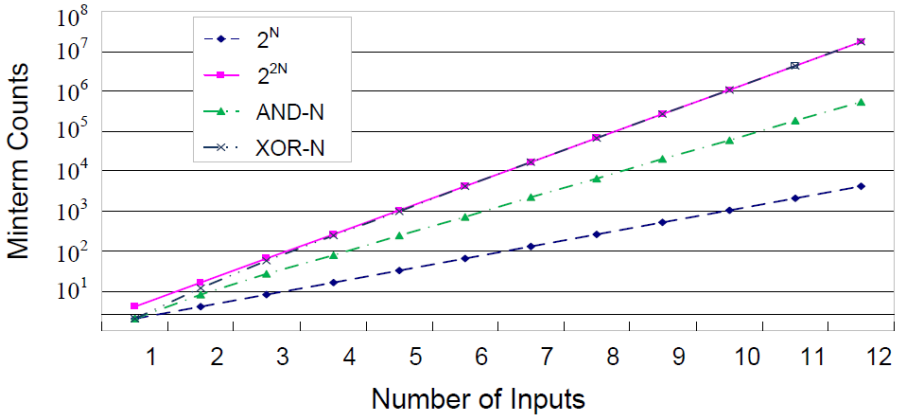

Figure 3.1. The number of minterms in the GLIFT logic function for AND and XOR. Minterms here are the number of terms in the GLIFT logic function. 2^N is the total possible number of minterms in the original function and 2^{2N} the number of total possible minterms in the GLIFT logic fucntion.

A similar behavior can be seen on some more standard benchmarks. In Figure 3.2, we can observe an exponential increase in the number of minterms for several adders, multipliers, comparators, multipliers, and multiplexors. This does not, however, imply that the actual optimized area will be exponentially larger than the original design. This just provides a sense of how complicated the GLIFT logic is with respect to the original logic.

To understand the concrete area overheads, we can examine Table 3.1. Here we analyze several IWLS and ISCAS benchmarks [3]. On average, for this set of benchmarks, the GLIFT logic is 3.25 times larger than the original logic. The results presented in this

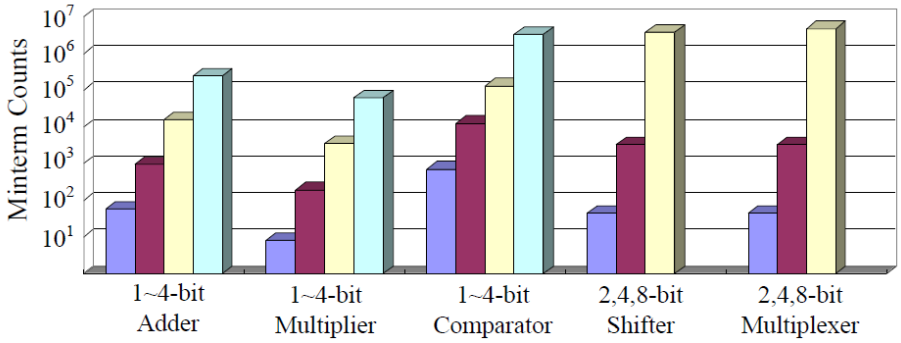

Figure 3.2. The number of minterms in the GLIFT logic function for several benchmarks. 1 to 4 bit Adders, Multipliers, and Comparators and 2, 4, and 8 bit Shift and Multiplexors.

table are from the Berkeley logic synthesis and verification tool, ABC [7], and are thus unitless. As the designs increase, however, there does not seem to be a trend of increased area. Intuitively, this makes sense because the constructive method provides a fixed overhead on each gate (for 2-input gates this is around 4X). If another logic generation method was used, as discussed in some of our other work [47, 46], these area numbers increase.

Table 3.1. Area of combinational logic benchmarks (ISCAS and IWLS) and overheads from the GLIFT logic using the constructive generation method discussed earlier in this thesis. Area numbers are unit-less and were produced using Berkeley's ABC tool [7]. The **average area increase is: 3.25X**

Benchmark	Original	Constructive	Size Increase
74283	109	179	1.64X
alu4_cl	671	2292	3.42X
s344/s349	488	1765	3.62X
C5315	2501	9166	3.66X
C7552	2897	9377	3.24X
i10	3583	14407	4.02X
C6288	4994	11965	2.40X
too_large	7812	15957	2.04X
des	14539	75262	5.18X

Although there is a 3.25X size increase in terms of area, the GLIFT logic itself is

never used in a physical design. It can be thought of as a logic which provides meta-data about information flow. It is used soley at design time to analyze the flow of information through the design and, once this analysis is complete, the logic will be remove prior to fabrication. In addition, GLIFT's structure provides some very powerful properties. For one, it can detect hardware timing channels (one primary purpose of this thesis). Another, yet more subtle property is that it can be used for formal verification. Since the GLIFT logic itself is a logic, it can be used to formally verify different security properties. Although this topic is not explored in this thesis, I will address potential future research directions at the end of this document.

This chapter, in small part, is a reprint of the material as it appears in the Design Automation Conference 2010. Oberg, Jason; Hu, Wei; Irturk, Ali; Tiwari, Mohit; Sherwood, Timothy; Kastner, Ryan; The dissertation author was the primary investigator and author of this paper.

Chapter 4

Exploring Information Leaks in Bus Protocols

Modern embedded systems rely on different modules communicating with one another. These different protocols should and need to be evaulated for security in our next generation automobiles, medical devices, mobile phones, and countless other applications. The analysis of bus protocols is important for both *integrity* and *confidentiality*. On the confidentiality side, potential flaws in the ways components communicate might improperly distribute keys or potentially leak other secret information through timing-channels. On the integrity side, misproper analysis of communication protocols can cause variants of denial of service and data corruption whether these were intended or not.

In this section, I will discuss how different protocols can be analyzed for security using GLIFT. Many of these are timing-related information leaks as will be explained throughout this chapter. This chapter specifically focuses on the Inter-Integrated Circuit I^2C and the Unniversal Serial Bus (USB) because of their wide adoption in embedded systems and their familiarity with many. The primary goal of this analysis is to show how the hardware security testing and verification techniques presented in this are invaluable assets for building stronger bus protocols.

4.1 Information Flows in I^{2}C

To start off, I will first begin to show how this analysis can be applied to the I^{2}C bus protocol. I use the information flow testing techniques presnted in Chapter 2 on I^{2}C to show how to obtain information flow isolation between devices on the bus.

I^{2}C is a 2-wire serial protocol consisting of a common clock and data line as shown in Figure 4.1 [5]. With that in mind, explicit information flows are quite simply identified since any device can openly snoop the bus even though transfer between the master and itself is never initiated. However, as this section will show, eliminating only explicit information flows in I^{2}C does not guarantee non-interference since information can flow through more difficult to detect side channels.

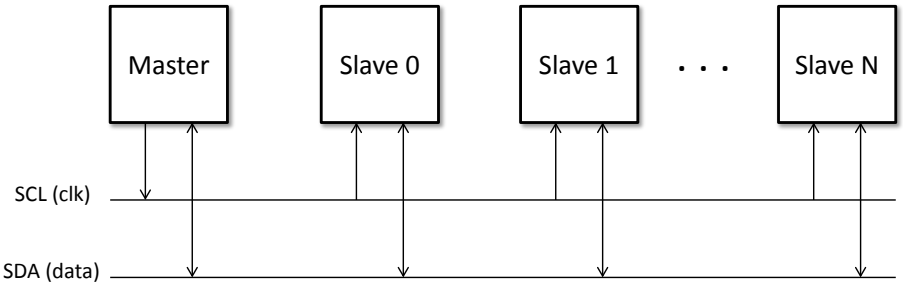

Figure 4.1. I^{2}C Bus configuration. I^{2}C can support several devices on a single global bus.

Our analysis of I^{2}C follows the same testing flow discussed in Chapter 2. We modeled I^{2}C devices as FSMs using RTL Verilog. It is not practical to enumerate all input combinations to this system since the number of states in which each device can be in grows exponentially. As a result, we chose to analyze the information flows for a common scenario in which a master writes data to a Slave 0 (as shown in Figure 4.1) and subsequently writes data to a Slave 1.

The design was synthesized using Synopsys Design Compiler and the gate-level functionality was also verified using Modelsim SE 6.6b [69]. Once verified, tracking

logic was associated with each gate and Slave 0 was marked as tainted because we wished to monitor where Slave 0's information flowed. During the communication between the Master and Slave 0, Slave 0 is required to send an ACK in response to receiving data. The Master's state depends explicitly on whether or not it receives an ACK. Since this ACK comes from tainted Slave 0, this ACK causes the Master's state machine to become tainted resulting in a taint explosion in the Master. As mentioned, such an explicit flow is expected since the Master and Slave 0 are directly communicating. However, once the Master subsequently communicates with Slave 1, this tainted information flows to Slave 1 resulting in a less obvious implicit information flow from Slave 0 to Slave 1. The tracking logic clearly identifies both information flows in this scenario.

In order to enforce non-interference between devices on the I^2C bus in this scenario, all information flows between slaves need to be eliminated. The following subsection discusses a useful technique for proving non-interference for this particular scenario.

4.1.1 Enforcing Non-interference in I^2C

To enforce non-interference between devices on the I^2C bus, we need to eliminate all information flows both explicit and implicit. This section discusses a solution for guaranteeing non-interference between devices on the I^2C bus.

As mentioned, there are obvious explicit information flows between devices since they are all connected via common wires. To eliminate explicit flows, we introduce an *adapter* which sits between the device and the bus as shown in Figure 4.2. This adapter arbitrates between the devices in a time division multiple access (TDMA) fashion such that only a single device is attached to the bus (in addition to the master) at any given time. In doing so, explicit information flows are eliminated since other devices are isolated from one another at all times. This does not completely eliminate all information

flows (as previously mentioned) since implicit information flows between devices via the master.

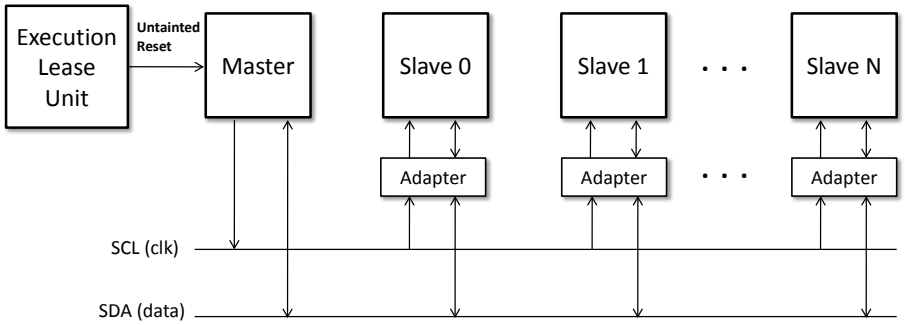

Figure 4.2. I^{2}C configured with an additional adapter to enforce TDMA. This enforces non-interference between devices under the presented test conditions.

To eliminate implicit timing information flows, we introduce an untainted *reset* for the master such that the master is restored to a known state prior to communication with another device. This execution lease unit monitors when a TDMA switch occurs and restores the master to an untainted state. By requiring a strict enforcement on bus access time and by restoring the master back to a known state, we eliminate any potential timing channels between devices. Since GLIFT also captures information flows elicited through timing channels, we are able to verify that these flows are in fact eliminated for this particular scenario.

4.1.2 I^{2}C Non-interference Overheads

The design was processed using the same aformentioned testing flow and synthesized with Synopsys Design Compiler. This particular scenario was tested for non-interference using Modelsim SE 6.6b [69]. We tested a round of communication for a single time slot and verified that the information flows were in fact contained. Since information flows were proven to be contained for a time slot, information flows will be contained for this test scenario in subsequent time slots.

Table 4.1. Time spent simulating our particular test scenario for both the original I^2C design and the one with GLIFT.

Design Type	**2 Slaves**	**4 Slaves**	**8 Slaves**
TDMA Gate Design	121ms	225ms	426ms
TDMA w/ GLIFT	192ms	389ms	770ms

To obtain simulation times, we execute the complete scenario mentioned and confirmed that the master returns back to a known state without leaking any information to any devices on the bus. The simulation times for this TDMA based solution and the design with GLIFT tested are shown in Table 4.1. As shown, we tested this scenario with 2, 4, and 8 slaves existing on the bus. Not suprisingly, the simulation times for both the original TDMA based solution and the one with GLIFT increased with the number of slaves present on the bus. Since we are scaling the number of slaves in the system by a factor of 2, this essentially doubles the hardware and resulting simulation time. Furthermore, the overhead of the GLIFT logic does have significant effect on the simulation time relatively speaking. However, this overhead is not unwieldy since the added simulation time can likely be tolerated for such a strong information flow guarantee.

In addition, we are required to have additional hardware (adapter) to eliminate explicit information flows between devices. Table 4.2 shows the sizes of each component in our testing scenario in terms of combinational and non-combinational area. It is important to note that our Master and Slave are of minimal functionality since we were only concerned with testing the previously discussed scenario. If additional complexity were added to the system (i.e., a fully functional I^2C system), the Adapter's area overhead would be much less significant since its functionality is fixed (i.e., it only performs arbitration). The overhead of the execution lease unit is insignificant and not shown since it is only needed to reset the master to a known state when its timer expires. Furthermore,

Table 4.2. Area for I^2C components in non-interference compliant design. This is the final system after testing and does not contain GLIFT logic.

Module	Gates	Flip-Flops
Master	145	26
Slave	125	24
Adapter	375	62

we are required to enforce a TDMA strategy which inherently reduces the bandwidth of the communication channel since an unused time-slot is wasted. However, this solution enforces non-interference proven by GLIFT in this particular scenario and such overheads could likely be tolerated for such a strong guarantee.

4.2 Information Flows in USB

Unlike I^2C, the Universal Serial Bus (USB) operates as a star tiered topology as shown in Figure 4.3. Devices are not sitting on one global bus in which explicit information may flow between one another. The Host node broadcasts data out to all Hubs and Devices. This downstream data (Host to Device) is observed by all devices and upstream data (Device to Host) is observed only by Hubs which are in the path of the stream [4]. As a result, devices are not able to snoop information sent from the Device to Host since information flows only through Hubs until it reaches the Host as shown in Figure 4.3. Devices can only potentially intercept information that is sent from the Host to Device since it is broadcasted. Thus the explicit information flows are less significant than in I^2C, assuming USB Hubs are properly routing information. However, timing channels are still very apparent in a similar manner as I^2C. This section discusses our analysis of the USB protocol along with a solution to enforce non-interference.

We are concerned with all information flows between devices. Although explicit information flows are less significant than in I^2C, they still occur from the host broad-

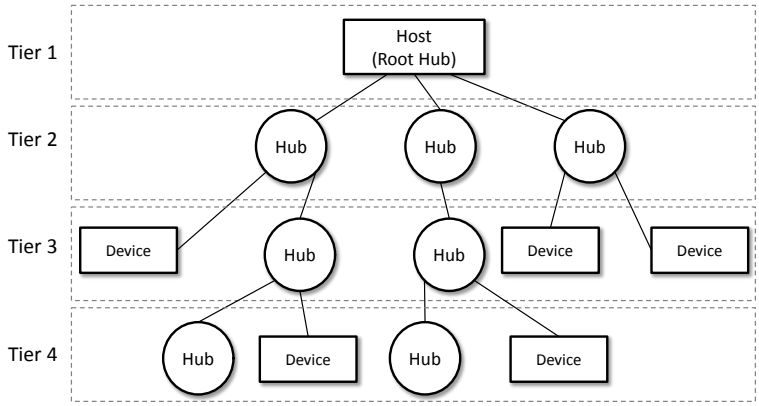

Figure 4.3. Packets sent from the host are broadcast onto the bus to all connected devices. The topology is a tiered star structure.

casting packets onto the USB. The less obvious types are the implicit information flows caused by state-effects on the host, i.e., a tainted device affecting the host's state. This implicit flow comes in the form of a timing channel because the amount of time in which the host communicates with a device can be observed by another device. It is very similar to the implicit information flow elicited by I^2C as previously discussed.

In order to accurately model the USB protocol, we designed a USB Host and Device in RTL Verilog HDL and followed the testing flow as shown in Figure 2.5 in Chapter 2. These behavioral Verilog modules were functionally verified at the RTL level using Modelsim SE 6.6b [69]. As mentioned for I^2C, testing all possible combinations is infeasible since the number of states is exponential. Thus, we have chosen a typical communication scenario consisting of 2 devices and a Host controller as shown in Figure 4.4. We have the Host send a packet indicating a write to Device 1 and then subsequently sends a data packet. Device 1 completes the transaction by responding with a handshake or acknowledgement packet. The Host then repeats the same procedure with Device 2.

Once the design was verified at the RTL level, we synthesized the designs to a gate-level netlist using Synopsys Design Compiler. We simulated the gate level design under the identical test conditions as those at the RTL level and verify that the circuit

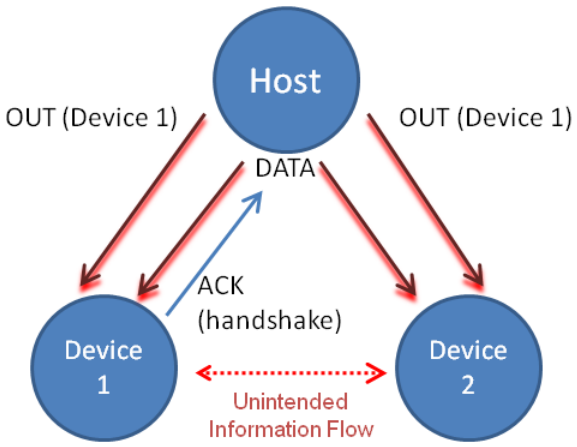

Figure 4.4. Host broadcasts to Device 1 and observed by Device 2. Subsequent broadcasts cause an implicit information flow between Device 1 to Device 2 through the Host.

has functionally equivalent operations. Once verified, the gate level designs were post processed such that the tracking logic for each gate is generated. Once completed, the entire circuit has its original functionality with the addition of precise tracking logic.

The resulting gate level design with GLIFT logic was then simulated again using the same test scenario except one of the devices was labeled as *tainted* (i.e., tainted 1). In doing so, we were able to confirm that two devices, which are not physically connected, influence each other through implicit channels as shown in Figure 4.4. Once the host controller finishes sending packets, its state is explicitly dependent on the handshake packet received from Device 1 in a similar manner as I^2C. This information flow is captured by GLIFT and the resulting host state machine becomes tainted. Subsequently, when the Host broadcasts data, this taint is propagated to Device 2 resulting in an implicit flow between the two devices on the bus even though they are not physically on the same wire. This flow is again the result of a timing channel.

To solve this problem, we need to devise a way to reset the master back to a known state and isolate specific paths from downstream transmission. The next subsection discusses a unique TDMA solution to preventing these unintended information flows and

providing isolation.

4.2.1 Enforcing Non-interference in USB

The TDMA solution works by modifying the Host such that it arbitrates between tainted and untainted states using a TDMA unit. In other words, the Host operates using a particular state in a fixed time slot. Once this time slot expires, the Host will switch out its state with another one (i.e., swaps out tainted for untainted). If the former was a tainted state, the switch will cause all the hardware in the Host to return to untainted. The timer allows each state machine to operate in a mutually exclusive manner. The fixed TDMA time slots prevent any timing information from flowing between the state machines since the state machines themselves have no influence on the arbitration. Conceptually, the TDMA unit in the Host acts as supervisor to the two state machines and has complete control of when tainted or untainted states can run. To account for explicit flows, the devices are tri-stated from the host when their time slot is not active to guarantee that they are not snooping on the bus.

With these additions, we synthesized the design and tested this scenario using the testing flow shown in Figure 2.5. We simulated our scenario for a complete time slot and verified that the information flows were contained. Again, this proves that unintended information flows are eliminated for subsequent time slots for this particular scenario. This includes information that flows through timing channels. As in I^2C, this solution results in some hardware and performance overheads and the next subsection will discuss these in more detail.

4.2.2 USB Non-interference Overheads

The new TDMA based solution to USB does have some minor penalties in simulation time. Yet, it results in minimal hardware overhead because the majority of

Table 4.3. Time spent simulating the mentioned test scenario on USB with and without GLIFT.

Design Type	**2 Devs**	**4 Devs**	**8 Devs**
TDMA Gate Design	110ms	171ms	281ms
TDMA w/ GLIFT	187ms	297ms	531ms

the hardware does not need to be reproduced. Specifically, we are only required to have additonal logic to arbitrate between states. Once the timer expires, a new state is loaded into the Host and the old state is overwritten in a similar manner as a context switch. All internal buffers, counters, etc. remain the same.

The simulation times for the original USB design and the one equipped with GLIFT can be seen in Table 4.3. As in I^2C, the aformentioned test scenario was executed with 2, 4, and 8 devices on the bus. This means that we were required to replicate 2, 4, and 8 state machines respectively in the Host. This is done because a state machine is needed for each outgoing port of the Host if non-interference is to be enforced between all devices. Unlike I^2C, the simulation time does not necessarily double as devices are introduced. We suspect this is due to the fact that the majority of the hardware in the host remains the same. Thus introducing devices to the system increases the simulation time by an amount proportional to the size of the device plus a small overhead in the Host due to additional arbitration logic. Also, the simulation times for the design with GLIFT scale roughly by the same factor as the design without GLIFT. Again, we expect that the difference in time between the system with and without GLIFT to be more significant as more test scenarios are performed.

This implementation incurs a 12.6% increase in area over the original host controller for replicating a single FSM. This overhead includes the timer and logic to select between state machines. Additional FSMs are needed for each port on the Host, assuming that non-interference is to be enforced between all devices. With that in mind, the area

Table 4.4. Area Overhead for Replicating State Machines

Number of FSMs	Area Overhead
2	12.6%
4	33.4%
8	77.4%
16	157.5%
32	322.9%

overhead increases linearly with the increase in FSMs as shown by Table 4.4. These results show that much of the hardware can be re-used since the amount of overhead increases by a constant factor associated with the TDMA and extra arbitration logic. With many state machines, this overhead becomes quite large because the extra arbitration logic begins to dominate the base functional logic. The significant drawback with this design is performance. With any TDMA based scheme, performance is potentially reduced because if a device does not use its time slot when it is active, the time slot is wasted. However, such a reduction in performance can likely be tolerated at the benefit of strong information flow policies.

This chapter, in full, is a reprint of the material as it appears in the Design Automation Conference 2011. Oberg, Jason; Hu, Wei; Irturk, Ali; Tiwari, Mohit; Sherwood, Timothy; Kastner, Ryan; This dissertation author was the primary investigator and author of this paper.

Chapter 5

Testing Timing Information Flows in Larger Systems

System-on-chips (SoCs) find themselves at the heart of these issues since they rely on the re-use of third-party intellectual property (IP) cores. These cores include memories, digital signal processors (DSP), graphical processing units (GPUs), analog RF blocks, I/O interfaces, and other various hardware accelerators (such as hardware encryption units). The SoC tightly integrates these cores together using a SoC bus architecture such as the Opencores WISHBONE. Ideally, integration of these components would be done in a realible and secure manner. Unfortunately, since many of these cores come from potentially untrusted sources, their use in high-assurance applications becomes extremely limited. This stems from the fact that these cores either come from an untrusted vendor or they have not been evaluated to the same extent as the trused cores. For example, the Mars Rover requires separation between the flight critical and scientific measurement systems simply because the flight critical components require detailed evaluation far beyond that of the measurement ones. A missed bug or vulnerability in the measurement components could affect the flight control components and desecrate the integrity the entire system.

One concern in mix-trusted SoC integration is due to malicious inclusions such

as hardware trojans. These trojans can violate security by using hidden circuitry to either covertly transmit information or insert a kill switch into the system. A survey by Tehranipoor et al. [99] covers many of the detection techniques including power and timing-based analyses. The work in this thesis can help deal with hardware trojans, but requires additional techniques to help mitigate their effect. We can ensure hardware trojans in untrusted cores do not affect trusted ones but we must explicitly assume trusted cores do not have trojans.

This thesis shows how a SoC can be designed using cores from different trust levels and have its security tested using GLIFT in Chapter 5. In doing so, I will demonstrate that untrusted cores never affect trusted ones. Specifically, we target the WISHBONE [84] SoC protocol using a cross-bar interconnect. We design a realistic system which resembles that of what one might find in high-assurance applications. Specifically, two processors (trusted and untrusted) which wish to share a hardware accelerator (AES encryption unit) in the SoC. Ideally, this sort of behavior should be allowed as long as the untrusted component does not interfere with the trusted one (and thereby compromise the integrity of the system). Using GLIFT, we show how a cross-bar can be designed and tested to be information flow secure such that the untrusted processor never affects the trusted one. This allows the hardware accelerator to be shared in a secure way without causing harmful side effects to the trusted computation. We demonstrate that this isolation is maintained across several different scenarios in which the untrusted processor is attempting to interfere with the trusted one.

Secure mix-trusted integration is not impossible if appropriate techniques are in place to build the system securely from the ground up. By designing a secure computing foundation, information flow can be tightly bounded in the system. Such techniques are hard to come by since information can flow through difficult to detect side-channels in hardware; e.g. the amount *time* a computation takes to execute. In this chapter, I will

demonstrate that GLIFT can be usd to detect and eliminate timing-based side channels in an SoC. These channels arise from contention of a shared hardware accelerator between two processors: one trusted and one not.

5.1 Designing a Secure Crossbar in Wishbone

WISHBONE is a SoC protocol originally developed by the Opencores community. It is a relatively simple protocol that allows easy integration of different cores into a design. WISHBONE itself is very flexible and allows many different interconnect configurations and bus transactions. WISHBONE allows many connectivity configurations including: point-to-point, data-flow, shared bus, and cross-bar interconnect.

WISHBONE itself is very flexible and allows many different interconnect configurations and bus transactions. WISHBONE allows many connectivity configurations including: point-to-point, data-flow, shared bus, and cross-bar interconnect. In this paper, we focus on the cross-bar interconnect since it provides a flexible interface for systems which contain large numbers of cores interacting in parallel. This particular type of configuration can be seen in Figure 5.1.

A cross-bar, put simply, is essentially a way for multiple devices to interact with each other. It consists of a set of control lines for "granting" access to a particular device if it is available. If some device A wishes to talk to device B, it will request a connection to device B through the crossbar and it will be granted if B is not busy (already talking to another device). More specifcally, the cross-bar interconnect logic is designed to observe requests from the masters by monitoring their `cyc` signal. If a request is received (`cyc` asserted), the cross-bar logic routes the connections from the master to the requested slave by decoding the address lines of the master. At this point, a transaction occurs between the master and slave: READ/WRITE, BLOCK transfer, or Read-Modify-Write (RMW). In this paper, we focus on the simple READ/WRITE bus cycles since they are

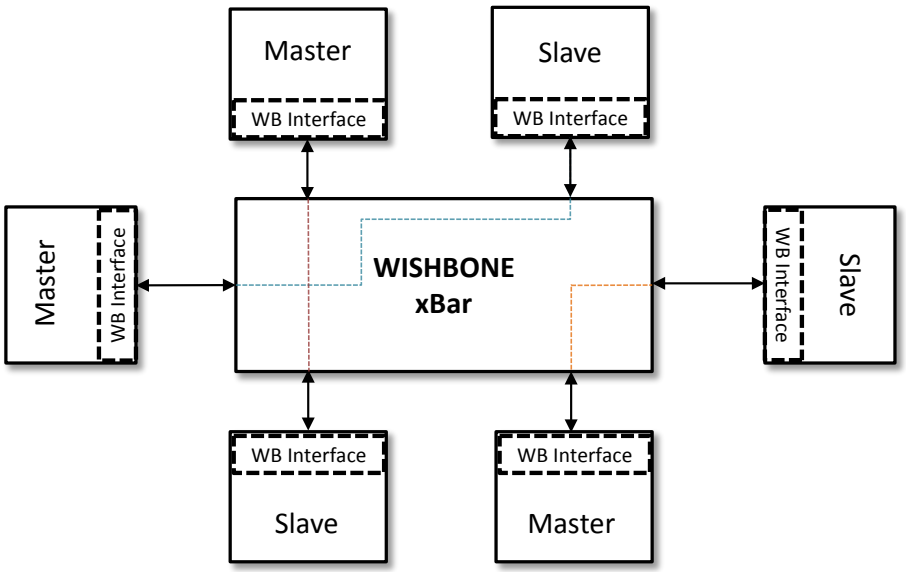

Figure 5.1. The WISHBONE system-on-chip architecture with cross-bar (xBar) interconnect. Many masters and slaves exist in the system and bus cycles initiate with a request by a master.

the most primitive of the three types. To complete a READ, the master deassarts the `we` signal, waits for the slave to assert `ack` and then reads the `data` lines and deasserts its `cyc` signal. A WRITE transaction occurs in a similar manner except `we` is asserted and the `data` lines are written by the master. Once a transaction completes, the interconnect logic disconnects the master and slave to make the slave available. If contention for a slave occurs, the master which issues its request later waits until the slave is available. WISHBONE itself supports time-outs and retries, but for this particular work we are not concerned with these optimizations. The verilog code for the cross-bar used in this system can be found in the Appendix of this dissertation.

We wish to demonstrate that multiple cores can access a shared resource in a safe and secure manner. We designed a system which consists of two MIPS-based processors and a 128-bit Advance Encryption Standard (AES) core. The two processors share the AES core over the WISHBONE interface. We assume that one of these processors runs critical code while the other is untrustworthy, e.g., running unknown (potentially

malicious) code or not being as thoroughly evaluated as the trusted core. Further details of this system are discussed in the next subsection.

5.1.1 Mix-Trusted System with Hardware Accelerator

Our system consists of two MIPS-based processors and a 128-bit AES core. We designed the MIPS based processor and the 128-bit AES was obtained from the Opencores [83] website. All cores are written in Verilog HDL. We chose this configuration because it well suits the common issues found in high-assurance applications. Namely, it is often desirable to share a hardware accelerator in a large SoC with mix-trusted components. Although this system is does not have all the complexity of commercial SoCs it does capture the main idea that multiple mix-trusted cores share common hardware resources and isolation between them should be maintained.

Figure 5.2 (a) shows the overview of our system. It consists of two of our processors and a 128-bit hardware AES unit. One of these processors is treated as untrusted (U) and the other trusted (T). In other words, we do not trust the behavior of processor U and assume its intentions are to corrupt the execution of T. Our MIPS based processor is fully functional and can execute many of the SPEC 2006 benchmarks (e.g. mcf, specrand, bzip2) [43]. To execute these applications (which are written in C), we used the SESC gcc cross-compiler to compile to MIPS binaries [90]. These binaries are loaded into our processor's memory and the executions are simulated using Mentor Graphics' Modelsim [69]. In order to communicate off-chip, we memory-mapped our processors WISHBONE I/O controller to a region of unused memory space. In order to communicate off-chip, we memory-mapped our processors WISHBONE I/O controller to a region of unused memory space. Specifically, we memory-mapped and added several WISHBONE configuration registers (`WB_CTRL_REG, WB_DATA_REG`, and `WB_ADDR_REG`) to our processor. Since we have a cross-compiler for our processor, we

wrote C-applications to push data out of the WISHBONE I/O interface. We wrote different applications for U and T to execute as we discuss later.

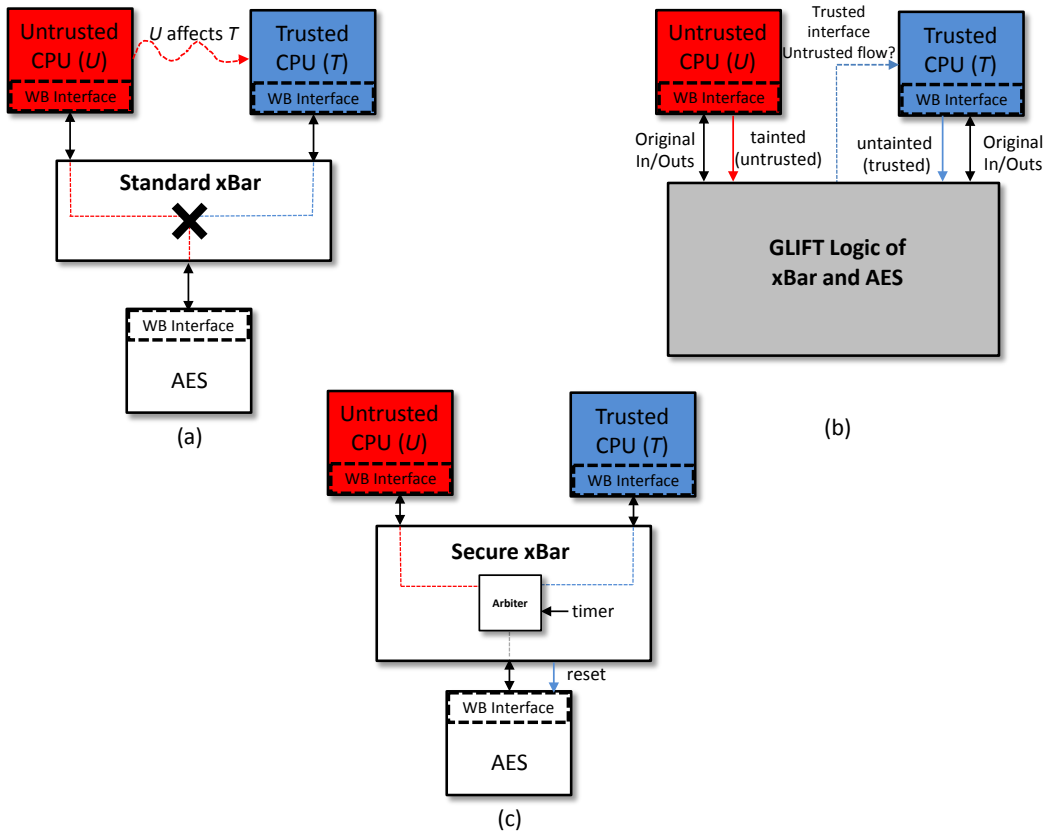

Figure 5.2. (a) The system used in our test scenario. This consists of two MIPS-based processors and a 128-bit AES encryption core. U and T contend for the use of the AES core. (b) The system after the AES core, xBar, and interface controllers have their GLIFT logic added. Information is observed to flow from U to T. (c) The final information flow secure system uses a time-multiplexed arbiter with a trusted reset to ensure information flow isolation between U and T. Adding the GLIFT logic to this system shows no information flowing from U to T.

We also designed the cross-bar interconnect to handle requests from the processors. The cross-bar interconnect is connected to each processor's WISHBONE controller (Figure 5.2 (a)). This cross-bar interconnect handles requests from the two processors in a round-robin fashion. This is simply for correctness and to prevent any sort of denial of service. Each processor can perform at most one transaction before having to relinquish

control of the bus. It waits for requests from a master and grants access to the slave at the address specified if the slave is available. In our scenario, we have only a single slave: a 128-bit hardware AES unit. Depending on the request type, this AES unit will take the data passed to it (in 32-bit chunks) and encrypt/decrypt a 128-bit block. The processor which requested the bus cycle polls until the transaction is complete and then retrieves the data from the AES unit. Upon completion, the next processor (if it has a pending request) will get access to the AES core.

Note that all the communication between the processor and AES unit are through WISHBONE and its cross-bar interconnect. In this system, since we have both trusted and untrusted processors contending for the use of the AES unit, there is likely to be information flows from U to T. Such a flow would violate the integrity of T and should be prevented. Moreover, this interference is not a denial of service attack since it is not possible for U to keep T from completing its work. Still, U can effect when T gets access to the AES block because it must wait for U's transaction to complete. For example, if U never wants to use the bus, and T performs continuous bus transactions, T can finish in some time t. However, if U performs bus transactions every time it is scheduled, T will finish its bus transactions in time $\approx 2t$. Thus U can affect the *time* in which T finishes execution but cannot prevent it from doing so. The next section discusses how we identify information flows in this system and how to eliminate them.

5.1.2 Building a Secure Cross-Bar for WISHBONE

To first illict how an information flow occurs from U to T, we test a scenario in which T encrypts a 128-bit block of text using the AES unit and subsequently decrypts the cipher-text to verify the result. In parallel with T, U continuously reads a configuration register on the AES core. We call this program executing on processor U as R_CONF. This scenario was chosen to show an information flow because U is not overwriting any

Table 5.1. Description and results of different applications executed on U and T. Untrusted flows are identified in the base cross bar for most scenarios and none are identified in the secure cross-bar. Flows do not occur if T does not use the WISHBONE interface as in the cases of running MM.

p **on** U	τ **on** T	**Flow in Secure xBar**	**Flow in Base xBar**
AES	*MM*	**NO**	**NO**
MM	*AES*	**NO**	**YES**
R_CONF	*AES*	**NO**	**YES**
R_ALL	*AES*	**NO**	**YES**
W_ALL	*AES*	**NO**	**YES**
AES	*AES*	**NO**	**YES**

of T's data since it is only reading. In other words, U is not directly corrupting T's data on the AES block and at first glance U seems to be non-interfering with T.

Since we are concerned with the information flow from U to T, we need to look at the information flowing out of U and in to T. To be precise, let $T_{in_t} = \{data\_i_t, ack\_i_t\}$ be the taint input wires to T from the wishbone logic. We determine whether or not a flow occured by identifying whether any wire in T_{in_t} is every set to 1. To do so, we must track the flow of information through the cross-bar, the AES unit's WISHBONE controller, and the AES unit itself. To track this flow of information, we follow the same static testing method presented in Chapter 2. Namely, we process the cross-bar and the AES unit with its WISHBONE interface through synthesis using Synopsys' Design Compiler to achieve a gate-level netlist. Subsequently, we add the GLIFT logic to these components and re-insert this logic into the system as shown in Figure 5.2 (b). We then execute *R_CONF* on U by simulating the Verilog in Modelsim [69]. From the simulation, as shown in Figure 5.3, a tainted flow is observed entering T's inputs as soon as it requests an AES transaction ($\{data\_i_t, ack\_i_t\} = \{0xF \cdots F, 1\}$). Since we only tainted the ouputs of U, it must be the source of this tainted information flow.

This flow occurs because U and T contend for the use of the encryption unit.

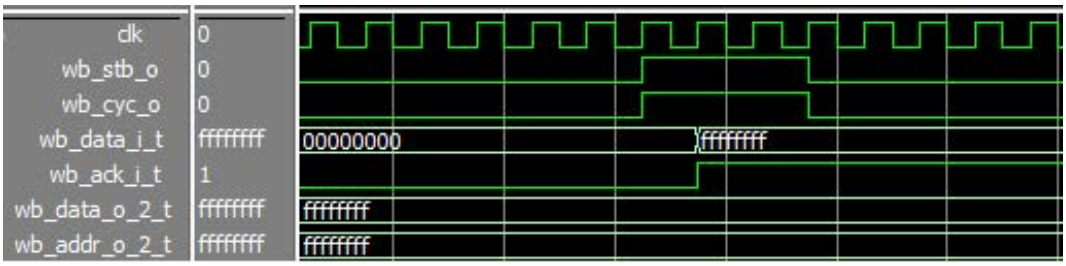

Figure 5.3. Waveform showing tainted information flow. As soon as T requests access to the AES unit (`wb_stb_o = wb_cyc_o = 1`) tainted information flows to its inputs ($\{$`data_i_t, ack_i_t`$\}$ = $\{$`0xF`$\cdots$`F, 1`$\}$). U's outputs were the only marked as tainted, so this flow must have originated from U.

Specifically, U affects the execution of T indirectly by its use of the AES unit. This flow can be regarded as occuring through a timing channel. That is, U is able to affect the *time* in which T finishes its computation (U is only reading and therefore does not directly affect the computation of T). Such channels can violate the integrity of the design because they can potentially violate real-time constraints where T must meet a critical deadline but is unable to because of U. To solve this problem, we put in place a way for U to never affect T's use of this resource. Specifically, we introduce a time-multiplexed arbiter with a *trusted reset* to the cross-bar which forces T and U to operate in mutually exclusive time slots as shown in Figure 5.2 (c). Upon expiration of a time-slot, the logic is restored to a known state to ensure harmful content is left behind. As we see in the next section, this new cross-bar eliminates this untrusted flow.

5.1.3 Secure Cross-Bar Evaluation

To demonstrate the lack of information flow using this new cross-bar, we construct several different programs which have malicious characteristics of causing interference to the trusted computation on T. Specifically, we show non-interference for a fixed set of programs. Non-interference states that U should never affect T through any sort of digital information. This includes both directly corrupting the data of T or affecting the

time in which programs on T take to complete. This ensures not only the integrity of the data on T, but also the integrity of the timing of the computation. To demonstrate this property for a set of programs, let $P = \{p_1, p_2, \cdots, p_n\}$ be a set of programs to be run on U. We want to show non-interference with respect to P by demonstrating that no untrusted information flows to the inputs of T:

$$\forall p \in P.\, S(p \,||\, \tau) \overset{c}{\Rightarrow} T_{in_t} = \{0,0\} \tag{5.1}$$

where $S(p \,||\, \tau)$ is the system executing with p on U and τ on T and $\overset{c}{\Rightarrow}$ is an implication over all clock cycles c. T_{in_t} is the set of taint inputs from the wishbone cross-bar as previously defined. This definition says that for any program p in a set P, when executing p on U with some trusted computation on τ on T, no untrusted information from U flows to the inputs of T during any clock cycle. Since GLIFT can also capture information flowing through timing channels as mentioned in the previous section, this includes information which affects the time in which τ takes to complete.

For our particular test scenario, we build the set $P = \{MM, R\_CONF,$ $R\_ALL, W\_ALL, AES\}$. MM is a simple matrix multiply program. R_CONF is the same program as before which continuously reads a configuration register on the AES core. R_ALL attempts to read the entire address space associated with the AES core. W_ALL attempts to write the entire address space associated with the AES core. Lastly, AES uses the AES core to encrypt then decrypt some information. All of these applications are written in C, compiled to MIPS, and loaded on to their respective processor's instruction memory. Table 5.1 presents an interesting subset of our test cases and summarizes the outcomes. We do not present all results due to space constraints but observed that Definition 5.1 holds for each τ we tested.

For all cases in which τ accesses the WISHBONE fabric, untrusted information

flows from U to T in the unsecure cross-bar, thus violating Definition 5.1. One interesting case is when *MM* and *AES* are run on T and U respectively. In this case, no untrusted information flows to T simply because τ never accesses the AES core. Its execution is independent of the behavior of U. Conversely, another interesting case arises when U runs *MM* and T runs *AES*. In this case, even though p is not using the AES core, the *lack* of its use still affects the behavior of τ. This lack of use allows τ to finish faster than if p were accessing it; a flow of information. GLIFT indicates no flow ($T_{in_t} = \{0,0\}$) for all applications when the secure cross-bar is used. In other words, non-interference is upheld for these computations on U.

It is important to make a couple of notes on this solution. First, the arbiter only time-multiplexes this specific resource and not the cross-bar as a whole. The goal of the cross-bar interconnect is to allow parallelism; multiplexing the entire cross-bar eliminates this flexibility. This parallelism can still be maintained since U can be granted access to other devices in the system in parallel with T and isolation can still be maintained. In addition, ideally this property (Definition 5.1) would be shown for all possible programs on U to demonstrate complete non-interference. However, such an exhaustive test would be impractical in this case. Some recent work on GLIFT has made an effort to solve this problem by introducing Star-Logic [102] which uses an abstract execution to make exhaustive testing possible. Unfortunately most of this work is still in its early stages, but we plan to employ these techniques in future research.

Overheads Due to GLIFT Analysis

It is difficult to quantify the amount of testing overhead required for this analysis since it requires both the intuition of the designer including the cleverness of chosen test vectors and detailed knowledge of the design. However, the overhead due to simulation time can be quanitified, so we use this as a metric for evaluating the overhead. It is worth

noting that the area overhead of GLIFT is substantial ($\approx 3.5X$), but as exemplified in previous sections, the logic is used simply during simulation and has no manifestation in the final design.

Table 5.2 shows the simulation times for the base RTL system, base system with GLIFT logic, the system with the secure cross-bar RTL, and the secure system with GLIFT logic. The simulation for the system with the base RTL is longer than the one with the secure cross-bar simply due to the isolated time-slots. In otherwords, U cannot affect the time it takes for T to access the slave. Therefore the secure cross-bar system completes in simply fewer simulated clock cycles. However, if we compare the RTL version with their respective GLIFT ones (which execute in the same number of respective clock cycles), the overheads in simulation times are apparent. This is due to the extra logic imposed by GLIFT and the requirement for the simulation tool to analyze more logic than the RTL. As shown in Table 5.2, the simulation time due to GLIFT logic is 2X longer than the original design in the worst case. However, for such a strong security guarantee, such overheads are likely to be tolerated especially if they provide the flexibility of using untrusted components in a SoC.

Table 5.2. Simulation times for various stages of our design. The base RTL refers to what is shown in Figure 5.2 (a). The GLIFT logic incurs at most a 2X overhead in simulation time in this system.

Design	Simulation Time
Base RTL	502.08 ms
Base w/ GLIFT	718.03 ms
Secure xBar RTL	353.60 ms
Secure xBar w/ GLIFT	698.50 ms

Building these systems in a secure manner requires strict design practices and tools. In this chapter, I showed how mix-trusted IP cores can be integrated in a secure manner. By using gate-level information flow tracking to show information flow isolation

between trusted and untrusted cores, we have constructed a secure cross-bar interconnect for the WISHBONE SoC bus architecture. This powerful property makes it possible to integrate mix-trusted cores and verify the security of their interactions. This ultimately reduces the cost and time associated with development and makes using untrusted cores in high-assurance applications more of a possibility.

This chapter 5, in full, is a reprint of the material as it appears in the IEEE journal on Design and Test of Computers 2013. Oberg, Jason; Sherwood, Timothy; Kastner, Ryan. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Formalizing Timing Channels at the Gate-level

GLIFT's low-level properties provide a promising remedy to identifying timing channels. Since GLIFT targets the lowest digital abstraction, it is able to detect and capture information leaking through time. This claim, however, is made in some of the initial work on GLIFT done by my colleagues and myself [103, 101, 78] but never thoroughly formalized. One of the specific contributions of this chapter is to make this formalism much more apparent: that GLIFT can detect timing channels.

In addition, if a hardware designer using GLIFT detects that there is an information flow, there is no way to separate out the timing information from other *functional* information. Using a shared bus as an example, if a hardware designer were to observe an information flow using GLIFT, it would not be obvious whether or not this flow was from direct means (a device corrupting data on the bus) or by affecting another device's response time. The help solve this problem, we present a formal model in Chapter 6.4 that, when used in conjuction with GLIFT, isolates timing information from other flows of information. This model incorporates details from my published work [79] and provides more thorough and complete definitions, another application example (a shared bus), and more detailed discussion. As briefly mentioned, whether or not these timing flows are in

the threat model depend on the system at hand. Nevertheless, this framework provides a way for hardware designers to *reason* about these timing flows.

6.1 Threat Model

The specific threat model we target is hardware with potential timing channels that might adversely affect confidentiality or integrity. For confidentiality, we address the issue of designers being unable to determine whether or not an information leak is from timing or direct means. For example, caches have been of big concern when processes from different trust levels share cache lines. Data used by a secret program can be, and has been, extracted solely from the time it takes to perform memory operations.

For integrity, we address concerns related to timing-based interference. For example, if a hardware designer is building a system-on-chip and wishes to isolate high-integrity cores from less trusted third-party ones, while still allowing resource sharing, then he could use this framework to reason about the timing effects that the less trusted cores have on the high-integrity ones.

In both cases, our framework gives designers further insight into potential vulnerabilities so they can make better decisions. In some cases, these timing flows might be of no concern at all; i.e., the attack space of the cache or the timing effects on high-integrity cores are simply not in the threat model of the designer. Regardless, this work provides hardware designers with tools to more accurately evaluate their threat model, giving rise to increased confidence and more secure designs.

6.2 Preliminary Definitions

Before defining information flows and related concepts, we must first define some preliminary notions formally. Many of these notions are commonly understood by hardware designers, but we formulate them in such a way as to fit our model in a clear

and concise manner. We start with the notion of time; as we are working at the gate level, the only notion of time that we consider is the system clock.

Definition 3. *We define the* clock *to be a function with no inputs that outputs values of the form $b \in \{0,1\}$. We define a* clock tick *to be the event in which the output of the clock changes from* 0 *to* 1*. Finally, we define a* time *t to be the number of clock ticks that have occurred, and we define T to be the set containing all possible values of t.*

Our formal definition of time captures what we intuitively expect: some stateless hardware component will output a stream of ticks, and a separate stateful component will measure the number of ticks and use this to keep track of time. By keeping track of time, we can define an *event* as a given value at a certain point in time.

Definition 4. [62] *For a set Y, a* discrete event *is the pair $e := (y,t)$ for $y \in Y$ and $t \in T$ (where we recall T is the set of all possible time values). We also define functions that recover the value and time components of an event as $\mathsf{val}(e) = y$ and $\mathsf{time}(e) = t$ respectively.*

To keep track of how values change over time, we can also define a sequence of events as a *trace*.

Definition 5. *For a value $n \in \mathbb{N}$ and a set Y, we define a* trace *$A(Y,n)$ to be a sequence of discrete events $\{e_i = (y_i,t_i)\}_{i=1}^{n}$ that is ordered by time; i.e., $\mathsf{time}(e_i) < \mathsf{time}(e_{i+1})$ for all i, $1 \leq i < n$, and such that $\mathsf{val}(e_i) \in Y$, $\mathsf{time}(e_i) \in T$ for all i, $1 \leq i \leq n$. When the values of Y and n are clear, we omit them and refer to the trace simply as A.*

The way in which we have currently defined an event is quite broad: any value at any time can be considered an event. As an example, consider a system that outputs some value on every clock tick; if we run such a system for k clock ticks and record each output, then we will obtain a trace of size k. In many cases, however, events in

this trace may be redundant, as the system might output the same value for many clock ticks while performing some computation. In this case, we would be interested not in the entire progression of events, but only in the case when the value of the output changes. To capture this, we define the *distinct* trace.

Definition 6. *For a trace $A(Y,n)$, we define the* distinct trace *of A to be the largest subsequence $d(A) \subseteq A(Y,n)$ such that for all $e_{i-1}, e_i \in d(A)$ it holds that* $\mathsf{val}(e_i) \neq \mathsf{val}(e_{i-1})$.

Constructing the distinct trace $d(A)$ of A is quite simple: first, include the first element of A in $d(A)$. Next, for each subsequent event e, check whether the last event e' in $d(A)$ is such that $\mathsf{val}(e') = \mathsf{val}(e)$; if this holds, then skip e (i.e., do not include it) and if it does not then add e to $d(A)$. As an example, consider a trace of two-bit values $A = ((00,1),(00,2),(01,3),(01,4),(11,5),(10,6))$. Then the distinct trace $d(A)$ will be $d(A) := ((00,1),(01,3),(11,5),(10,6))$, as the values at time 2 and 4 do not represent changes and will therefore be omitted.

With these definitions in hand, we can model a finite state machine system F that takes as input a value x in some set X and returns a value y in some set Y in a similar manner as past work [61]. To be fully general and consider systems that take in and output vectors rather than single elements, we assume that $X = X_1 \times \ldots \times X_n$ and that $Y = Y_1 \times \ldots \times Y_m$ for some $m,n \geq 1$, which means that an input x looks like $x = (x_1,\ldots,x_n)$ and an output y looks like $y = (y_1,\ldots,y_m)$. To furthermore acknowledge that the system is not static and thus both the inputs and outputs might change over time, we instead provide as input a trace $A(X,k)$ for some value k, and assume our output is a trace $A(Y,k)$.

Definition 7. [61] *A finite state machine (FSM) F is defined as $F = (X,Y,S,\delta,\alpha)$, where X is the set of inputs, Y the set of outputs, and S the set of states. $\delta : X \times S \to S$ is the*

transfer function and $\alpha : X \times S \to Y$ *is the output function.*

Since we are dealing with circuit implementations of finite state machines, both δ and α are represented as combinational logic functions. In addition, both δ and α can be called on a trace. $B = \alpha(A, s_0)$ generates a trace of output events $B = (e_0, e_1, \cdots e_k)$ during the execution on input trace A starting in state s_0. This notation describes α executing recursively; it takes a state and trace as input and executes to completion producing an output trace. When the starting state is assumed to be the initial state, we use the notation $\alpha(A)$.

Now, since we are concerned with flows of information from a specific set of inputs (the subset of inputs which are of security concern), we need to formalize how to constrain the others. Recall first our intuition: an information flow exists for a set of inputs to the system F if their values affect the output (either the concrete value or its execution time). One natural way to then test whether or not these inputs affects the output is to change their value and see if the value of the output changes; concretely, this would mean running F on two different traces, in which the values of these inputs are different. In order to isolate just this set of inputs, however, it is necessary to keep the value of the other inputs the same. To ensure that this happens, we define what it means for two traces to be *value preserving*.

Definition 8. *For a set of inputs* $\{x_i\}_{i \in I}$ *and two traces* $A(X,k) = (e_1, \ldots, e_k)$ *and* $A(X,k)' = (e'_1, \ldots, e'_k)$*, we say the traces are* value preserving *with respect to I if for all* $e_i \in A$ *and* $e'_i \in A'$ *it is the case that* $\mathsf{time}(e_i) = \mathsf{time}(e'_i)$*, and if* $\mathsf{val}(e_i) = (a_1, \ldots, a_n)$ *and* $\mathsf{val}(e'_i) = (a'_1, \ldots, a'_n)$*, then* $a_i = a'_i$ *for all* $i \notin I$*.*

If two traces are value preserving, then by this definition we know that the only difference between them is the value of the *tainted* inputs $\{x_i\}_{i \in I}$. Taint will be formally defined shortly, but, as an example, secret data would be tainted and then tracked to

ensure that it is not leaking to somewhere harmful. In this example, the set of secret inputs would be the set I. We will use this definition in the next section to prove that GLIFT detects both functional and timing information flows.

6.3 Recap–Information Flow Tracking and GLIFT

As disucssed in Chapter 2, information flow tracking is a common method used in secure systems to ensure that secrecy and/or integrity of information is tightly controlled. Given a policy specifying the desired information flows, such as one requiring that secret information should not be observable by public objects, information flow tracking helps detect whether or not flows violating this policy are present. To assist in the understanding of the details presented in this chapter, this information will be quickly reviewed.

In general, information flow tracking associates data with a label that specifies its security level and tracks how this label changes as the data *flows* through the system. As an example, consider a system with two labels: `public` and `secret`, and a policy that specifies that any data labeled as `secret` (e.g., an encryption key) should not affect or flow to any data labeled as `public` (e.g., a malicious process). More generally, information flow tracking can be extended to more complex policies and labeling systems (i.e., in general `high` data should never flow to `low`); as such, it has been used in all levels of the computing hierarchy, including programming languages [94], operating systems [59], and instruction-set/microarchitectures [98, 27]. Recently, information flow tracking was used by Tiwari et al. [103] at the level of logic gates in order to dynamically track the flows of each individual bit.

In the technique used by Tiwari et al., called gate level information flow tracking (GLIFT), the flow of information for individual bits is tracked as they propagate through Boolean gates; GLIFT was later used by Oberg et al. [78] to test for the absence of all information flows in the I^2C and USB bus protocols and by Tiwari et al. [102] to build a

system that provably enforces strong non-interference. Further, it has been used to prove timing-based non-interference for a network-on-chip architecture in the research project SurfNoC [109]. Since its introduction, Tiwari et al. have expanded GLIFT to what they call "star-logic" which provides much stronger guarantees on information flow [102]. Briefly, GLIFT tracks flow through gates by associating with each data bit a one-bit label, commonly referred to as *taint*, and tracking this label using additional hardware known as *tracking logic*.

6.3.1 Formal definitions for GLIFT

To be precise, we present definitions of tracking logic and taint. Some of these definitions were presented earlier on in this thesis, but they will be re-discussed here for completeness. First, it is important to understand how a "wire" in a logic function is tainted. We define this formally as follows (this is the same as Definition 1 for taint presented in Chapter 3):

Definition 9 (Taint)**.** *For a set of wires (inputs, outputs, or internals) X, the corresponding* taint set *is X_t. A wire x_i for $x = (x_1, \ldots, x_i, \ldots, x_n) \in X$ is* tainted *by setting $x_{it} = 1$ for $x_t \in X_t$ and $x_t = (x_{1t}, \ldots x_{it}, \ldots x_{nt})$.*

In this definition, and in what follows, the elements of X and X_t are given as vectors; i.e., an element $x \in X$ has the form $x = (x_1, \ldots x_n)$ for $n \geq 1$. For single-bit security labels (which we use exclusively in this paper), $x \in X$ and its corresponding taint vector $x_t \in X_t$ are the same length.

Now that we have a definition for taint, we can formally define the behavior of a tracking logic function and information flow with a tracking logic function. The tracking logic function definition is the same as Definition 2 in Chapter 3.

Definition 10 (Tracking logic)**.** *For a combinational logic function $f : X \to Y$, the respective* tracking logic function *is $f_t : X_t \times X \to Y_t$, where X_t is the taint set of X*

and Y_t the taint set of Y. If $f(x_1,\ldots,x_n) = (y_1,\ldots,y_m)$, then $f_t(x_1,\ldots,x_n,x_{1t},\ldots,x_{nt}) = (y_{1t},\ldots,y_{mt})$, where $y_{it} = 1$ indicates that some tainted input x_j (i.e., an input x_j such that $x_{jt} = 1$) can affect the value of y_i.

Definition 11 (Information flow)**.** *For a combinational logic function $f : X \to Y$ and a set of inputs $\{x_i\}_{i\in I}$, an* information flow *exists with respect to an output y_j if $f_t(x_t) = (y_1,\ldots,y_{j-1},1,y_{j+1},\ldots,y_m)$, where each entry x_{it} of x_t is 1 if $i \in I$ and 0 otherwise. If there exists an index j such that $y_j = 1$, we just say an* information flow *exists.*

To understand how the tracking logic is used, consider a function with `public` and `secret` labels; then a label x_{it} is 1 if x_i is secret, and 0 otherwise. When considering a concrete assignment $(a_1,\ldots a_n)$ with each a_j being 0 or 1, running $f(a_1,\ldots,a_n)$ will produce the data output $(y_1,\ldots,y_i,\ldots,y_m)$, and running $f_t(a_1,\ldots,a_n,a_{1t},\ldots,a_{1n})$ will indicate which tainted input can affect the values of which outputs (by outputting $y_{it} = 1$ if a tainted input affects the value of y_i and 0 otherwise). Going back to our sample function, if we observe some output $y_{it} = 1$ from f_t, we know that a secret input affects the output y_i of f. If y_i is public, then this flow would violate the security policy.

Typically, each individual gate and flip-flop is associated with such tracking logic in a compositional manner. In other words, for each individual gate (AND, OR, NAND, etc.), tracking logic is added which monitors the information flow through this particular gate. By composing the tracking logic for each gate and flip-flop together, we can form an entire hardware design consisting of all the original inputs and outputs, with the addition of security label inputs and outputs. Care must be taken to derive the tracking logic for each gate separately, however, as the way in which the inputs to a gate affect its output vary from gate to gate. As an example, consider the tracking logic for a AND gate as shown in Figure 6.1.

Simply by definition, we know that if some input of a AND gate is 0, the output

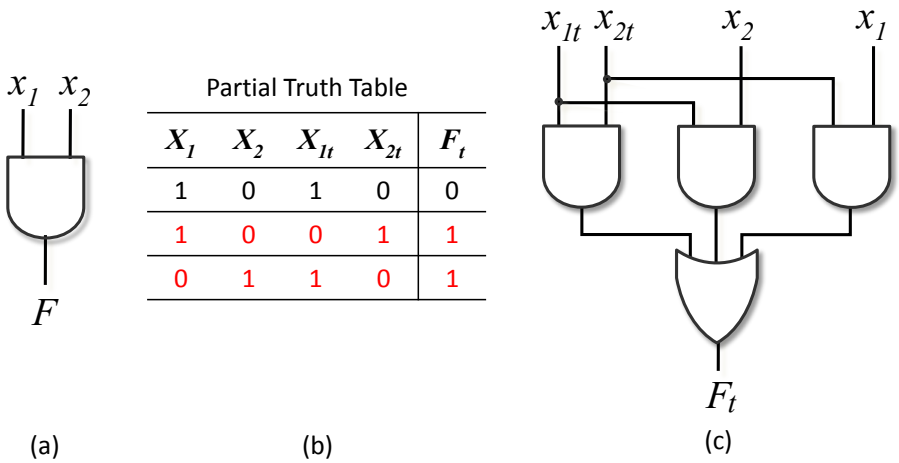

Figure 6.1. (a) A simple AND gate. (b) A partial truth table for the tracking logic of an AND gate. $F_t = 1$ *iff* a tainted input affects F. (c) The tracking logic for an AND gate.

will always be 0 regardless of the other inputs. In other words, if we have inputs $x_1 = 1$ and $x_2 = 0$ with security labels $x_{1t} = 1$ and $x_{2t} = 0$ as shown in Figure 6.1, then the output will actually be *untainted* even though $x_{1t} = 1$, because the value of x_1 has no observable effect on the output of the gate (again, because $x_2 = 0$ and thus the output will be 1 regardless). By building a truth table for every gate primitive, tracking logic can be derived in this manner and stored in a library; the tracking logic can then be applied to the gate in a manner similar to technology mapping. As an example of how to compose these tracking logics, we consider a 2-input multiplexer (MUX), which is composed of two AND gates and a single OR gate where the output of the AND gates feed the inputs of the OR gate. First, the tracking logic for each AND gate and the single OR gate is generated. Then, the output of the tracking logic for each AND gate is fed as inputs to the tracking logic for the OR gate.

To use GLIFT in practice, a hardware description of the design is written in a hardware description language (HDL), such as Verilog or VHDL, and this description is then synthesized into a gate-level netlist using traditional synthesis tools such as Synopsys' Design Compiler. A gate-level netlist is a representation of the design completely in logic

gates and flip-flops. Next, the GLIFT logic is added in a compositional manner (as we just described); i.e., for every gate in the system, we add associated tracking logic which takes as input the original gate inputs and their security labels and outputs a security label. Given a security policy such as our confidentiality example (i.e., secret inputs should not flow to the public output), GLIFT can then be used to ensure that the policy is not violated by checking that the output of the tracking logic f_t is not 1. It is important to remember that f_t is defined to report 1 *iff* a tainted input can actually *affect* the output. In other words, it will report 1 if at any instant in time a tainted input can affect the value of the output.

One of GLIFT's key properties is that it targets a very low level of computing abstraction; at such an abstraction, all information becomes explicit. In particular, because GLIFT tracks individual bits at this very low level, it can be used to explicitly identify timing channels. To support this claim, the following sections present some preliminary definitions and a model that, when used in conjunction with GLIFT, can test for timing channels. Such a model will be used in this paper to identify timing channels in a shared bus in Section 6.5 and CPU cache in Section 6.6.

6.3.2 GLIFT and Timing Channels

In order to have a clear understanding of timing channels, it first helps to specify a definition of a timing channel familiar to hardware designers. We define specifically a timing-only flow, where an input affects only the timestamp of output events and not the values. To be clear, we are concerned with timing leaks at the cycle level. Stated differently, we assume that an attacker does not have resources for measuring "glitches" within a combinational logic function itself. Rather, he can only observe timing variations in terms of number of cycles at register boundaries. With these assumptions, we present this definition in order to prove that GLIFT in fact captures such channels.

Definition 12 (Timing-only flow)**.** *For a FSM F with input space X and output function α, a* timing-only flow *exists for a set of inputs $\{x_i\}_{i\in I}$ if there exists some value $k \in T$ and two input traces $A(X,k)$ and $A(X,k)'$ such that A and A' are value preserving with respect to I, and for $B = \alpha(A)$ and $B' = \alpha(A')$ it is the case that $\mathsf{val}(e_i) = \mathsf{val}(e'_i)$ for all $e_i \in d(B)$ and $e'_i \in d(B')$ and there exist $e_j \in d(B)$ and $e'_j \in d(B')$ such that $\mathsf{time}(e_j) \neq \mathsf{time}(e'_j)$.*

This definition captures the case in which a set of inputs affect only the *time* of the output. In other words, changing a subset of the tainted inputs will cause a change in the time in which the events appear on the output, but the values themselves remain the same. Before we can use this definition to prove that GLIFT captures timing-only channels, we need to define the GLIFT FSM F_t.

Referring back to Definition 7, a FSM consists of two combinational logic functions α and δ. Thus there exists tracking logic functions α_t and δ_t according to Definition 10. Using this property, we can define the GLIFT FSM F_t, which will be used to prove that GLIFT detects timing-only flows.

Definition 13. *Given a FSM $F = (X,Y,S,\delta,\alpha)$, the FSM tracking logic F_t is defined as $F_t = (X,X_t,Y_t,S,S_t,\delta_t,\alpha_t)$ where X and S are the same as in F, S_t is the set of tainted states, X_t is the set of tainted inputs, Y_t is the set of tainted outputs, δ_t the tracking logic of δ and α_t the tracking logic function of α.*

Now that these definitions are in place, we can prove that GLIFT can detect timing-only flows.

Theorem 1. *The FSM tracking logic F_t of a FSM F captures timing-only channels.*

Proof. Suppose there exists a timing-only channel for a finite state machine F with respect to the set of tainted inputs I. By Definition 12, this means there must exist value-preserving traces $A(X,k)$ and $A(X,k)'$ such that, for $B = \alpha(A)$ and $B' = \alpha(A')$,

$\mathsf{val}(e_i) = \mathsf{val}(e'_i)$ for all $e_i \in d(B)$ and $e'_i \in d(B)$, but there exist $e_j \in d(B)$ and $e'_j \in d(B')$ such that $\mathsf{time}(e_j) \neq \mathsf{time}(e'_j)$. Since $e_j \in d(B)$ implies that $e_j \in B$ (and likewise for e'_j), this means that $B \neq B'$.

F generates an output every clock tick, so for all $e_j \in B$ and $e'_j \in B'$, $\mathsf{time}(e_j) = \mathsf{time}(e'_j)$, and thus there must exist some $e_\ell \in B$ and $e'_\ell \in B'$ such that $\mathsf{val}(e_\ell) \neq \mathsf{val}(e'_\ell)$ (because $B \neq B'$). By Definition 8, all input values remain the same for all $i \notin I$, meaning the only difference between them is in the tainted inputs, and thus the difference in output must have been caused by a tainted input. By Definition 10, α_t would thus have an output of $(y_{1t}, \ldots, y_{\ell t} = 1, \ldots, y_{mt})$, as the value if y_ℓ in the output of α was affected by a tainted input. By Definition 11, this means GLIFT has indicated an information flow must exist. As the only possible flow is timing-based, GLIFT thus captures timing-only flows. □

Since GLIFT operates at the lowest level of digital abstraction, all information flows become explicit. Thus, if at any instant in time a tainted input can affect the value of the output, GLIFT will indicate so by definition. At the FSM abstraction, as defined in Definition 12, this type of behavior often presents itself as a timing channel. This proof demonstrates that GLIFT can in fact identify these types of information flows. What is needed, however, is to formally understand how to separate these types of timing flows from other *functional* ones. In the next section we demonstrate how GLIFT can be used in conjuction with finding functional flows to isolate this timing information.

6.4 Isolating Timing Channels

As discussed in the previous section, GLIFT allows system designers to determine if any information flows exist within their systems even those through timing-channels. To be concise, at the digital level, there are two possible types of flows which we name *functional* flows and *timing*, as seen in Figure 6.2. Intuitively, a functional flow exists for

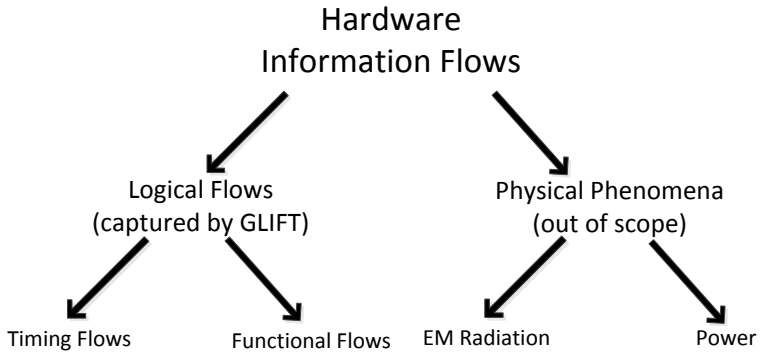

Figure 6.2. The classes of information flows in hardware. In this work, we are concerned with logical flows that GLIFT captures, including timing and *functional* flows. Physical phenomena are out of the scope of this work.

a given set of inputs to a system if their values affects the values output by the system (for example, changing the value of a will affect the output of the function $f(a,b) := a+b$), while a timing flow exists if changes in the input affect how long the computation takes to execute. While GLIFT will tell the designer only if any such flows exist, in this section we create a formal model for determining whether or not the system contains specifically functional flows. When used in conjunction with GLIFT, this technique therefore allows us to also determine what type of flow is occuring: if GLIFT determines that no flow exists, then clearly there is no flow. If instead GLIFT determines that a flow does exist but we can demonstrate that no functional flow exists, then we know that a timing flow must exist. What is left open, however, is the interesting case in which GLIFT determines that a flow exists but we determine that a functional flow does exist; in this case, we are unable to determine if a timing flow exists as well.

6.4.1 Finding Functional Flows

We can then create a testing framework as shown in Figure 6.3. Here GLIFT is used in conjuction with finding functional flows to isolate timing information. If GLIFT determines that there is no flow, we know there is no functional nor timing information

flow. If, however, GLIFT determines there is a flow and we can find no functional flow, then we know that the information flow occured from a timing channel. In this section, we discuss how to find functional flows. We begin with the strongest possible definition and then weaken it to make it more amenable to testing techniques familiar to hardware designers.

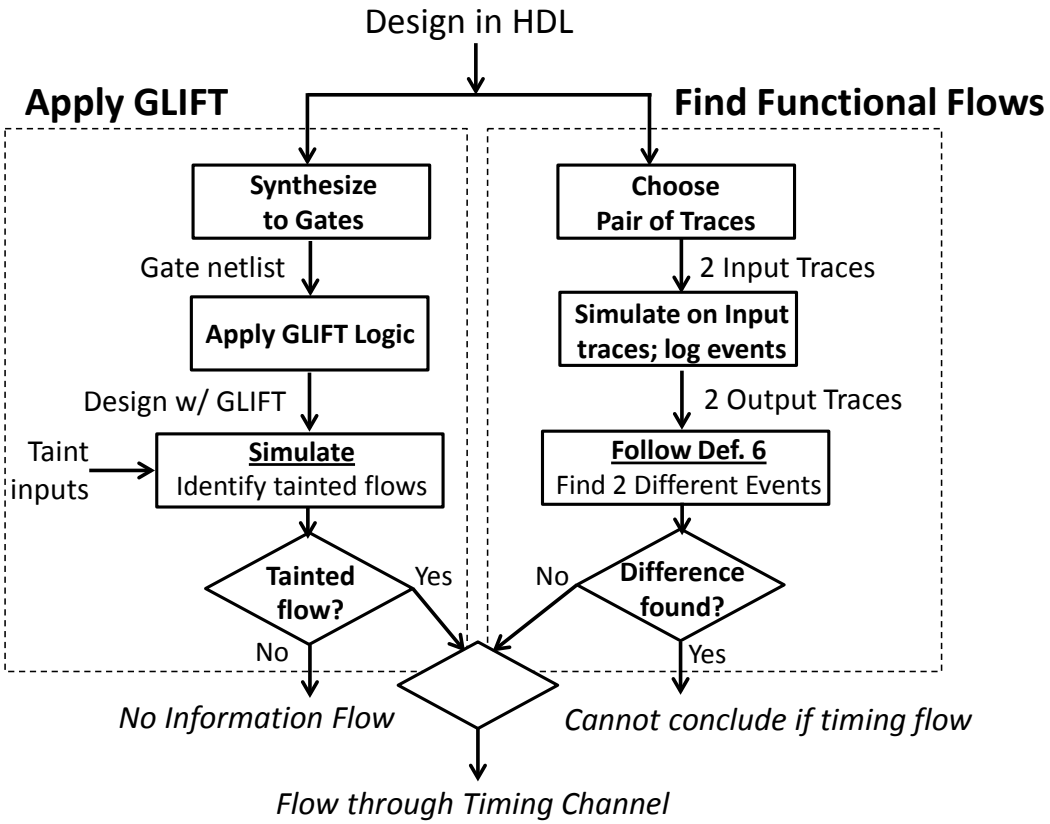

Figure 6.3. How our method can be used with GLIFT to isolate timing channels. If GLIFT says there is a flow and we do not find a functional flow, we know there exists a timing channel. If we find a functional flow we cannot conclude the existence of a timing channel.

Definition 14 (Functional flow)**.** *For a deterministic FSM F with input space X and output function α, we say that a* functional flow *exists with respect to a set of inputs $\{x_i\}_{i\in I}$ if there exists some value $k \in T$ and two input traces $A(X,k)$ and $A(X,k)'$ such that A and A' are value preserving with respect to I, and for $B := \alpha(A)$ and $B' := \alpha(A')$*

it is the case that there exists $e_i \in d(B)$ *and* $e_i' \in d(B')$ *such that* $\mathsf{val}(e_i) \neq \mathsf{val}(e_i')$.

This definition says that, if there is some functional flow from this set of inputs to the output, then there exist input traces of some size k that will demonstrate this flow; i.e., if a different output pattern is observed by changing only the values of these particular inputs, then their value does affect the value of the output and a functional flow must exist. In practice, however, this definition is not entirely useful: a system designer wanting to isolate timing flows by ensuring that no functional flows exist would have to look, for every possible value of k, at every pair of traces of size k in which the value of this set of inputs differs in some way; only if he found no such pair for any value of k would he be able to conclude that no functional flow exists. We therefore consider how to meaningfully alter this definition so as to still provide some guarantees (albeit weaker ones) about the existence of functional flows, without requiring an exhaustive search (over a potentially infinite space!).

Definition 15 (Functional flow)**.** *For a deterministic FSM* F *with input space* X *and output function* α*, we say that a* functional flow *exists with respect to a set of inputs* $\{x_i\}_{i \in I}$ *and an input trace* $A(X,k)$ *if there exists an input trace* $A(X,k)'$ *such that* A *and* A' *are value preserving with respect to* I *and for* $B := \alpha(A)$ *and* $B' := \alpha(A')$ *it is the case that there exists* $e_i \in d(B)$ *and* $e_i' \in d(B')$ *such that* $\mathsf{val}(e_i) \neq \mathsf{val}(e_i')$.

At first glance, this definition already seems much more useful: instead of looking just at the set of inputs, we also consider fixing the first trace. If we then construct our second trace given this first trace to ensure that the two are value preserving, then comparing the distinct traces of the output will tell us if a functional flow exists for the trace. Once again, however, we must consider what a system designer would have to do to ensure that no functional flow exists: given the first trace A, he would have to construct all possible traces A'; if the distinct traces of the outputs were the same for all such A',

then he could conclude that no functional flow existed with respect to A. Once again, this search space might be prohibitively large, so we consider one more meaningful weakening of the definition.

Definition 16 (Functional flow)**.** *For a deterministic FSM F with input space X and output function α, we say that a* functional flow *exists with respect to a set of inputs $\{x_i\}_{i\in I}$ and input traces $A(X,k)$ and $A(X,k)'$ that are value preserving with respect to I if for $B := \alpha(A)$ and $B' := \alpha(A')$ it is the case that there exists $e_i \in d(B)$ and $e_i' \in d(B')$ such that $\mathsf{val}(e_i) \neq \mathsf{val}(e_i')$.*

While this definition provides the weakest guarantees on the existence of a functional flow, it allows for the most efficient testing, as we need to pick only two traces. In addition, the guarantees of this definition are not as weak as they might seem: they say that, given the output B, by observing B' as well, we are not learning any additional information about the inputs $\{x_i\}_{i\in I}$ than we learned just from seeing B. Again, while this does not imply the complete lack of any functional flow, it does provide evidence in that direction (and running this procedure with more, carefully chosen pairs of traces would only strengthen that evidence).

Finally, we discuss our requirement that the system F be deterministic, and observe that it is not as strict as it might seem. As discussed at the beginning of the section, we are interested only in flows that are detectable by GLIFT. Physical processes that can be used to generate randomness, such as the current power supply or electromagnetic radiation, are therefore out of the scope of this work. We can nevertheless consider randomness, however, in the form of something like a linear feedback shift register (LFSR), which is in fact deterministic given its current state; the randomness produced by an LFSR can therefore be held constant between two traces by using the same initial state.

6.4.2 A sample usage: fast/slow multiplier

To build intuition for how our model determines whether or not a functional flow exists, we consider a simple system as shown in Figure 6.4.

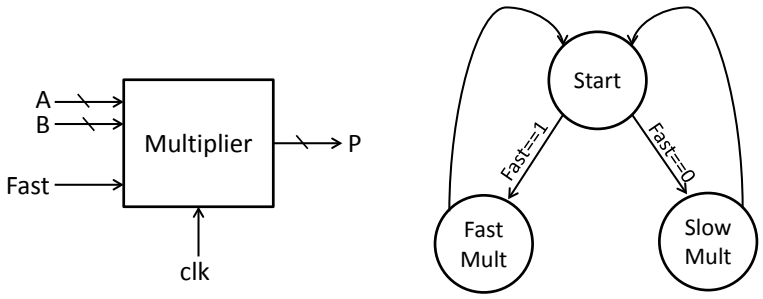

Figure 6.4. On the left, we can see the inputs and outputs of the system S: it takes in two multi-bit inputs A and B and two single-bit inputs, fast and a clock input clk, and outputs $P := A \times B$. On the right, we can see that the system first picks an ALU to use based on the value of fast and then uses that ALU to perform the multiplication.

As we can see, the system consists of a pair of two-input multipliers, one fast and one slow. On inputs A, B, and fast, the system will use fast to determine which of the hardware multipliers to use. For both A and B, there is a clear functional flow from the input to the output, as $P := A \times B$. The input fast, however, has no effect on the value of the output P, as it simply selects whether to perform a fast or slow multiply. There is therefore no functional flow from fast to the output, but there is a clear timing flow, as we can see that the latency with which P is computed is highly dependent on the value of fast.

To confirm this intuition that the flow from fast must be timing rather than functional, we look at this input through the lens of our technique described above. Using as F the system in Figure 6.4, we can define the input space to be $X := (\mathbb{Z}, \mathbb{Z}, \{0,1\})$; i.e., all tuples consisting of two integer values and one bit, and our output space to be $Y := \mathbb{Z}$. As mentioned, we are interested in whether or not a functional flow exists for fast, so we will define this to be our set of inputs. Now, we pick values A_0 and B_0 for

A and B respectively, and set our first trace to be $A := ((A_0, B_0, 0), t_0)$; i.e., the single event (at an arbitrary time t_0) in which A_0 and B_0 are multiplied using the slow ALU. We then set our second trace to be $A' := ((A_0, B_0, 1), t_0)$, and run these two traces to obtain output traces $B = (P, t)$ and $B' = (P', t')$. As A_0 and B_0 were the same for both traces, it is clearly the case that $P = P'$ and thus $\mathsf{val}(e_i) = \mathsf{val}(e'_i)$ for all $e_i \in d(B)$ and $e'_i \in d(B')$, meaning no functional flow exists with respect to these two traces. As discussed above, this also provides evidence that no functional flow exists for fast at all, although further testing would likely be required to rule out this functional flow completely.

Although this example is a bit contrived, it effectively shows that finding hardware timing channels in practice is non-trivial, and testing for them requires some intuition (for example, knowing which traces to pick). In the next section, we discuss a more complex example in which we examine how timing channels can be detected and eliminated in a shared bus system.

6.5 The Bus Covert Channel

Shared buses, such as the inter-integrated circuit (I^2C) protocol, universal serial bus (USB), and ARM's system-on-chip AMBA bus, lie at the core of modern embedded applications. Buses and their protocols allow different hardware components to communicate with each other. For example, they are often used to configure functionality or offload work to co-processors (GPUs, DSPs, FPGAs, etc.). As the hardware in embedded systems continues to become more complex, so do the bus architectures themselves, which makes it non-trivial to spot potential security weaknesses in their construction.

In terms of such security weaknesses, a global bus that connects `high` and `low` entities has inherent security problems such as denial-of-service attacks, in which a malicious device can starve one of higher integrity, and bus-snooping, in which a low device can learn information from a high one. It is not uncommon for designers to go as

far as building physically isolated `high` and `low` buses in high-assurance applications to avoid these vulnerabilties.

The covert channels associated with common buses are well researched. One such channel, the *bus-contention channel* [44] arises when two devices on a shared bus communicate covertly by modulating the amount of observable traffic on the bus. For example, if a device A wishes to send information covertly to a device B, it can generate excessive traffic on the bus to transmit a 1 and minimal traffic to transmit a 0. Even if A is not permitted to directly exchange information with B, it still may transmit bits of information using this type of covert channel.

The two most well-known solutions to the bus-contention channel are *clock fuzzing* [44] and *probabilisitic partitioning* [39]. The first, clock fuzzing, works by presenting a skewed and seemingly random input clock to the system. Such a "noisy" clock makes it stochastically difficult for the two devices to synchronize, and thus to perform this type of covert communication. Although often effective for specific applications, in reality clock fuzzing only reduces the bandwidth of the channel with an added performance overhead [39]. Probabilistic partioning, on the other hand, works by permitting devices access to the bus in isolated time slots in a round-robin fashion. Two modes are chosen at random: secure and insecure. In insecure mode, the bus operates in the standard fashion where devices contend for its usage. In secure mode, the bus is allocated to each device in a time-multiplexed round-robin manner; this therefore limits access the bus to only one device at a time, thus eliminating the potential for covert communication.

Both clock fuzzing and probabilistic partitioning have proven to be effective at reducing, if not eliminating, the bus-contention channel. They do not, however, expand beyond this particular channel and explore whether or not information might leak through other timing channels associated with the bus architecture. In addition, previous work

using GLIFT has shown that strict information flow isolation can be obtained in a shared bus [78], but the work states nothing about how this information relates to timing. In what follows, we demonstrate how to use GLIFT and the techniques presented in Section 6.4 to prove that certain information flows in I^2C occur through timing channels.

6.5.1 Identifying Timing Flows in I^2C

The inter-integrated circuit (I^2C) protocol is a simple 2-wire bus protocol first proposed by Philips [5]. We chose to look specifically at I^2C because of both its wide usage in embedded applications for configuring peripherals and its simple structure; there is no reason, however, why the techniques presented here could not be applied to more sophisticated architectures or protocols.

In the I^2C protocol (seen in Figure 6.5), a "master" of the bus initiates a transaction by first sending a *start* bit by pulling down the data line (SDA) with the clock line (SCL) high. "Slaves" on the bus then listen for the master to indicate either a read or a write transaction. For write transactions, the master first sends a device address indicating a write and the device that matches this address responds with an acknowledgement (ACK). At this point, the master can transmit an internal register address (sub-address for the device) and the actual data. The transaction terminates with the master sending a *stop* bit. A similar behavior occurs for a read transaction, except here data transfers from a slave to the master. Since I^2C shares a common bus, there is the potential for several different covert channels, in addition to the bus-contention channel described above. To explore these different channels, we look at three configurations of the I^2C bus and discuss the potential ways in which information can be communicated covertly. We furthermore discuss how the flows in each of these covert communications can be classified as either a functional or timing flow using the techniques presented in Section 6.4.

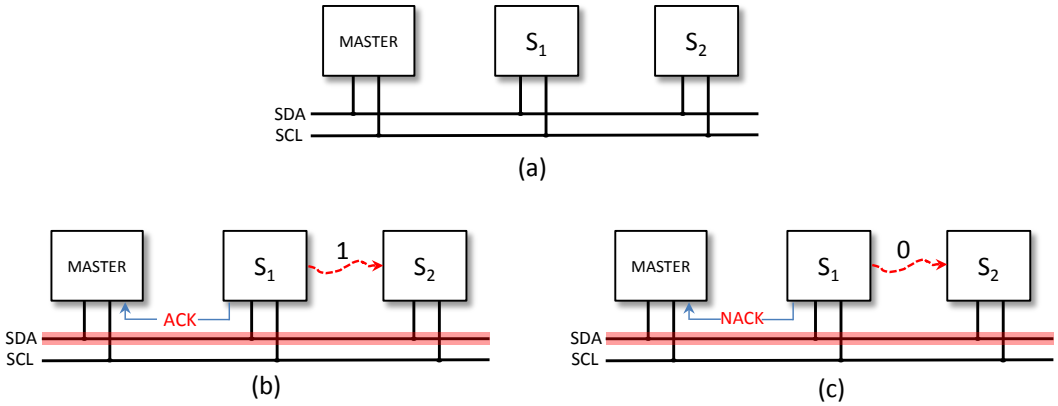

Figure 6.5. (a) Standard I^{2}C configuration. (b) S_1 can covertly communicate a 1 to S_2 by sending an acknowledgement. (c) S_1 can communicate a 0 covertly to S_2 by sending a negative-acknowledgement.

Case 1: global bus

A global bus scenario, wherein multiple devices contend for a single bus, is the most general and commonly found bus configuration. Consider the example in which two devices wish to communicate covertly on the I^{2}C bus as shown in Figure 6.5. At first glance, there exists an obvious information flow in this architecture since the devices themselves can "snoop" the bus. For example, a device S_1 can send an acknowledgement to the master to covertly transmit a 1 to another device S_2; conversely, it can send a negative-acknowledgement to send a 0. Since S_2 observes all activity on the bus, it can simply monitor which type of message S_1 sends and thus determine the communicated bit. While this is by no means the only type of flow, for the sake of simplicity we will stick with this scenario throughout the rest of the section.

To put our model to use on this scenario, we designed the system shown in Figure 6.5 in Verilog by constructing I^{2}C Master and Slave controllers. Since we were interested in the flows between S_1 and S_2, we processed the designs in the manner presented in Section 6.3 and in the previous work. To be concrete, we took the slave and master RTL descriptions and synthesized them down to logic gates using Synopsys'

Design Compiler. For each gate primitive in the system, we added the appropriate GLIFT logic. The result is a system which contains a master and two slaves, each of which also has tracking logic associated with it. In a manner similar to that of previous work, we executed a test scenario wherein the master performs a write transaction with S_1 and S_1 sends an acknowledgement by simulating it in ModelSim 10.0a [69], a Verilog simulator. We observed that the GLIFT logic indicates a flow to S_2. At this stage, we have therefore identified that some type of information flow exists, but it is not entirely obvious if this was a functional or timing flow.

Since the devices can directly observe all interactions on the bus, one might expect this to be a functional flow. Not surprisingly, we utilized the model presented in Section 6.4 to show exactly that. To put this model to use, we abstract the output $y = \langle \text{SCL}, \text{SDA} \rangle$ of our model since these are the only two signals observable by S_2 (recall that SCL is the clock line and SDA the data line). In addition, we abstracted the input traces to our system as $A_1(X,k) := \langle S_1 \text{ sending NACK} \rangle$ and $A_2(X,k) := \langle S_1 \text{ sending ACK} \rangle$; running these through the system produced two output traces A_{G_1} and A_{G_2}. In a bit more detail, we collected A_{G_1} by logging the discrete events that occured when S_1 failed to acknowledge a write transaction from the master (thus intending to covertly transmit a 0). We then obtained a related trace A_{G_2}, in which S_1 does acknowledge the write. By analyzing these traces, we identified events $e_j \in d(A_{G_1})$ and $e'_j \in d(A_{G_2})$ (recall that $d(A_{G_1})$ and $d(A_{G_2})$ are the *distinct* traces of A_{G_1} and A_{G_2} respectively, as defined in Definition 6) such that $\mathsf{val}(e_j) \neq \mathsf{val}(e'_j)$. As a result, from Definition 16 of a functional flow, we know that a functional flow must exist. Recall, however, that this does not mean that there exists *only* a functional flow. Since GLIFT indicates that there exists a flow, it may be the case that information flows from S_1 to S_2 through both functional and timing channels.

The next case discusses how such a functional flow can be easily prevented using

time-multiplexing of the bus in a manner similar to probabilistic partitioning [39].

Case 2: strict time-multiplexing of the bus

A seemingly easy solution to eliminate this information flow presented in Case 1 is to add strict partitioning between when devices may access the bus, as shown in Figure 6.6. Here, slaves on the bus may view the bus only within their designated time slots; this prevents devices from observing the bus traffic at all times. In this work, we partition over-conservatively by allowing the bus to be multiplexed between statically set time slots. In terms of probabilistic partitioning, we test the case in which the system is running in secure mode. We are interested in the same scenario as before: S_1 wishes to transmit information covertly with S_2; now, however, the bus-contention channel is eliminated, as partitioning has made contention impossible.

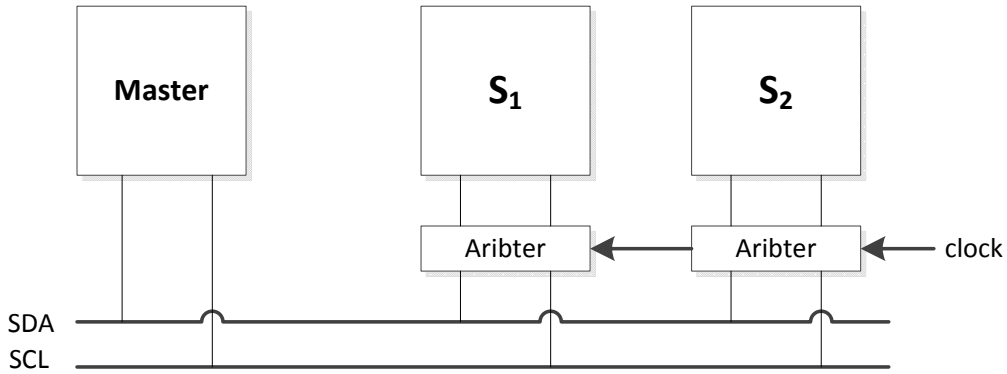

Figure 6.6. Adding strict time-partitioning of the I^2C bus. The bus is only accessed by S_1 and S_2 in mutually-exclusive time slots.

Because the bus-contention channel has been ruled out, one might think that a covert channel between S_1 and S_2 no longer exists. Nevertheless, information can still be communicated covertly through the internal state of the master; to therefore transmit a covert bit, S_1 need only leave the master in a particular state before its time slot expires. For example, many bus protocols have a time-out period in case a device fails to respond

to a request. If S_1 leaves the master in such a state prior to its time-slot expiring, S_2 can observe this state in the following time slot and conclude, based on the response time from the master, whether a 0 or a 1 is being transmitted: if the master's response time is short, S_2 can conclude S_1 wishes to communicate a 1, and if the response time is long it can conclude a 0. Although this type of covert channel is quite subtle, by using the model from Section 6.4 we can prove that this information flow occurs through a timing channel.

To make use of our model, we again synthesize the Verilog master, slave, and arbiter (as shown in Figure 6.6) into gates and annotate the design with GLIFT logic exactly as we did in Case 1. We then executed the same scenario as Case 1 by having the master request a write to S_1 during S_1's allocated time slot and having S_1 either acknowledge or not to covertly transmit a 1 or 0 respectively. After tainting the data out of S_1, the GLIFT logic indicated that there is indeed information flowing from S_1 to S_2. As S_2 can access the bus only after S_1's time slot has expired, this flow must occur through the state of the master.

To prove that this is not a functional flow, we abstract this system in the same manner as Case 1, except we now use $y = \langle SDA_{S_2}, SCL_{S_2} \rangle$, where SDA_{S_2} and SCL_{S_2} are the wires observable by S_2. In the same manner as Case 1, we set input traces $A_1(X,k) := \langle S_1 \text{ sending NACK} \rangle$ and $A_2(X,k) := \langle S_1 \text{ sending ACK} \rangle$ to collect output traces A_{TDMA_1} and A_{TDMA_2} respectively. Following our model, we worked to find the existence of an event $e_j \in d(A_{TDMA_1})$ and $e'_j \in d(A_{TDMA_2})$ such that $\mathsf{val}(e_j) \neq \mathsf{val}(e'_j)$; we found, however, that no such events existed for this particular testing scenario. As discussed in Section 6.4, this provides evidence for the absence of a functional flow; although it does not completely rule out the existence of such a flow, because we have chosen our input traces to represent essentially opposite events (sending a negative-acknowledgement and sending an acknowledgement), if a functional flow did exist then

it is very likely it would be captured by these two traces. We therefore conclude that, because GLIFT did indicate the existence of some information flow and we have provided strong evidence that a functional flow does not exist, this flow is from a timing-channel.

Case 3: time-multiplexing with master reset

The work of Oberg et al. [78] using GLIFT for the I^2C channel indicated that all information flows are eliminated when the master device is reset back to a known state on the expiration of a slave's timeslot. In particular, this implies that no timing channels can exist, and thus the attack from Case 2 no longer applies. In practice, this trusted reset would need to come from a trusted entity such as a secure microkernel; we will therefore assume for our testing purposes that this reset comes from a reliable source once this subsystem is integrated into a larger system. With this assumption, we validated this scenario by adapting the test setup in Case 2 to incorporate the master being restored to an initial known state once S_1's time slot expires.

We again took the slave, master, and arbiter Verilog modules, synthesized them into logic gates, and applied the GLIFT tracking logic just as we did in Case 2. Running this test scenario, we observed that, as previous work had indicated, GLIFT shows that there is no information flowing from S_1 to S_2. At this point, we could conclude that no information flow exists (either functional or timing), but for the sake of completeness we again used our model to test the existence of a functional flow for this test case.

In the same manner as Case 2, we abstract the output $y = \langle SDA_{S_2}, SCL_{S_2} \rangle$. We create input traces $A_1(X,k) := \langle S_1 \text{ sending NACK} \rangle$ and $A_2(X,k) := \langle S_1 \text{ sending ACK} \rangle$ to log output traces A_{TDMA_1} and A_{TDMA_2} respectively. As expected, $d(A_{TDMA_1} = d(A_{TDMA_2})$, and thus we again obtain strong evidence that a functional flow does not exist.

As is hopefully demonstrated by these three cases, identifying the presented

covert channels is not necessarily intuitive; furthermore, hardware designers are likely to easily overlook these problems when building their bus architectures or designing secure protocols. By combining the tracking logic of GLIFT with our model, we provide a method for hardware engineers to systematically evaluate their designs to determine whether or not techniques such as those used in Case 3 can in fact eliminate covert channels such as the ones presented in Case 1 and Case 2.

6.5.2 Overheads

To provide an understanding of the associated overheads with these techniques, we present the simulation times needed to execute them. We collected the simulation times by using ModelSim 10.0a [69] and its built-in `time` function. The simulations were run on a machine running Windows 7 64-bit Professional with an Intel Core2 Quad CPU(Q9400) @ 2.66GHz and 4.0GB memory.

Table 6.1. Simulation times in milliseconds associated with the three presented cases for I^2C, and for a single trace. GLIFT imposes a small overhead in the simulation time for these test cases.

	Case 1	Case 2	Case 3
GLIFT	223.95 ms	230.29 ms	222.40 ms
RTL	210.45 ms	211.72 ms	219.04 ms

As seen in Table 6.1, there is not a significant difference between simulating the designs with GLIFT logic and the base register-transfer level (RTL) designs. This is likely due to the small size of the designs and the relatively short input traces required for these particular tests. The overheads associated with GLIFT become more apparent in Section 6.6 when we discuss identifying timing channels associated with a CPU cache.

Finally, we mention that, although we consider two input traces for each case, we present in Table 6.1 our simulation times for only a single input trace. We do this because, as mentioned in Section 6.4, designers may wish to check even beyond two traces to

gain more assurance that a functional flow does not exist. Since the simulation time of a particular input trace is independent of the others, we chose to present the results for a single trace but note that they can be appropriately scaled to consider more traces as well.

6.6 Cache Timing Channel

Recent work has shown CPU caches to be one of the biggest sources of hardware timing channels in modern processors [10, 18, 86, 40]. In a modern computing system, a cache can be seen as a performance optimization that provides a "quick look-up" for frequently used information. Caches are typically built from faster and higher power memory technologies, such as SRAM, and sit between slower main memory (typically DRAM) and the CPU core. When a memory region is referenced by a program, it is brought into the cache for fast access.

Broadly, there are two basic types of caches: *direct-mapped* and *set-associative*. For a direct-mapped cache, a memory address A is divided into three fields: *tag*, *index*, and *disp*, where *tag* is meta-data used when indexing into the cache, *index* is the line index, and *disp* is the block offset. When accessing a direct-mapped cache, *index* is first checked to determine which cache line A references. Next, *tag* is checked; if *tag* matches what is stored in the cache, a *hit* occurs and the data is accessed with low latency from the cache. If it does not match, a longer latency *miss* requires that the data be accessed from main memory and brought into the cache. In an effort to minimize misses, N-way set-associative caches allow N cache lines to exist in a set. In the same manner as a direct-mapped cache, the cache is first accessed using *index*. Then, within the index, each of the N lines' *tag* entry are checked against the *tag* of A. If none match then a miss occurs, which requires one of the N cache lines to be evicted and replaced by referenced data using a replacement policy such as least-recently used (LRU).

Since most programs have spatial locality in their memory reads and writes, the

presence of a cache makes the overall performance significantly higher than if main memory were accessed on every memory operation. However, caches' non-deterministic latencies are the direct source of timing channels. As mentioned, when a memory region is referenced that is currently stored in the cache (a cache hit), the time to receive the data is significantly faster than if it needs to be retrieved from main memory (a cache miss). Many data encryption algorithms, such as the advanced encryption standard (AES), use look-up tables based on the value of the secret key. Since a look-up table will return a value in an amount of time that is directly correlated with whether or not the value is already cached, observing the timing of interactions with the look-up table produces valuable information about the secret key.

In previous work, this vulnerability has been used to completely extract the secret key; these attacks have been divided into three categories: trace-driven [10], time-driven [18, 86], and access-driven [40]. Briefly, trace-driven attacks require that an adversary have detailed cache profiling information such as the number of cache hits and misses. This means that the adversary requires either physical access to the machine or other means for obtaining these fine granularity details. Time-driven attacks instead collect timing measurements over several encryptions by a remote server and correlate their running time to the value of the secret key; this type of attack was carried out most notably by Bernstein [18], who used it to extract a complete 128-bit AES key. To do this, Bernstein exploits timing fluctuations of $\approx 2^{22}$ samples and uses stochastic differences between encryptions to identify the value of a byte of the secret key. He then repeats this procedure for each of the key bytes to eventually extract the key in its entirety. Although this type of attack is the most general, it requires large numbers of encryptions ($\approx 2^{26}$) in order to obtain enough data to make this attack practical. Finally, access-driven attacks exploit knowledge about which cache lines are evicted. In particular, a malicious process observes the latency of cache misses and hits and uses these patterns to deduce which

cache lines are brought in/evicted, which in turn leaks information about the memory address (e.g., the secret key in AES table look-ups). In this work, we chose to look at access-driven attacks, as they are the easiest for us to demonstrate given our current test setup. Furthermore, this type of cache attacks has applications beyond just encryption; for example, as demonstrated by Ristenpart et al. [91] in their attack on virtualized systems.

6.6.1 Overview of Access-Driven Timing Attacks

At a high-level, an access-driven cache timing attack first flushes the cache using some malicious process. Next, a secret process uses a secret key to perform encryption. Finally, the malicious process tries to determine which of the cache lines were brought into the cache in the encryption process. Since the key is XORed with part of the plaintext before indexing into a look-up table, the malicious process can correlate fast accesses with the value of the secret key. As noted by Gullasch et al. [40], this attack assumes that the secret and malicious process share physical memory; an attack in which the secret and malicious process do not share physical memory would require slightly different behavior from the malicious process.

In a bit more detail, we can see a depiction of this attack in Figure 6.7. Assume we have a malicious process M and secret process V (for victim). First, as seen in part (a), M flushes all contents of the cache. Next, as seen in part (b), V subsequently runs AES using a secret key as input for a short duration; this process fills the contents of the cache. Now, in part (c), M reads memory locations and observes the latency of each access. Since M and V share physical memory, M will receive memory responses with lower latency if V accessed this data prior to the context switch, as it will already reside in the cache. Because the secret key used by V is an index into look-up tables, the access latencies of M (i.e., a cache hit or miss) directly correlate with the value of the secret key.

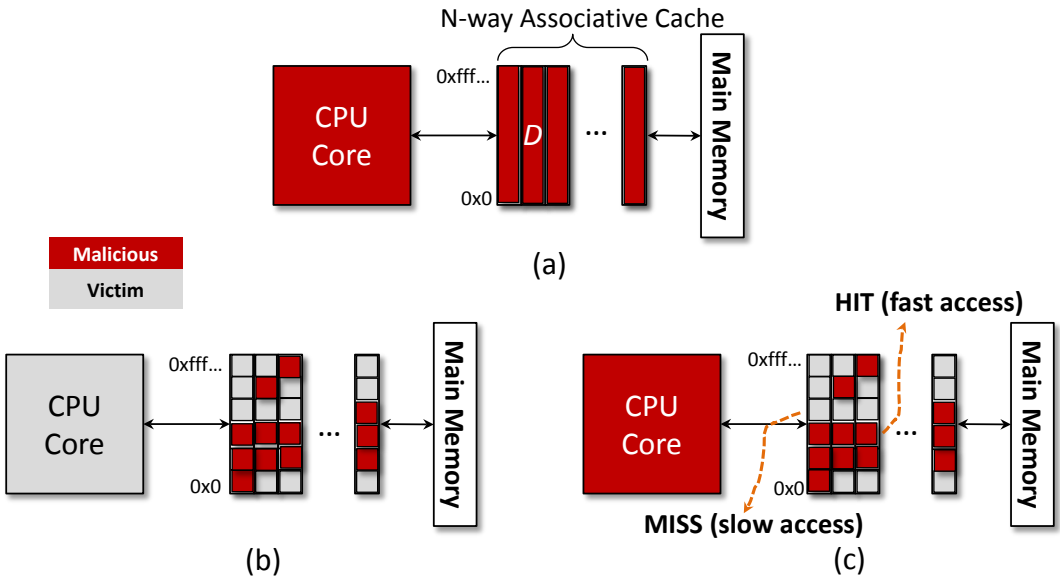

Figure 6.7. (a) A typical CPU cache. The attack operates by malicious process first flushing the cache. (b) The victim process encrypts some data with its secret key, thus bringing in cache lines. (c) The malicious process can observe which cache lines are present from latency, thus deducing the address and value of key used to index look-up table.

6.6.2 Identifying the Cache Attack as a Timing Channel

As the above attack critically requires on the timing information available to *M*, it can clearly be identified as a type of timing attack. In this section, we demonstrate this fact more formally by using GLIFT and our model from Section 6.4 to prove that any information flows are timing-based.

To put this scenario to test, we designed a complete MIPS based processor written in Verilog. The processor is capable of running several of the SPEC 2006 [43] benchmarks including `mcf`, `specrand`, and `bzip2`, in addition to two security benchmarks: `sha` and `aes`, all of which are executed on the processor being simulated in ModelSim SE 10.0a (a commercial HDL simulator) [69]. All benchmarks are cross-compiled to the MIPS assembly using gcc and loaded into instruction memory using a Verilog testbench. The architecture of the processor consists of a 5-stage pipeline and 16K-entry direct mapped

cache (1-way cache). We chose to use a direct-mapped cache for our experiments for ease of testing, but note that this analysis would apply directly to a cache with greater associativity.

Since our particular region of interest is the cache, we focus our analysis directly on this subsystem. To do so, we apply GLIFT logic to the cache system as described in section 6.3. Specifically, we remove the hardware modules associated with the cache (cache control logic and the memory itself) and synthesize them to logic gates and flip-flops using Synopsys' Design Compiler using their `and_or.db` library; this library contains basic 2-input ANDs, ORs, inverters and flip-flops, and thus our resulting design contains only these primitives. We then process each gate and flip-flop in the design and add its associated tracking logic in a compositional manner. That is, we process each gate and flip-flop linearly and add the GLIFT logic associated with their function (AND, OR, etc.). This new "GLIFTed" cache is re-inserted into the register-transfer level (RTL) processor design in the place of the original RTL cache. Pictorially, this can be seen in Figure 6.8. The input and output to the cache system include address and data lines and control signals (write-enable, memory stall signals, etc.); each such input and output is now associated with a taint bit which will be essential to testing whether or not information flows from our victim process V to our malicious process M.

To execute the test scenario, we follow the same procedure as the access-driven timing attack previously discussed by having malicious and victim executions share the cache. We have M first flush the cache by resetting all data in the cache. We then have V execute AES with all inputs to the cache marked as tainted (i.e. secret). Subsequently, we have M execute and observe whether or not information from V flows to M. As expected, we observe that as M reads from memory locations, secret information immediately flows out of the cache. We therefore know that a flow exists, but at this stage it is still ambiguous whether the flow is functional or timing.

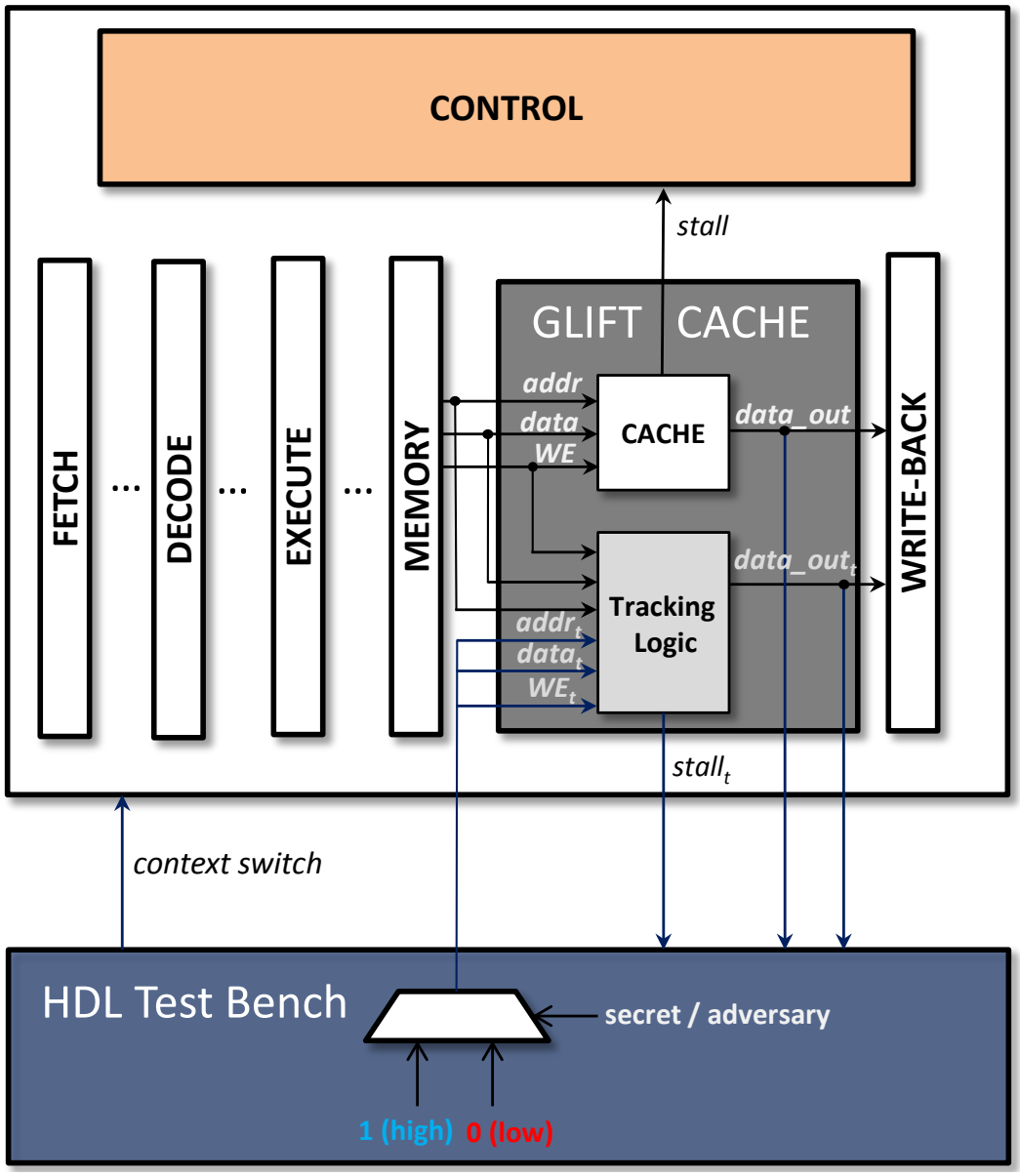

Figure 6.8. A block diagram of a simple MIPS-based CPU. The cache is replaced by one which contains the original cache and its associated tracking logic. Our testbench drives the simulation of the processor to capture the output traces.

To identify exactly which type of channel was identified by GLIFT, we leverage the benefits of our model by working to identify a functional flow; as previously discussed, if we detect no functional flow, then we know the flow must be from a timing channel. To fit our model, we abstract the output of the cache as $y = \langle data_M \rangle$ to indicate the cache output observable by M (note that, in particular, *stall* is not included in this output, as it cannot be observed directly by M). Following our model, we then defined two traces: $A_1(X,k) := \langle V \text{ using } K_1 \rangle$ and $A_2(X,k) := \langle V \text{ using } K_2 \rangle$; i.e., the cases in which V encrypts using two different keys. We then simulated both of these scenarios and logged all of the discrete events captured by ModelSim to obtain to output traces A_{C1} and A_{C2}; by definition of y, these output traces contain all events observable by M. Once we collected these traces, we checked whether or not a functional flow exists for these particular traces by looking for the existence of events $e_j \in d(A_{C1})$ and $e'_j \in d(A_{C2})$ such that $\mathsf{val}(e_j) \neq \mathsf{val}(e'_j)$. For these particular traces, we found no such pair of events. Again, although the fact that no functional flow exists with respect to these particular traces does not imply the lack of a functional flow for any traces, it does lend evidence to the theory that the flow must be timing-based rather than functional (and additional testing with different keys would provide further support).

6.6.3 Overheads

As we did for I^2C in Section 6.5, we evaluated the overheads associated with our technique by measuring simulation time. We collected our measurements using ModelSim 10.0a and its built in `time` function running on the same Windows 7 64-bit Professional machine with an Intel Core2 Quad CPU(Q9400) @ 2.66GHz and 4.0GB of memory. We measured the time for the secret process (V) to run AES on a secret key K_1 followed by a malicious process (M) attempting to observe which cache lines were evicted. This measurement was repeated for both the design with and without GLIFT.

For completeness, we repeated the same process for the second input traces; namely when V executes AES using K_2 followed by M attempting to observe which cache lines were evicted. The resulting times from these simulations can be found in Table 6.2.

Table 6.2. Simulation times in seconds for AES running with different encryption keys, with and without GLIFT tracking logic. In general, simulating a design with GLIFT logic causes large slow-downs.

	AES_{K_1}	AES_{K_2}
GLIFT	381.49 s	392.60 s
RTL	66.30 s	66.76 s

As Table 6.2 shows, there is a large overhead ($\approx 6X$) for using GLIFT to detect whether or not a flow exists. Furthermore, since the behavior of M is fixed between both input traces and the only value changing is the secret key, the results clearly show that a timing channel exists with regards to the cache, as the execution time for AES on K_2 is longer than that of K_1; the existence of such a timing channel was also identified by GLIFT and our model.

6.7 Timing Channels in RSA Encryption Core

As an additional point of reference, this section describes how this model can be applied to detect a timing channel in an RSA cryptographic core. The RSA public-key cryptosystem [92] is one of the most widely used data encryption and digital signtuare algorithms. In short, the algorithm uses modular exponentiation to encrypt and decrypt data. Computing this exponentiation can be done quickly and efficiently in hardware.

One approach for computing decryption: $C^d \pmod{n}$, where C is the ciphertext, d is the private key and n the public modulus, is to employ a square-and-multiply algorithm which iterates over all key bits and performs a multiply each iteration depending on the value of the key bit. The details of this algorithm can be seen in Algorithm 2 (note

that this is the same algorithm presented in Chapter 2.3). If the current key-bit is 1, a multiply is performed otherwise the operation is skipped. A square is computed every iteration.

Algorithm 2. Basic algorithm for square-and-multiply to compute modular exponentiation. It computes $C^d \pmod{n}$.

```
R = 1;
temp = C;
for i = 0 to |d| − 1 do
  if bit d[i] = 1 then
    R = R · temp (mod n)
  end if
  temp = temp^2 (mod n)
end for
return R
```

As one might expect, on iterations where an additional multiply is performed, the run time will be slower. Essentially, the value of the key will have great influence on the run-time of the decryption and thus attackers can (and have [55]) exploited this timing variation to extract the private key.

In hardware, an RSA decryption module[1] will not only have Key and Ciphertext inputs and a Message output, but other control signals as well. For example, a signal is needed to notify when the algorithm should begin (*start*) and also an output to say when the decryption is completed (*rdy*). If the key affects when the Message is ready (i.e. the time in which *rdy* is asserted), this timing variation can be exploited by an attacker.

6.7.1 Detecting Leak as Timing Channel

To this end, we apply our analysis to the BasicRSA core from opencores [82] and determine whether or not there is a timing channel in the design. Following the same GLIFT analysis flow, we detect that the key does in fact affect *rdy*. Now, to classify this

[1] We use decryption here because RSA decrypts using a private key and encrypts with a public one

as a timing leak, we apply the model presented in Chapter 6.4 and abstract the input traces $A := \langle$RSA on Key 1$\rangle$ and $A' := \langle$RSA on Key 2$\rangle$ using two randomly chosen keys and record the output traces B and B' by logging the values of the *rdy* signal for the duration of the decryption. When applying our model to these output traces, we find that $val(e_i) = val(e'_i)$ for all $e_i \in d(B)$ and $e'_i \in d(B')$. Since GLIFT indicates that there is an information leak and we did not detect a functional flow, we know that this leak must be from a timing-channel.

The analysis of this core brings up a necessary discussion. As described, our model cannot detect the presence of a timing when a functional one exists as well. For example, the ciphertext of an encryption algorithm (like RSA) will always be functionally affected by the key. However, as demonstrated here, by discovering the key's effect on the time in which *rdy* is asserted, it is possible to conclude that it affects the time in which the cryptographic process completes. In other words, this technique is able to conclude that the core has a timing channel.

Chapter 6, in full, is a reprint of the material as it appears in the conference on Design Automation and Test in Europe 2013 and also in submission at the IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems 2014. Oberg, Jason; Meiklejohn, Sarah; Sherwood, Timothy; Kastner, Ryan. The dissertation author was the primary investigator and author on both of these papers.

Chapter 7

Conclusion

The number of computers we rely for our personal health, safety, and security is ever increasing. Many of these computers we completely take for granted and put complete trust that they will behave in the manner promised to us. Unfortunately, we have already encountered many serious problems in automobiles [57, 100] and medical devices [41, 75] and it seems likely that further attacks and vulnerabilities will be exposed in the near future. With the increasing complexity of hardware, design evaulation is becoming an increasingly important problem [50]. Security can no longer be a second-class concern and needs to be managed as a primary design constraint in the development of our hardware.

As described in this thesis, hardware itself is becoming more frequently exploited by attackers. Side-channel attacks in the form of both power [54, 66, 70, 71, 85] and timing [22, 10, 86, 18, 40, 12, 9, 11] are becoming more prevelant making it easier for attackers to thwart the mathematical power of cryptogrphic algorithms. Timing-attacks are often powerful because they often do not require physical access to a device. In many of these attacks, such as the one performed by Bernstein [18], was done on a remote machine. In order to appropriately manage these timing-based information leaks, hardware designs need appropriate tools to help them evaluate and debug the security of their designs.

In the past chapters in this thesis, I presented a technique for detecting timing-based information flows in hardware. This technique leverages past work on gate-level information flow tracking [103] and demonstrates that it can be nicely applied to hardware security testing and verification. Unlike prior work on GLIFT, the concepts presented here demonstrate that GLIFT can in fact be used as a powerful, static, testing technique. To demonstrate its effectiveness, I have showed several different application use cases for using GLIFT statically. In Chapter 4, I demonstrated that GLIFT can be an invaluable tool for analyzing different information flows in common bus protocols. Although I specifically targetted I^2C and USB, the ideas presented in this chapter extend easily to others. Further, to show that this technique can be applied to larger systems, I built a system-on-chip (SoC) that was composed of two processors sharing a hardware accelerator (an AES core) as discussed in Chapter 5. Here I demonstrated that only the interfaces need to be analyze since we were concerned with the interaction of the two processors. Using this analysis, I was able to demonstrate that the processors could not affect one another even through timing. Lastly, in Chapter 6, I formalized exactly how GLIFT detects timing channels and also presented a formal model for separating these timing leaks from more functional ones. All of these contributions demonstrate that GLIFT is well suited for static testing.

The static testing techniques and detection of timing channels presented in this thesis will provide valuable insight for security testing and verification in next generation hardware. Security can no longer be overlooked be hardware designers. Companies have already begun this push by putting together hardware security teams focusing primarily on potential vulnerabilities in their designs. Intel's Security Center of Excellence (SeCoE) is a good example of where the hardware design community is moving. Although companies are apparently moving in the right direction, the tools and methods for assisting in this movement are far behind. The methods I presented in this thesis provide

such a set of techniques which will aid in the development of our future hardware in a more secure manner. The hope is that these and similar techniques will be adopted in order for our embedded systems, that manage many important aspects of our lives, to be more trustworthy, safe, and reliable.

Chapter 8

Future Research in Hardware Security

There are many avenues for future research using GLIFT. There are many other necessary formalisms related to the preciseness (see [46, 47] for details of preciseness) of the GLIFT logic and timing channels. For example, I have observed behavior that the GLIFT logic will detect an information leak for finite amounts of time. The duration in which GLIFT asserts there is a flow is both dependent on the timing channel and the preciseness of the generated tracking logic. In the future, I will explore these properties further. The hope in studying this specific topic more is that may be possible to measure the bandwidth of a timing channel. It also might be possible to profile different types of inputs on how much information they are leaking. For example, it may be the case that different values of secret keys actually leak more information than others. I believe GLIFT has the power to explore some of these very important and intricate security properties.

Another avenue for further development is to use GLIFT to make much more formal guarantees and truly prove hardware security properties. For example, most formal tools, such as those in the open-source ABC tool developed at Berkeley [7], are based on formal solvers such as SAT and SMT solvers. The Boolean satisfiability problem (SAT) is computationally difficult (*NP*-hard). Due to this inherent difficulty, extremely complex problems, such as providing static and provable security properties,

can be decided using SAT-solvers. Even further, the efficiency of these SAT solvers is increasing dramatically with increase computing power and further research. As an example, in the annual SAT competition, the run time to solve 80 problems was 1000 seconds in 2002 and 40 seconds in 2010 [2]. This makes proving hardware security properties much more achievable than in the past. These SAT solvers require that the problem be expressed as a formal logic. The solvers then work to determine whethe or not this logic can ever be satisfiable, or true. Since digital hardware is composed of Boolean functions separated by stateful latches, SAT solvers are a very natural fit to GLIFT. The mapping between our information flow security properties to a formal logic happens neatly since it already exists as one. A future research direction would be to make this formal integration a reality by coupling our gate-level logic with a formal SAT-based tool using ABC. This would provide a seemless integration so that hardware designers wishing to prove security properties about their RTL can easily pass in a set of security properties and the RTL and receive output about whether or not they passed or failed.

Beyond all the work I have done with my collaborators related to GLIFT [77, 78, 46, 47, 102, 80, 79, 45, 109, 48] I have contributed to the development of two secure hardware description languages. The Caisson language, which was presented in the symposium on Programming Language Design and Implementation in 2011 [64], leverages a static type system to allow hardware designers to formally prove non-interference about their register transfer level (RTL) code. My contribution to the Caisson language was assessing the overhead that this new language caused on the resulting hardware and an invaluable comparison to a system with GLIFT logic. Further, in a much more improved language called Sapper, which will appear at the conference on Architectural Support for Programming Languages and Operating Systems in 2014 [63], I designed and built the processor used for the performance assessment of the language. Sapper

improved the hardware overhead imposed by Caisson by generating concise logic for dynamic information flow tracking. In other words, security properties were enforced in the physical hardware rather than at design time.

I have a huge interest in hardware description languages for both security and easier design. Current languages for hardware design are not only difficult for designers to write, but they do not offer good methods for security. Ideally, assertions, such as System Verilog Assertions (SVAs), would have the ability to concisely specify information flow security properties. There are large avenues for future research in this space and I hope to leverage my past experience in this space to explore it more in the future.

Bibliography

[1] Common criteria for information technology security evaluation. http://www.commoncriteriaportal.org/cc/.

[2] International sat competition. http://www.satcompetition.org/.

[3] Iwls 2005 benchmarks. http://iwls.org/iwls2005/benchmarks.html.

[4] Usb 2.0 specification, April 2000. http://www.usb.org/developers/docs.

[5] I2c manual. http://www.nxp.com/documents/ application_note/AN10216.pdf, March 2003.

[6] What does cc eal6+ mean?, November 2008. http://www.ok-labs.com/blog/entry/what-does-cc-eal6-mean/.

[7] Berkeley logic synthesis and verification group, abc: A system for sequential synthesis and verification, 2013. http://www.eecs.berkeley.edu/ alanmi/abc/.

[8] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005.

[9] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'10, pages 110–124. Springer-Verlag, 2010.

[10] Onur Aciiçmez and Çetin Kaya Koç. Trace-driven cache attacks on aes (short paper). In *ICICS*, pages 112–121, 2006.

[11] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' track at the RSA conference on Topics in Cryptology*, pages 225–242, 2006.

[12] Onur Aciiçmez and Werner Schindler. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'08, pages 256–273. Springer-Verlag, 2008.

[13] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The em side-channel(s). In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, pages 29–45, 2003.

[14] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 297–307. ACM, 2010.

[15] D. Asonov and R. Agrawal. Keyboard acoustic emanations. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 3–11, May 2004.

[16] J. Barnes. *High integrity software: The SPARK approach to safety and security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[17] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. *Technical report, Technical Report MTR-2547*, 1973.

[18] Daniel J. Bernstein. Cache-timing attacks on aes. technical report. technical report, 2005.

[19] K. J. Biba. Integrity considerations for secure computer systems. *MITRE Technical Report TR-3153*, 1977.

[20] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystems. In *Advances in CryptologyCRYPTO 2000*, pages 131–146. Springer, 2000.

[21] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology, CRYPTO'97*, pages 513–525. Springer, 1997.

[22] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. cryptographic hardware and embedded systems. In *Lecture Notes in Computer Science series 4249*, pages 201–215. Springer, 2006.

[23] Haibo Chen, Xi Wu, Liwei Yuan, Binyu Zang, Pen chung Yew, and Frederic T. Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. *isca*, 0:401–412, 2008.

[24] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 50–62. ACM, 2009.

[25] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[26] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium*, January 1998.

[27] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO 2004*, pages 221–232, 2004.

[28] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ISCA 2007*, pages 482–493, 2007.

[29] Ivan Bjerre Damgård. A design principle for hash functions. In *Advances in CryptologyCRYPTO89 Proceedings*, pages 416–427. Springer, 1990.

[30] Dorthy E. Denning. A lattice model of secure information flow. *Communications of ACM*, 19(5):236–243, May 1976.

[31] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processing in the asbestos operating system. In *In Proceedings of the Symposium on Operating System Principles (SOSP)*, 2005.

[32] ESCRYPT. Etas plans acquisition of system house escrypt, May 2012. https://www.escrypt.com/company/single-news/detail/etas-plans-acquisition-of-system-house-escrypt.

[33] Federal Aviation Administration (FAA). Boeing model 787-8 airplane; systems and data networks security isolation or protection from unauthorized passenger domain systems access. http://cryptome.info/faa010208.htm.

[34] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting General Security Attacks. In *Annual IEEE/ACM International Symposium on Microarchitecture*, December 2006.

[35] L.J. Fraim. Scomp: A solution to the multilevel security problem. *Computer*, 16(7):26–34, July 1983.

[36] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. Cryptology ePrint Archive, Report 2013/857, 2013. http://eprint.iacr.org/.

[37] E. L. Glaser, J. F. Couleur, and G. A. Oliver. System design of a computer for time sharing applications. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, pages 197–202, 1965.

[38] J. A. Goguen and J. Meseguer. Security policies and security models. *Security and Privacy, IEEE Symposium on*, 0:11, 1982.

[39] James W. Gray III. On introducing noise into the bus-contention channel. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, pages 90–98, 1993.

[40] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[41] Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *IEEE Symposium on Security and Privacy*, pages 129 –142, 2008.

[42] Nevin Heintze and Jon G. Riecke. The slam calculus: programming with secrecy and integrity. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, New York, NY, USA, 1998. ACM.

[43] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, pages 1–17, 2006.

[44] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 8 –20, 1991.

[45] Wei Hu, J. Oberg, J. Barrientos, Dejun Mu, and R. Kastner. Expanding gate level information flow tracking for multilevel security. *Embedded Systems Letters, IEEE*, 5(2):25–28, 2013.

[46] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. Theoretical fundamentals of gate level information flow tracking. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(8):1128–1140, 2011.

[47] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. On the complexity of generating gate level information flow tracking logic. *Information Forensics and Security, IEEE Transactions on*, 7(3):1067–1080, 2012.

[48] Wei Hu, Jason Oberg, Dejun Mu, and Ryan Kastner. Simultaneous information flow security and circuit redundancy in boolean gates. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '12, pages 585–590, 2012.

[49] IEEE Computer Society, New York, NY, USA. *IEEE Standard Verilog Hardware Description Language*. IEEE Std 1364-2001.

[50] Intel Corporation. AAJ1 Clarification of TRANSLATION LOOKASIDE BUFFERS, Intel® Core™ i7-900 Desktop Processor Extreme Edition Series and Intel® Core™ i7–900 Desktop Processor Series Datasheet. May 2011.

[51] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 270–283. ACM, 2010.

[52] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Trans. Softw. Eng.*, pages 1147–1165, 1991.

[53] Ryan Kastner, Jason Oberg, Wei Hu, and Ali Irturk. Enforcing information flow guarantees in reconfigurable systems with mix-trusted IP. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2011.

[54] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[55] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, 1996.

[56] B. Köpf and G. Smith. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 44–56, July 2010.

[57] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *Proceedings of IEEE Symposium on Security and Privacy ("Oakland") 2010*, pages 447–462, 2010.

[58] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *In Proceedings of the Symposium on Operating System Principles (SOSP)*, 2007.

[59] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *SOSP 2007*, pages 321–334, 2007.

[60] Lap Chung Lam and Tzi cker Chiueh. A general dynamic information flow tracking framework for security applications. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[61] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, aug 1996.

[62] Edward A Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 17(12):1217–1229, 1998.

[63] Xun Li, Vineeth Kashyap, Jason Oberg, Mohit Tiwari, Vasanth Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A language for hardware-level security policy enforcement. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, 2014.

[64] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: a hardware description language for secure information flow. In *PLDI 2011*, pages 109–120, 2011.

[65] S. Mane, M. Taha, and P. Schaumont. Efficient and side-channel-secure block cipher implementation with custom instructions on fpga. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 20–25, Aug 2012.

[66] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[67] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 118–129, 2012.

[68] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, 2013.

[69] Mentor Graphics Corporation. *ModelSim Reference Manual*, 2010.

[70] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. On the vulnerability of fpga bitstream encryption against power analysis attacks: Extracting keys from xilinx virtex-ii fpgas. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 111–124, 2011.

[71] Amir Moradi, David Oswald, Christof Paar, and Pawel Swierczynski. Side-channel attacks on the bitstream encryption mechanism of altera stratix ii: Facilitating black-box analysis using software reverse-engineering. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 91–100, 2013.

[72] A. C. Myers. Jflow: Practical mostly-static information flow control. In *In Proceedings of the 26th Symposium on Principles of Programming Languages (POPL)*, 1999.

[73] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *In Proceedings of the 16th Symposium on Operating System Principles (SOSP)*, 1997.

[74] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. software release. located at http://www.cs.cornell.edu/jif, July 2001.

[75] "BBC News". Warning over medical implant attacks, April 2012. http://www.bbc.co.uk/news/technology-17623948.

[76] T. Newsham. Format string attacks, September 2000. http://hackerproof.org/technotes/format/formatstring.pdf.

[77] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Theoretical analysis of gate level information flow tracking. In *Design Automation Conference*, pages 244–247, 2010.

[78] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Information flow isolation in I2C and USB. In *Proceedings of Design Automation Conference (DAC) 2011*, pages 254 –259, 2011.

[79] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. A practical testing framework for isolating hardware timing channels. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1281–1284, 2013.

[80] Jason Oberg, Timothy Sherwood, and Ryan Kastner. Eliminating timing information flows in a mix-trusted system-on-chip. *Design Test, IEEE*, 30(2):55–62, 2013.

[81] National Institute of Standards and Technology (NIST). Federal information processing standards (fips) publications: Fips 140–2, security requirements for cryptographic modules, May 2001.

[82] Opencores.org. Basicrsa encryption engine. http://opencores.org/project,basicrsa, March 2009.

[83] Opencores.org. 128-bit verilog aes core. http://opencores.org/project ,systemcaes, April 2010.

[84] Opencores.org. Wishbone specification. http://opencores.org/opencores,wishbone, June 2010.

[85] Sddka Berna Ors, Elisabeth Oswald, and Bart Preneel. Power-analysis attacks on an fpga - first experimental results. In *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003), Lecture Notes in Computer Science Volume 2779*, pages 35–50, 2003.

[86] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology*, pages 1–20, 2006.

[87] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against spn structures, with application to the aes and khazad. In *Cryptographic Hardware and Embedded Systems-CHES 2003*, pages 77–88. Springer, 2003.

[88] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[89] Francois Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

[90] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[91] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In *Proceedings of CCS 2009*, pages 199–212, 2009.

[92] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communication of ACM*, 21(2):120–126, February 1978.

[93] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 35–45, New York, NY, USA, 2008. ACM.

[94] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[95] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3):157–170, 1972.

[96] M.E. Smid and D.K. Branstad. Data encryption standard: past and future. *Proceedings of the IEEE*, 76(5):550–559, May 1988.

[97] Green Hills Software. The integrity real-time operating system. http://www.ghs.com/products/rtos/integrity.html.

[98] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS 2004*, pages 85–96, 2004.

[99] M. Tehranipoor and F. Koushanfar. A survey of hardware trojan taxonomy and detection. In *IEEE Design and Test*, 2010 2010.

[100] EE Times. Acceleration case: Jury finds toyota liable, October 2013. http://www.eetimes.com/document.asp?doc_id=1319897.

[101] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: a hardware-supported mechanism for enforcing strong non-interference. In *MICRO 2009*, MICRO 42, pages 493–504, 2009.

[102] Mohit Tiwari, Jason Oberg, Xun Li, Jonathan Valamehr, Timothy E. Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proceedings of ISCA 2011*, pages 189–200, 2011.

[103] Mohit Tiwari, Hassan Wassen, Bita Mazloom, Shashidhar Mysore, Frederic Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proceedings of ASPLOS 2009*, 2009.

[104] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *In Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, 2004.

[105] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Fourteenth International Symposium on High Performance Computer Architecture (HPCA)*, pages 196–206, New York, NY, USA, 2008. ACM.

[106] I. Verbauwhede, F. Hoornaert, J. Vandewalle, and H.J. De Man. Security and performance optimization of a new des data encryption chip. *Solid-State Circuits, IEEE Journal of*, 23(3):647–656, June 1988.

[107] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA 2007*, pages 494–505, 2007.

[108] Zhenghong Wang and Ruby B Lee. A novel cache architecture with enhanced performance and security. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 83–93, 2008.

[109] Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 583–594. ACM, 2013.

[110] Robert S Winternitz. A secure one-way hash function built from des. In *IEEE Symposium on Security and Privacy*, pages 88–90, 1984.

[111] John C. Wray. An analysis of covert timing channels. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 2–7, 1991.

[112] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[113] Heng Yin, Dawn Song, Egele Manuel, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[114] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres. Making information flow explicit in histar. In *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.