UNIVERSITY OF CALIFORNIA
Santa Barbara

Design and Verification of Information Flow Secure Systems

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

Doctor of Philosophy

in

Computer Science

by

Mohit Tiwari

Committee in Charge:

Professor Timothy P. Sherwood, Chair

Professor Frederic T. Chong

Professor Tevfik Bultan

Professor Ben Hardekopf

Professor Ryan Kastner

September 2011

The Dissertation of
Mohit Tiwari is approved:

Professor Frederic T. Chong

Professor Tevfik Bultan

Professor Ben Hardekopf

Professor Ryan Kastner

Professor Timothy P. Sherwood, Committee Chairperson

July 2011

Design and Verification of

Information Flow Secure Systems

Copyright © 2011

by

Mohit Tiwari

Curriculum Vitæ
Mohit Tiwari

Education

2011	Doctor of Philosophy, University of California, Santa Barbara
2010	Masters in Science, University of California, Santa Barbara
2005	Bachelor of Technology, Indian Institute of Technology, Guwahati, India

Honors and Awards

Micro Top Pick	IEEE Micro's Top Picks from Computer Architecture Conferences, January-February 2010.
Best Paper	Parallel Architecture and Compiler Techniques (PACT), Sept 2009. Raleigh, NC.
Outstanding TA	Department of Computer Science, UC Santa Barbara. March 2006.

Research Experience

Sept 2005 – 2011	Graduate Research Assistant, University of California, Santa Barbara.
Jun–Aug 2007	Summer Intern, NEC Labs, Princeton, NJ.
Jun–Aug 2004	Summer Intern, EDA Lab, Politecnico di Torino, Italy.

Publications

Mohit Tiwari, Jason Oberg, Xun Li, Jonathan K Valamehr, 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 the International Symposium of Computer Architecture (ISCA), June 2011. San Jose, CA*

Susmit Biswas, Mohit Tiwari, Luke Theogarajan, Timothy Sherwood, and Frederic T Chong. Fighting Fire with Fire: Modeling the Data Center Scale Effects of Targeted Superlattice Thermal Management. In *Proceedings of the International Symposium of Computer Architecture (ISCA), June 2011. San Jose, CA*

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

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

Xun Li, Mohit Tiwari, Ben Hardekopf, Timothy Sherwood, and Frederic Chong. Secure Information Flow Analysis for Hardware Design: Using the Right Abstraction for the Job. In *Proceedings of the Fifth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2010.

Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood and Ryan Kastner. Theoretical Analysis of Gate Level Information Flow Tracking. In *Proceedings of the 47th Design Automation Conference (DAC)*, June 2010.

Jonathan Valamehr, Mohit Tiwari, Timothy Sherwood, Ryan Kastner, Ted Huffmire, Cynthia Irvine, and Timothy Levin, Hardware Assistance for Trustworthy Systems through 3-D Integration, In *Annual Computer Security Applications Conference (ACSAC), December 2010. Austin, TX.*

Mohit Tiwari, Xun Li, Hassan Wassel, Bita Mazloom, Shashidhar Mysore, Frederic Chong, and Timothy Sherwood. Tracking Information Flow at the Gate-Level for Secure Architectures. In *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences (IEEE Micro - top pick)*, January-February 2010.

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 *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2009. New York, NY

Mohit Tiwari, Shashidhar Mysore, and Timothy Sherwood. Quantifying the Potential for Program Analysis Peripherals. In *Parallel Architecture and Compiler Techniques (PACT)*, Sept 2009. Raleigh, NC

Mohit Tiwari, Hassan M G Wassel, Bita Mazloom, Shashidhar Mysore, Frederic Chong, and Timothy Sherwood. Complete Information Flow Tracking from the Gates Up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2009. Washington, DC

Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan K Valamehr, and Timothy Sherwood. A Small Cache of Large Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, November 2008. Lake Como, Italy

Abstract

Design and Verification of Information Flow Secure Systems

Mohit Tiwari

We show that it is possible to construct hardware-software systems whose implementations are verifiably free from all illegal information flows. This work is motivated by high assurance systems such as aircraft, automobiles, banks, and medical devices where secrets should never leak to unclassified outputs or untrusted programs should never affect critical information. Such systems are so complex that, prior to this work, formal statements about the absence of covert and timing channels could only be made about simplified models of a given system instead of the final system implementation.

This thesis proposes a verification technique, Gate Level Information Flow Tracking, that allows system implementations to be analyzed for all digital information flows. The key insight is that all such flows look surprisingly similar at the level of logic gates, where covert channels through low-level software and timing channels arising from the underlying hardware all become explicit information flows. Our verification technique first constructs a sound approximation of all possible system behaviors, representing an entire system as a large, synchronous

state machine, and then analyzes information flows through this abstract state machine logic to ensure that all possible executions arising from unknown state and inputs conform to a given security policy.

We then devise an architecture and programming model, Execution Leases, that allows programmers to explicitly construct space-time sandboxes to run secret or untrusted programs. Further, we show how to construct a usable embedded system around the constraints imposed by complete information flow tracking. We implement a system that includes a micro-kernel running on top of a processor with complex features such as pipelining and caches, all implemented and tested on an FPGA and verified at the gate level to be free from illegal information flows.

Professor Timothy P. Sherwood
Dissertation Committee Chair

Contents

List of Figures

Chapter 1

Introduction

This dissertation presents a verification technique and design methodology to construct information flow secure systems. The need for formal, implementation-level information flow security is motivated by high assurance computing systems that control aircraft, automobiles, and medical devices. With the growing use of on-demand, untrusted compute facilities for sensitive commercial and personal purposes, the demand for strong information flow-security is increasingly becoming more mainstream.

One desirable information flow policy is to maintain *integrity*, so that no untrusted component of the system can disrupt the execution of trusted components. Another policy can be to preserve *confidentiality*, where no secret information ever flows undetected to an unclassified component. These policies map intuitively to real concerns such as passenger internet never affecting critical aircraft controls or the root key of a bank not leaking to a malicious program. In both cases, we label

the inputs to the system as being either trusted/untrusted or secret/unclassified, and track the flow of these labels to ensure that no untrusted data affects trusted outputs or that no secrets leak to unclassified outputs.

We show that only by peering far below the hardware-software interface, at a level where bits are operated on by simple logic gates, is it possible to account for all information flows and build systems with strong, implementation-level guarantees for information flow policies. Specifically, we show that all digital information flows through a system, whether they look like explicit, implicit or timing channels to the software, can be expressed as logic functions of its inputs and input labels. We present Gate Level Information Flow Tracking (GLIFT), a technique that can track all information flows precisely through an arbitrary combinational block, and extend it to compositionally analyze gate-level descriptions of entire microprocessors. We then describe a novel processor architecture, Execution Leases, that presents programmers with explicit control over all information flows, and use GLIFT to verify its information flow behavior at the gate-level. Finally, we design and implement a complete embedded system, including a CPU with optimizations such as caches and pipelines, a separation kernel, and an I/O sub-system, by constructing the system around a small, trusted hardware-software skeleton that manages information flows through the rest of the system. Together, these steps take us closer to our goal of an end-to-end information flow secure system.

Before we dive into the details of each step, we begin with some background on the growing demand for high assurance systems in traditional and emerging domains.

1.1 High Assurance Systems

The strict demands on the design of high assurance systems are a direct function of the cost of failure. As an example, Boeing plans to use a single physical network for both aircraft control data and passenger network data in its new 787 [26], and keeping these two *very strictly* separated is obviously critical. Surprisingly, while avionics controllers in the 787 require about 6.5 million lines of code, a premium-class automobile today includes 70 – 100 electronic control units running close to 100 million lines of code. These control units interact over one or more shared networks to monitor the state of the machine and control everything from the brakes to the radio [5]. With the push towards "car as a platform" for third-party applications, that provide services such as vehicular internet and entertainment, the on-board software is expected to grow to 200 – 300 million lines of code in the near future. Even though it is clear that all untrusted programs and hardware components should be strictly sandboxed, virtually any malicious

control unit can bypass the rudimentary protection mechanisms and take over control of all automotive functions [43].

While the above examples discuss the problem of interference between various software components on the same processor from the view point of preserving *integrity*, exactly the same problem occurs in systems concerned with *secrecy*. If a secret such as a private key is used in performing an operation, an attacker may be able to reverse engineer the key through direct timing observations[39], cache interference[55], even through the state of the branch predictor [10]. Once an attack has been identified and publicized, effective countermeasures can be deployed, e.g. randomizing the replacement policy for the cache[48], but this constant cycle of attack-and-respond is unsatisfying when the cost of leaking some data is extraordinarily high (for example the root private key for a bank, or a military authorization code). Such attacks have now assumed a much wider significance due to the growth of cloud computing, where malicious virtual machines can mount side-channel attacks over shared hardware and thus infer secret information from a target virtual machine [57].

Although there are many useful software techniques to help developers tackle information flow security at the programming language level [75, 49, 52, 62], in special purpose operating systems [36, 51, 4, 82, 59, 44], with software virtual machines [7, 20, 28, 81], or through careful static analysis [14], true end-to-end

properties are often foiled by the refinement from language-level specification to actual hardware-level execution. It is estimated that verifying software running on commodity hardware to an assurance rating of EAL6 costs on the order of $10k per line of code and takes over 10 years [4], and in the end it is not clear that the end product will ever even make it to EAL7 because of the difficulties in formally proving the most critical aspects of the design. One of the primary difficulties in getting software verified at these levels is that modern machines are simply not built with the idea of information containment in mind.

1.2 Thesis Statement and Dissertation Roadmap

I propose that all digital information flows become explicit at the level of logic gates, an observation we can then use to verify gate-level implementations of arbitrary systems, and specifically to verify a complete embedded system that allows explicit programmer control over all information flows.

The rest of this chapter provides an overview of our proposed steps towards a system that provides strong information flow guarantees.

1.2.1 Gate-Level Information Flow Tracking

We first present a novel technique capable of tracking all information flows within a machine, including all explicit data transfers and all implicit flows (those subtly devious flows caused by not performing conditional operations). Through the application of our novel gate-level information flow tracking (GLIFT) method, we show how all flows of information can be tracked precisely for small circuits and compositionally for larger designs. From this foundation, we then describe how a class of architectures can be constructed, from the gates up, to completely capture all information flows and we measure the impact of doing so on the hardware implementation, the ISA, and the programmer. While the problem is impossible to solve in the general case, we create a machine that avoids the general-purpose programmability that leads to this impossibility result, yet is still programmable enough to handle a variety of critical operations such as public-key encryption and authentication.

1.2.2 Information Flow Secure Architecture and Programming Model

We describe a new method for creating architectures that both a) makes the complete information-flow properties of the machine fully explicit and available to

the programmer and b) allows those properties to be verified all the way down to the gate-level implementation the design. The core of our contribution is a new call-and-return mechanism, *Execution Leases*, that allows regions of execution to be tightly quarantined and their side effects to be tightly bounded. Because information can flow through untrusted program counters, stack pointer or other global processor state, these and other states are *leased* to untrusted environments with an architectural bound on both the time and memory that will be accessible to the untrusted code. We demonstrate through a set of novel micro-architectural modifications that these leases can be enforced precisely enough to form the basis for information-flow bounded function calls, table lookups, and mixed-trust execution, and show the effectiveness of the resulting design through the development of a new language, compiler, ISA, and synthesizable prototype.

1.2.3 Building Full Systems with Kernel and I/O

High assurance systems often rely on a small trusted base of hardware and software to manage the rest of the system. Crafting the core of such a system in a way that achieves flexibility, security, and performance requires a careful balancing act. Simple static primitives presented above, with hard partitions of space and time, are easier to analyze formally, but such strict approaches to the

problem at the hardware level are extremely restrictive, failing to allow even the simplest of dynamic behaviors to be expressed.

Our approach to this problem is to construct a minimal but configurable *architectural skeleton*. This skeleton couples a critical slice of the low level hardware implementation with a microkernel in a way that allows information flow properties of the entire construction to be statically verified all the way down to its gate-level implementation. This strict structure is then made usable by a runtime system that delivers more traditional services (e.g. communication interfaces and long-living contexts) in a way that is decoupled from the information flow properties of the skeleton. To test the viability of this approach we design, test, and statically verify the information-flow security of a hardware/software system complete with support for unbounded operation, inter-process communication, pipelined operation, and I/O with traditional devices. The resulting system is provably sound even when adversaries are allowed to execute arbitrary code on the machine, yet is flexible enough to allow caching, pipelining, and other common case optimizations.

Chapter 2

Complete Information Flow Tracking from the Gates Up

The enforcement of information flow policies is one of the most important aspects of modern computer security, yet is also one of the hardest to get correct in implementation. The recent explosion of work on dynamic dataflow tracking architectures has led to many clever new ways through which information can be accounted for in modern software, leading to novel ways of detecting everything from general code injection attacks to cross-site scripting attacks [24, 79]. The basic scheme keeps track of a binary property, trusted or untrusted, for every piece of data. Data from "untrusted" sources (e.g. from the network) are marked as untrusted, and the output of an instruction is marked as untrusted if any of its inputs are untrusted. While these systems will likely prove themselves useful in a variety of real-life security scenarios, ultimately it is impossible for

these techniques, or in fact for any security system running on a general-purpose processor, to provably capture all of the information flow within the machine [25].

The problem is that in a traditional microprocessor, information is leaked practically everywhere and by everything. If you are executing an exceedingly critical piece of software, for example, using your private key to sign an important message, information about that key is leaked in some form or another by almost everything that you do with it. The time it takes to perform the authentication, the elements in the cache you displace due to your operations, the paths through the code the encryption software takes, even the paths through your code that are never taken can leak information about the key.

While this information leakage may not be a consideration when you are executing a word processor, leakage can be a serious problem for exceptionally sensitive financial, military, and personal data. Developers in these domains are willing to go to remarkable lengths to minimize the amount of leaked information, for example, flushing the cache before and after executing a piece of critical code [55], attempting to scrub the branch predictor state [8], normalizing the execution time of loops by hand [39], and by randomizing or prioritizing the placement of data into the cache [48]. While these techniques make it more difficult for an adversary to gain useful knowledge of sensitive information, at the end of the day these heuristics cannot bring the system significantly closer to a formally strong no-

tion of information flow tracking because they do not take into consideration the intricate logic and timing that compose the implementation.

In this chapter we present the first ever processor architecture and implementation that can track *all* information-flows. On such a microprocessor it is impossible for an adversary to hide the flow of information through the design, whether that flow was intended by both parties (e.g. through a covert channel) or not (e.g. through a timing-channel). One of the key insights in this paper is that all information flows, whether implicit, covert, or explicit, look surprisingly similar at the *gate level* where weakly defined ISA descriptions give way to precise logical functions. While past approaches have assumed that any use of untrusted data should lead to an untrusted output, we observe that at the gate level this is overly conservative. If one input to an AND gate is 0, the other input can *never* affect the result and thus *should have no bearing on the trust of the output*. Based upon this observation, we introduce a novel logic discipline, *Gate-Level Information-Flow Tracking (GLIFT)* logic, which is built around a precise method for augmenting arbitrary logic blocks with tracking logic and a further method for making compositions of those blocks. Using this discipline we demonstrate how to create an architecture that, while unconventional in ways required by the very nature of being free from the problems of implicit-flow, is both programmable and capable of performing useful computation. We present a synthesizable processor

implementation with a restricted ISA, predicated execution, bounded loops, and an iteration-coupled load/store architecture. Combined with GLIFT logic, these restrictions provide tractable and provably-sound information-flow tracking, yet allow tasks such as public-key cryptography and message authentication to be performed.

In Section 2.2 we describe how architectural information flows at the level of gates and present a novel compositional method by which arbitrary logic functions can be analyzed to create the fundamental building blocks of our secure hardware. In Section 2.3 we then describe the three major pitfalls in designing an architecture free of implicit flows, how our ISA avoids them, and how our gate-level implementation correctly tracks the resulting information flows in a provably-sound way. To ensure that the resulting architecture is not unreasonable in the additional overhead it incurs, in Section 2.4 we describe how this microprocessor compares with a conventional microcontroller in terms of area and performance. However, before we can begin the details of our solution, we need to begin with a discussion of the great deal of related work in both computer architecture and security that this work has built upon.

2.1 Related Work

The idea of tracking and constraining the flow of information is one of the primary tenets of computer security, and all manner of work has examined both the practical and theoretical limitations of mechanisms that perform this function. As has been pointed out countless times before, the general problem of determining whether information flows in a program from variable x to variable y is undecidable, as "any procedure purported to decide it could be applied to the statement **if** $f(x)$ halts **then** $y := 0$ and thus provide a solution to the halting problem for arbitrary recursive function" [25]. This is a classical example of an *implicit flow*, where information flows between variables by virtue of their *not* being accessed. For example, in the pseudo code "**if** i **then** $j := 1$", even if "$j := 1$" is *never* executed because i is always false, by observing j we can learn something about i and hence there is an information flow between i and j. If you have a Turing-complete machine, it is impossible to bound the set of possible actions that the machine might make in some conditional situation (à la the Halting Problem), and hence for any general-purpose programmable machine, it is impossible to precisely prevent *all* implicit flows. We believe our solution to this quandary is unique in that we have built a machine that, by construction, will not allow unbounded execution. In fact our design, which is still programmable through an ISA (al-

beit a non-traditional one), is theoretically equivalent to a single very large state machine. While this certainly limits the applicability of the machine, unbounded execution is not required to sort a bounded-size list, encrypt a message, or even verify a message signature. In the end we have created a machine in which *all* hidden flows of information are made explicit.

Using hardware to track the flow of information through a processor is by no means a new idea. DIFT [67], Minos [22], Rifle [71], Raksha [24], FlexiTaint [72], Log-Based Lifeguards [61] and a host of other proposals suggest the use of data-flow tracking hardware to track the flow of untrusted network, file and user inputs through memory. The basic idea behind these tools is to assign a "tag" with every word of physical memory indicating which words of memory can be trusted, and then to track these tags around the machine as operations are performed. Every time an arithmetic operation uses an untrusted input, the output is marked as "tainted", and whenever an untrusted memory word is used for a sensitive operation like a jump address condition or a system call, the tool generates a warning for the user. Our approach, while inspired by these methods, seeks to strongly couple the notion of information flow to *all* parts of the machine at the gate level, not just the data paths, so that we know for certain that there is no way for information to be manipulated in such a way that it will "lose" the tag that represents its trust.

The idea of data-flow tracking is not limited to hardware-only options. Software projects have shown that data-flow tracking can be useful in detecting a variety of attacks [28, 21, 20, 7, 79, 73, 18], some with surprisingly low overhead (e.g. LIFT [28] and Speculation to Security [19]). In fact this idea can be extended to a generic taint- tracking framework that allows arbitrary policies to be enforced. Dytan [20], GIFT [45], Taint-Enhanced Policy Enforcement [79], Raksha [24], System Tomography [53] and FlexiTaint [72] are all examples of flexible systems for tracking data and/or enforcing polices based on those tags. In addition to explicit dataflow tracking, some prior work has examined the problem of tracking implicit information flows [73, 71, 20, 78]. These approaches track information at the ISA level and attempt to combine dynamic taint tracking with limited static analysis to improve the precision of flow tracking. Our approach is different from these prior methods in that we would like to be able to precisely track all flows for any software that can be written in our ISA, and because we have knowledge of underlying hardware, we can take into consideration the logical implementation including all of its undocumented features, bugs, and timing channels.

It is worth noting explicitly what information leaks and attacks our proposed approach, taken in isolation, does not address. We do not explore the untrusted hardware component problem or physical attacks that may tamper with memory.

There is already a great deal of work on tamper resistant computing [66]. Nor do we consider non-digital side-channel attacks (such as those informed by observation of power distribution [38] or RF radiation [29]), as again, there are many circuit level methods for dealing with those. Instead, our approach allows us to treat the microprocessor simply as an object through which both trusted and untrusted information flows, allowing us to be certain as to which resulting outputs rely on that untrusted input. We have already begun to see mainstream processors with physically isolated protection domains, such as ARM's TrustZone [11] and Cell Broadband Engine's Synergistic Processor Element [63], as a first step towards preventing trusted and untrusted data from intermingling. While, as you will see, our resulting system is not yet efficient in the traditional sense, we believe it is a leap toward the goal of a microprocessor capable of provably tracking and policing all information flows on chip.

2.2 Gate Level Information Flow Tracking

Tracking all information flows through a full microprocessor is a daunting task, but one that we can tackle by breaking it down into small pieces. In this section, we begin with the smallest atomic units of logic in the microprocessor: gates. Once we precisely understand how information flows through the primitive NOT, AND,

and OR gates, we can begin to compose these gates together into more complex structures such as multiplexers, arithmetic units, and eventually full processors that are able to manage and manipulate information in such a way that trust can be tracked through the implementation in a sound and precise way.

While our techniques can be extended to cover a variety of information flow security scenarios, for the purpose of this paper we will restrict our discussion to simple binary tags. Data and code are simply either "trusted" (represented logically as 0) or as "untrusted" (represented logically as 1). We have chosen a representation that is close to "taint" tracking, although we adopt the nomenclature of the security community as this is a more general information flow tracking problem rather than specifically data flow tracking. We wish to treat our whole processor as a logical function, one which operates on a set of inputs (some of which are trusted and some of which are not) and results in a set of outputs. The trust of the outputs should be determined based on the trust of the inputs, and more specifically on *how* untrusted inputs affected those outputs. To more fully illustrate the notion of trust propagation at the logical level, let's consider a very simple gate, AND. Surprisingly, even for this simple gate, the trustworthiness of the output is a complex function of the trustworthiness of the inputs *and* the actual logical values of those inputs.

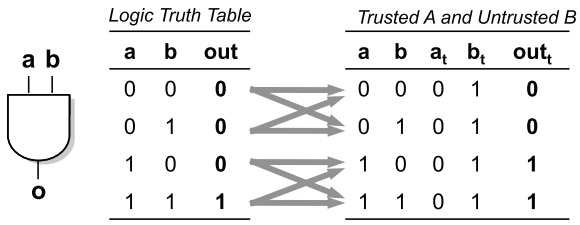

Figure 2.1: Tracking Information Flow through a 2-input AND Gate. Figure shows truth table for the AND Gate (left) and a part of its shadow truth table (right). The shadow truth table shows the interesting case when only one of the inputs a and b is trusted (i.e. $a_t = 0$ and $b_t = 1$). Each row of the shadow table calculates the trust value of the output (out_t) by checking whether the untrusted input b can affect the output out. This requires checking out for both values of b in the table on the left. The gray arrows indicate the rows that have to be checked for each row on the right. For example, when $a = 1$, b affects out (row 3 and 4 on the left). Hence row 3 and 4 on the right have out_t as untrusted.

2.2.1 Information Flow Tracking in an AND gate

Consider an AND gate (shown in left side of Figure 2.1) with two binary inputs, a and b, and an output o. Let's assume for right now that this is our entire system, and that the inputs to this AND gate can come from either trusted or untrusted sources, and that those inputs are marked with a bit (a_t and b_t respectively) such that a 1 indicates that the data is untrusted. The basic problem of gate-level information flow tracking is to determine, given some input for a and b and their corresponding trust bits a_t and b_t, whether or not the output o is trusted (which is then added as an extra output of the function o_t).

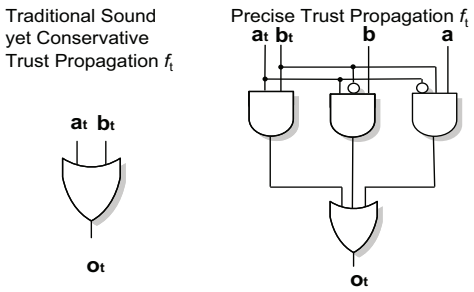

Figure 2.2: Shadow logic for an AND Gate. Conventional information flow tracking reports the output as untrusted if any one of the inputs is untrusted. The circuit on the right shows our shadow AND gate that marks out_t as untrusted only if *out* depends upon an untrusted input.

To the best of our knowledge, all prior work in the area has assumed that if you compute a function, any function, of two inputs, then the output should be tagged as tainted if *either* of the inputs are tainted. This assumption is certainly sound (it should never lead to a case, wherein output which should not be trusted is marked as trusted) but it is over conservative in many important cases, in particular if something is known about the actual inputs to the function at runtime. In fact, from an information theoretic standpoint, the output of a logical function should only be untrusted if some untrusted input actually had an opportunity to affect the output[1].

[1]while this is jumping ahead somewhat, readers familiar with implicit flows may think this sounds dangerously similar. The key difference is that we are talking about logical functions, and in a logical function it is completely possible for some inputs to have absolutely no bearing on any measurable output. The danger of implicit flows in a microprocessor is different because an action which did not happen (for example a branch of code not being taken) may result in a measurable difference of output (for example a variable not being set equal to 1).

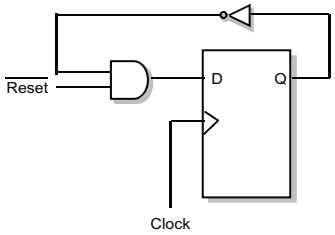

Figure 2.3: A 1-bit counter with reset. With the conventional technique of ORing all input shadow values, the feedback loop ensures that a counter shall never be trusted once it gets marked as untrusted. Our shadow logic is more precise and recognizes that a trusted reset guarantees a trusted 0 in the counter value.

To see why, let us just consider the AND gate, and all of the possible input cases. If both of the inputs are trusted, then the output should clearly be trusted. If both the inputs are untrusted, the output is again clearly untrusted. The interesting cases are when you have a mix of trusted and untrusted data. If input a is trusted and set to 1, and input b is untrusted, the output of the AND gate is always equal to the input b, which, being untrusted, means that the output should also be untrusted. However, if input a is trusted and set to 0, and input b is untrusted, the result will always be 0 regardless of the untrusted value. The untrusted value has absolutely no effect on the output and hence the output can inherit the trust of a. By including the actual values of the inputs into the determination of whether the output is trusted or not trusted, we can more precisely determine whether the output of a logic function is trusted or not.

So, how do we formalize this notion of untrusted inputs "affecting" outputs? Essentially we are going to create a new truth table, which will *shadow* the original logic, but instead of computing the output (o), it will compute the trust of the output (o_t) as a function of the logical inputs (a and b), the trust of those inputs (a_t and b_t), and the truth table of the original function. Let us consider the case again where a is trusted (untrusted bit set to 0) and b is not (again in Figure 2.1). To compute the first line in our shadow truth table, we must consider all the possible values of the untrusted inputs (b), and if by changing b we can cause the output (o) to be a different value, then we know that the result cannot be trusted. For the first line of the shadow truth table, it means we need to consider the first two lines of the original truth table (the dependencies are drawn with gray arrows in the figure). Because the output is 0 for both values of b, we know that b, even if it was trying to, cannot affect the output. For the last line of the shadow truth table, we need to consider the bottom two lines of the original truth table. Because b can have an effect on the different outputs, the resulting value cannot be trusted. We can continue this process and enumerate the truth table (with 16 entries in all) for the AND gate. After minimizing to an or-of-ands representation, the resulting shadow logic is shown in Figure 2.2.

While this seems like an awful lot of trouble to track the information flow through an AND gate, the difference in terms of the ability to build a machine that

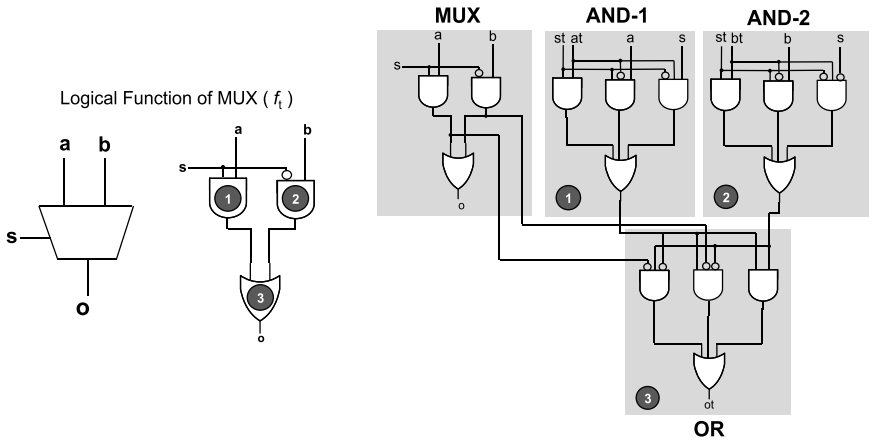

Figure 2.4: Composing information flow tracking logic for larger functions using basic shadow cells. Figure shows a 2-input MUX composed of AND gates (1 and 2) and an OR gate (3). A shadow MUX is composed of shadow AND-1, shadow AND-2 and a shadow OR cells wired together the same way as the original AND1,2 and OR gates.

effectively manages the flow of information is immense. Consider an extremely simple 1-bit counter that increments (or toggles in this case) every cycle, or gets cleared back to zero due to a reset. If we implement that counter as depicted in Figure 2.3, and use the conservative scheme from above, there is no way for that counter to ever come to a trusted state once it has been marked untrusted. However, if you use our gate-level information flow to determine the trust value, once a trusted reset has been set we know that the counter is in a trusted state 0. While this example is extremely simple, we can continue this analysis further and cover the other primitive gates and eventually analyze even the most complex of logical functions.

2.2.2 Composing Larger Functions

While the truth table method that we describe above is the most precise way of analyzing logic functions, our end goal is to create an entire processor using this technology. Our resulting machine is essentially going to be a large logic function which transforms a state (including the internal state of the processor, such as the program counter, and all architecturally visible state, such as the register file), to a new state based on inputs. Clearly, enumerating this entire truth table (which would have approximately 2^{769} rows, where 769 is the number of state bits in our processor prototype) is not feasible, therefore we need a way of composing functions from smaller functions in a way that preserves the soundness of information flow tracking. Again, taking a smaller example to demonstrate the larger principle, let's consider a multiplexer.

A multiplexer is small enough that we could enumerate the entire function, but another way to create one is from logical gates such as AND and OR. Figure 2.4 shows both the logical implementation and the shadow logic. To create this shadow logic we need access to all the inputs of the MUX, and all the connections between the gates from which it is constructed. Each one of the gates from which the MUX is constructed (two AND and one OR) has a corresponding shadow logic instantiated. For example the shadow logic for ANDs (1) and (2) in the figure is simply the logic derived in Section 2.2.1. The shadow logic for OR

(3), created in the same way as the AND gate, is then instantiated, and is fed the inputs from the outputs of the AND gates and the outputs of the AND shadow logic.

One nice thing about considering a smaller example is that it is still possible to write the truth table for this example, and compare it to the result of this composition. Surprisingly, the functions are not quite identical. The shadow logic created compositionally is, in fact, slightly more conservative than the shadow logic derived directly from the truth table. This is because the compositional approach cannot take advantage of the fact that, due to the particulars of this logic, it's impossible for the outputs of AND-1 and AND-2 to both be set to 1 at the same time, yet our OR-gate shadow logic is assuming this is possible. In this way, a compositional approach may not be exactly precise, but will always be sound. In trying to calculate whether or not an untrusted input can affect output, we are essentially assuming that those uninterested inputs have more flexibility in trying to affect the output than they actually do. For our MUX example, both the precise shadow logic and the one resulting from our compositional approach are precise enough to allow us to build useful architectures. Both capture the notion that if the select line is trusted, and the input it is selecting is trusted, the resulting output should be trusted regardless of the trustworthiness of the other input (which makes intuitive sense from an architectural perspective). Further,

if the select line is untrusted, the output of the MUX will always be untrusted, except for the case when both inputs are trusted and equal. This behavior is desirable since both inputs being trusted and equal is the only case where an untrusted select cannot affect the MUX's output. More precisely, the trust value of the output of a MUX can be described by:

$$o_t = a_t s \vee b_t \bar{s} \vee s_t a_t \vee s_t a \vee s_t b_t \vee s_t b$$

In fact the MUX, by being able to select between trusted and untrusted inputs in a way that does not propagate excessively conservatively, is the foundation of our entire architecture. For example, in Section 2.3.1, we will discuss how we use predication to avoid the standard implicit flow problems encountered with branches, and architecturally, predication is really a programmer-visible MUX.

2.3 Designing a Processor for Gate-Level Verification

Now that we have discussed our GLIFT logic method, the next question then becomes how that method can be applied to a programmable device to create an air-tight information flow tracking microprocessor. The goal of our architecture design is to create a full *implementation* that, while not terribly efficient or small, is programmable enough and precise enough in its handling of untrusted data that

it is able to handle several security related tasks, while simultaneously tracking any and *all* information flows emanating from untrusted inputs.

To understand how information flows manifest themselves at the gate level, let us begin with the small snippet of pseudo assembly below which captures nicely the notion of implicit flows discussed in Section 2.1. Assuming `X` is untrusted, should either of `R1` and `R2` be marked as trusted?

```
0x01 br ( X == 0 ) 0x03
0x02 R1 := 1
0x03 R2 := 1
```

Let us start with what the programmer would expect the correct answer to be: `R2` does not appear to depend on the untrusted variable, and hence appears to be trusted. If `X` $\neq 0$ then `R1` should clearly be marked as untrusted (it is set to 1 only because of a decision made on an untrusted variable). In fact, even if `X` $= 0$, the value of `R1` is *still* dependent on `X` (the value of `X` affected value of `R1` and hence there is an implicit flow).

Now let us consider what these operations would look like at the gate level on a traditional architecture that has been augmented with gate-level flow tracking. Figure 2.5 shows a simple example of a branch instruction implemented in hardware. The comparison occurs, and the result is used to control the select line to the Program Counter, which means the PC can no longer be trusted. Once the PC is untrusted, there is no going back because each PC is dependent on the

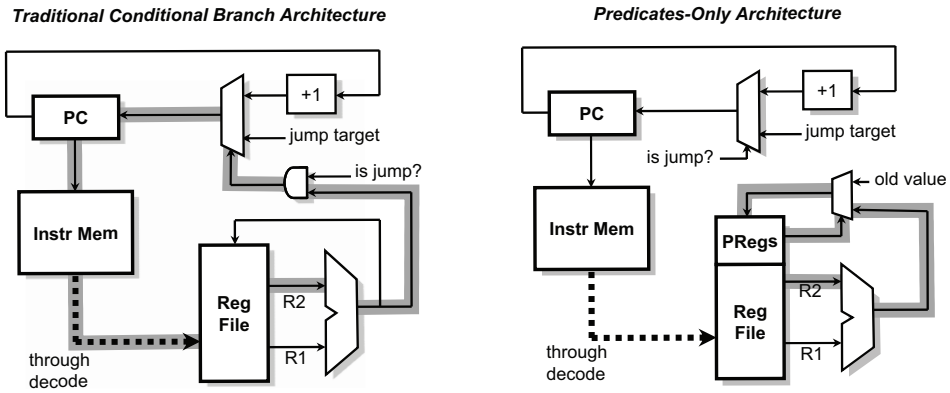

Figure 2.5: Implementation of a conditional branch instruction in a traditional architecture compared to ours. The highlighted wires on the left figure shows the path from an untrusted conditional to the PC. In contrast, we eliminate the path in our architecture so that the PC never gets untrusted.

result of the last. In our example, not only will `R1` be marked as untrusted, `R2` will (seemingly needlessly) be marked as well. In fact, it is even worse than that – because the PC determines the bits that set the register to writeback (and because the PC is marked as untrusted) *all* of the registers (and maybe all of memory) must also be marked as untrusted.

In the architecture described above, `R2` will be marked as untrusted, but is information really flowing from `X` to `R2`? In fact, at the gate level, it is. There is a *timing* dependence between the value of `X` and the time at which `R2` is written. Such timing observations, while seemingly harmless in our example, do represent real information flow and have been used to transmit secret data [9] and reverse engineer secret keys [8].

Modern processors are simply not built to constrain information flow. Rather, they are built to get things done as quickly as possible, often times making use of as much information as possible at each step to make that happen. Our approach to the problem is to restrict both the ISA of the machine and the actual gate level implementation so much that, a) all information flow will be obvious and well understood at the assembly level, b) the actual propagation of trust-bits corresponds closely with this understanding, c) it is impossible to write programs that will result in "explosions" of untrusted state, d) the information flow will be precisely tracked no matter what binary is given to the machine (there is no compiler pre-analysis step required to ensure the strength of information flow tracking) e) it is always possible to return the machine to a trusted state, and f) the shadow information-flow-tracking logic can be composed and added automatically in the way described in Section 2.2 resulting in the tracking of *all* information flows (implicit, timing, covert, or otherwise).

The resulting processor looks like a large state machine, where the state is defined by the architectural and internal state of the processor (PC, flags, registers, counters, etc.), and an arbitrarily large but *finite* amount of memory (a subtle but important distinction). Given the current state at cycle i, you simply compute the next state for cycle $i+1$. In the subsections below we describe several devious

ways in which information will flow through a machine in ways the programmer is not intending, and the architecture changes required to avoid them.

2.3.1 Step 1: Handling Conditionals

As is apparent from our previous example, traditional conditional jumps are problematic, both because they lead to variations in timing and because information is flowing through the PC (which has many unintended consequences). Removing conditionals presents a challenge: how to provide conditional operations without modifying the PC? Predication, by transforming if–then–else blocks into explicit data dependencies (through predicate registers), provides an answer. The effect of an instruction is guarded by a specified predicate register, and if our gate-level information flow method works correctly, the trust-bit of the destination register should be updated regardless of the value of the predicate. Since operations for both cases (predicate true/false) get executed, the augmented processor should track the information flow through every instruction that a program *could possibly* execute, even though only the instructions whose predicates evaluate to true actually write their value back to a register. As shown in Figure 2.5, this ensures the PC is only ever incremented, and no possible flow from untrusted input to the PC is possible.

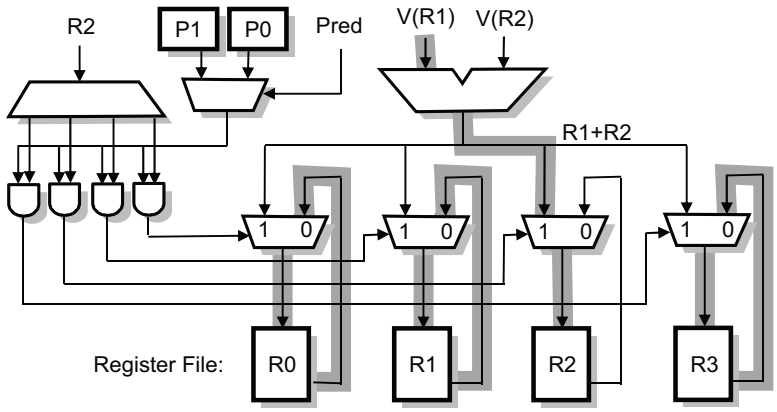

Figure 2.6: Implementation of our predicated architecture. The predicate bits are used to control MUXs that decide whether a register is updated with a new value or gets its old value written back into it. If the predicate bit is untrusted, the shadow MUXs ensure that all registers that *could* have had an untrusted value get marked as untrusted, thus turning implicit information leaks into explicitly tracked trust values.

Figure 2.6 shows the actual logical implementation of predication in our processor. As in a normal predicated architecture, the instruction word specifies the source registers (e.g. `R1` and `R2`) for the instruction, destination register (e.g `R2`), and a predicate register or constant (e.g `P0` or `P1`). If the predicate register stores a 0, then the instruction doesn't write back and instead the old value is written back, but if the predicate is 1 then the new value is written. The shaded lines in the figure illustrate this point more fully. In addition to implementing predication, this example demonstrates a crucial role the MUX plays in our architecture by managing to switch between trusted and untrusted values. Let us consider the

following predicated code, and how trust-bits would flow through the logic in this example.

```
0x01 ( 1)  P1 := not( P0 )
0x02 (P0)  R2 := R1 + R2
0x03 (P1)  R0 := R1 + R2
```

In this code, either of `R0` or `R2` gets the sum (`R1` + `R2`) written into it (based upon the conditional P0). Let us consider what happens to the architecture pictured in Figure 2.6 on instruction 0x02 if `P0` is untrusted. First, the untrusted predicate will be selected by the MUX, and will be used (in conjunction with `R2`) to select the register to write back (this is happening at the bank of small AND gates). As the number 2 flows through the decoder, all of those lines feeding the AND gates *except for the one line controlling* `R2` will be set to 0. For each of those lines, the untrusted predicate is now irrelevant because we can trust that output of the AND gate will be 0 no matter what (as per our discussion of AND gates earlier in the paper), hence, the values on those lines can be trusted. For the one remaining line (the one controlling `R2`), one of the inputs to the AND gate is 1, while the other input in untrusted, and hence the result on that line must be untrusted *no matter whether the predicate is true or false*. That untrusted line will then control the final MUX that determines if the new value or old value should be written back, which will result in `R2` being marked as untrusted (again, regardless of the predicate being true or false).

As a programmer, this complex interplay between the original logic and the information tracking logic is actually quite intuitive. If you predicate an instruction on an untrusted predicate, the destination register will be marked as untrusted. It is as simple as that. As an architect, once you manage to eliminate the spurious information flows, the automated methods described in Section 2.2 actually manage to augment the logic in a way that is both sound and in-line with programmer expectations.

2.3.2 Step 2: Handling Loops

While we can use predication to eliminate the use of conditional jumps in the case of if-then-else blocks, handling loops requires a different approach. Loops are surprisingly difficult to constrain as there are so many different ways for information to leak out in non-obvious ways. Consider a simple while-loop on an untrusted condition. Every instruction in that loop may execute an arbitrary number of times, so everything those instructions touch is untrusted. In fact, everything that *could have been modified*, even if it wasn't, needs to be marked as untrusted. Consider a loop with i going from 0 to X, and setting $\mathrm{A}[i] := 1$. The fact that $\mathrm{A}[X+1] = 0$ tells us something about X, and so there is information flow from X to $\mathrm{A}[X+1]$. In fact there is information flow from X to $\mathrm{A}[X+n]$ for all n less than the maximum possible value X can ever have. Even the fact

that the loop may take an arbitrary number of cycles creates an implicit timing channel with all of the instructions downstream from it.

To limit the effect that loops have on the untrusted state of the system, we have to severely constrain the types of loops that are possible in the system by bounding the side-effects that a loop can have. It needs to be clear, both to the programmer and at the logical implementation, exactly what state has the *possibility* of being affected by the loop. While predication makes the side effects of conditional operation explicit, to deal with loops we use a special `countjump` instruction that specifies statically the number of iterations that should be executed, along with the jump target for the iterations. The processor implementation then maintains a unique iteration counter for the loop instruction and ensures that the counter cannot be modified explicitly by the program.

Counting loop instructions have existed in the context of DSPs for some time, but we believe this is the first time they are being used to aid information tracking. The `countjump` instruction has three interesting details. First, `countjump` has to be unpredicated, implying that it will always commit and a constant amount of jumps to the jump target will always be performed. If `countjump` were to be predicated, it would be exactly equivalent to a conditional jump and would carry all of the same problems discussed in the section above. Second, it is supported by an internal counter that gets set the first time the instruction is encountered.

On all subsequent executions, the counter decrements by 1 until it reaches 0. One further execution will find the counter at 0 and advance the PC by 1 to exit the loop instead of jumping to the jump target. The third detail is that, in order to support nested loops, if a dynamic instruction instance finds the counter at 0, then it gets reset back to the specified value and the entire loop is restarted. This functionality is implemented by an internal state machine that sets the counter back to an uninitialized state when the counter is found to be 0 and the loop is found to be terminated. In Section 2.4.2 we discuss the ramifications of this on the programmer in a bit more detail.

2.3.3 Step 3: Constraining Loads and Stores

The example for loops above also demonstrates a third architecture feature that is problematic for information flow tracking: indirect loads and stores. Most ISAs support indirect memory addressing, where a register's contents provides the address for a load or a store. If the register's contents are untrusted, then using it as an address for a store instruction would implicitly mark the entire address space as untrusted (as any of those addresses could have been affected by that untrusted data). At the logical level, this shows up as the untrusted data address makes its way to the address decoder, and all of the lines of that decoder become untrusted.

Original Logic	Added Logic
reg [31:0] gen_reg [7:0];	reg [31:0] gen_reg_shadow [7:0];
wire [31:0] mux2greg0;	wire [31:0] mux2greg0_shadow;
always @ (posedge clk) begin	
g_reg[0] <= mux2greg0;	g_reg_shadow[0] <= mux2greg0_shadow;
end	
assign is_store = instrn[29] | instrn[22];	assign is_store_shadow = (instrn_shadow[29] & Instrn_shadow[22])
	| (instrn_shadow[29] & (~ instrn_shadow [22]) & (~ instrn [22]))
	| ((~ instrn_shadow[29]) & instrn_shadow[22] & (~ instrn[29]));
mux2x1_32b my_mux0(.in0(g_reg0),	mux2x1_32b_shadow sh_my_mux0(.in0(g_reg0), .in0_t(g_reg0_shadow),
.in1(newval), .sel(p_sel0),	.in1(newval), .in1_t(newval_shadow), .sel(p_sel0), .sel_t(p_sel0_shadow),
.result(mux2greg0));	.ot(mux2greg0_shadow));

Figure 2.7: An example of how our very structured verilog code can be automatically augmented with the logic required to track the trust through the hardware implementation. Each wire, register, and signal is augmented with a corresponding shadow element that stores the 1-bit trust value for each.

Intuitively, the problem is that accessing one untrusted address causes every other address to become implicitly untrusted by virtue of them *not* being accessed or modified. To limit this implicit untrusted state explosion, in our prototype design we have limited our ISA to only support *direct* and *loop-relative* loads and stores. Direct loads use an address encoded in the immediate field, and are used to access fixed memory addresses. To allow access to arrays without resorting to general purpose indirect loads and stores, we have a loop-relative addressing mode, where loads access a variable which is at a fixed constant offset from a loop index (the loop counter used in the `countjump` instruction). This reduces convenience of programming in our ISA substantially but it allows us to precisely track any memory references. We support these by incorporating two new instructions: `load-looprel` and `store-looprel`. These are used to load and store

values from a fixed base address (specified as an immediate field) and an offset stored as set of counters (C0...C7 in our prototype) that can be explicitly initialized and incremented by a fixed value using two new instructions: `init-counter` and `increment-counter`. For example, `load-looprel R0, 0x100, C0` loads the value of `M[0x100 + C0]` into `R0`. The number of times these instructions execute depends upon the number of iterations of the loop, which is fixed, and (as we did for the `countjump` instruction), the local counter initialization and increment instructions commit unconditionally so the set of all addresses that can possibly be accessed in the loop can be determined at run-time.

2.3.4 Implementation and Automatic Shadow Logic Generation

Our prototype processor is implemented in Verilog, and we use Altera's QuartusII software to synthesize it onto a Stratix II FPGA. It is a 32-bit machine with 64KB each of Instruction and Data Memory. It has a program counter, 8 general purpose registers, 2 predicate registers, 8 registers to store loop counters (that count down the number of iterations) and 8 other registers to store explicit array indices (used as offsets for load-looprel and store-looprel instructions). To make the semantics of a state machine precise, all logic is triggered on the positive clock edge, and each cycle simply transforms the set of machine state into a new state

through simple combinational logic. This logic uses the PC to read out an instruction word, decode it, perform data memory accesses and ALU computations and finally write back new values into registers, memory and PC. In practice, block RAMs in Stratix II FPGAs are synchronous and require two cycles to read data out. Our simple processor executes an instruction every 5 cycles similar to the classical 5-stage multicycle machine. We have avoided the complications of pipelining (especially the forwarding logic it requires) for the purposes of this proof-of-concept.

Our processor is written in structural verilog as a composition of gates and module instantiations, along with registers and RAMs to store processor state. To augment this processor with GLIFT logic we proceed in two steps. First, each bit of processor state is explicitly shadowed, meaning every register gets a shadow register, and every memory has a shadow RAM (that stores the 1-bit trust values for each bit of the orginal memory). Second, the logic and signals are shadowed by generating the proper trust propagation logic as described in Section 2.2.

The first step is easily accomplished by simply duplicating the declarations for registers and memory. To handle the second step we create a library of shadow cells that perform information flow tracking for each basic processor component like AND and OR gates, MUX-es, decoders, ALU etc. The shadow logic is wired up with both the inputs to the original function, and also with corresponding

Instruction	Pred	Action	Information Flow
load-immediate	yes	Rdest := immed	Rdest inherits the trust of the predicate
load-direct	yes	Rdest := M[immed]	Rdest is truted if both the predicate and the memory value are trusted
store-direct	yes	M[immed] := Rsrc	The memory value is trusted if the predicate and Rsrc are trusted
load-looprel	yes	Rdest := M[immed + LCount]	Rdest is trusted if the memory value and the predicate are trusted
store-looprel	yes	M[immed + LCount] := Rdest	The memory values is trusted if the predicate and Rdest are trusted
add, sub, and, or, not, xor, shl, shr, cmpeq, cmplt	yes	Standard 3-address register to register operations	Rdest is trusted if the both the inputs to the ALU operation are trusted
predset	yes	Pdest := Rsrc	Pdest is trusted if the predicate and Rsrc are trusted
countjump	no	Jump to target exactly N times (N specified in immediate field)	The loopcounter can only be written by an immediate and should never become untrusted
init-counter	no	LCount := 0	LCount is trusted
increment-counter	no	LCount := LCount + 1	LCount remains trusted

Figure 2.8: An overview of the ISA of our prototype architecture, and the information flow tracking policies that are extracted from the actual logic level implementation.

shadow inputs. While we could spend time describing more formally how this happens, it is easiest to simply see from the resulting verilog code (Figure 2.7).

Programming in the resulting ISA

Figure 2.8 summarizes our instruction set. We eliminate conditional jumps and indirect loads and stores from our ISA, and introduce a countjump instruction to execute fixed-size loops, predicated instructions to implement conditional execution, and restricted loads and stores that use only immediate values. In addition to these instructions, we support various logical (AND, OR, NOT, XOR), arithmetic (ADD, SUB), bitwise (SHR, SHL) and comparison operators. As an example usage of the new instructions, let us consider a code snippet from the

```
0x01  ( 1) load-immediate     P1 := 0              #
0x02  ( 1) init-counter       C0 := 0              # i = 0
0x03  ( 1) load-looprel       R0 := M[0x100 + C0]  # R0 = state[i]
0x04  ( 1) init-counter       C1 := 0              # j = 0
0x05  ( 1) cmpeq              P1 := C1, R0         # if ( j == R0)
0x06  (P1) load-looprel       R1 := M[0x300 + C1]  #    R1 = SBox[j]
0x07  ( 1) increment-counter  C1 := 1              #    j++
0x08  ( 1) countjump          (0x05), 255          # loop back 255 times
0x09  ( 1) store-looprel      M[0x100 + C0] := R1  # state[i] = R1
0x0a  ( 1) increment-counter  C0 := 1              # i++
0x0b  ( 1) countjump          (0x03), 15           # loop back 15 times
```

Figure 2.9: Example usage of the new GLIFT instructions: a code snippet from the `SubBytes` function in the GLIFT implementation of the AES [23] encryption algorithm.

`SubBytes` function in the GLIFT implementation of the AES [23] encryption algorithm (in Figure 2.9). The function substitutes values in the `state` matrix with values in the `SBox`. The code below loads the value in the `state` matrix (which in this example is stored starting at address 0x100). Every loaded element serves as an index into the `SBox`, and is substituted by the value in the `SBox` (which is stored starting at address 0x300). The `state` has 16 elements and the `SBox` is a 256 entry table, correspondingly, the `countjump` instructions 0x0b and 0x08 loop back a fixed number of times (15 and 255 respectively).

2.4 Evaluation

To demonstrate that our proposed architecture is actually implementable, we have built a working model of our processor on an FPGA, and we have written several application kernels to help us quantify the overheads involved. Figure 2.10 shows one portion of that result, the area and frequency overhead of our proposed architecture, both with and without GLIFT logic added, as compared to a NIOS processor.

2.4.1 Hardware Impact

We use Altera's Nios processor as a point of comparison as it has a RISC instruction set, and, as a commercial product, is reasonably well optimized. The Nios can be instantiated as either an economy core (Nios-econ) or a standard core (Nios-std). The economy version is an unpipelined 6 stage multicycle processor without caches, branch-predictors etc. (most closely comparable with our core), while the standard version is pipelined and has an additional 4KB instruction cache. The area and timing numbers have been generated by synthesizing the GLIFT-base (with no information flow tracking logic), GLIFT-full and Altera-Nios processors onto a Stratix II device with compilation settings balancing optimization for both area and delay. In Figure 2.10, the left Y-axis shows the

area required to implement the processors measured in Adaptive Look-Up Table (ALUT) units (the logic cells used by Altera Stratix II FPGAs), while the right Y-axis shows the maximum frequency (Fmax) of the processors.

Our base processor is almost equal in area to Nios-standard, and about double the size of Nios-economy. Adding the information flow tracking logic to the base processor increases its area by 70%, to about 1700 ALUTs. However, in terms of absolute size, even the now outdated Stratix II FPGAs have upto 180K ALUTs, while all the above processors consume only in the range of 1K-2K ALUTs. On the right Y axis, Fmax for Altera Nios processors is around 160MHz, while both the base and full GLIFT processors have an Fmax of around 130MHz. In terms of delay, both GLIFT and Nios are multi-cycle processors with the path through the ALU to the destination register being the most critical. The extra tracking logic does not impose a significant overhead on the Fmax, reducing it from 131MHz to 129MHz. Further, the GLIFT processors operate at 130 MHz as opposed to 160 MHz because we include a barrel shift that the Nios does not (with 1-b shifts, our processor also operates at 160MHz.). While these overheads are certainly non-trivial, keep in mind that this version of the processor shadows *every* bit in the machine. By trading off precision for efficiency it may be possible to keep the soundness of our result while reducing the performance impact.

2.4.2 Analysis of Application Kernels

To test the programmability of our design, we have hand coded a set of applications kernels onto our ISA. This allows us to examine the impact of our modified ISA on the static code size and the dynamic instruction count of the programs. Our kernels are drawn from the potential program security uses of a strong information flow tracking system including a public key encryption algorithm (RSA), a block cipher (AES), a cryptographic hash (md5), along with a small finite state machine (CSMA-CD), and a sorting algorithm (bubble-sort).

Mapping applications onto our ISA requires converting conditional if-else constructs into predicated blocks, turning variable sized loops into fixed size ones (by bounding them), and turning indirect loads/stores into direct memory accesses or loop-relative ones using the loop counters. In general, any application that has predominantly regular behavior should execute without much additional overhead, while dynamic behavior such as irregular array accesses will incur much greater inefficiency. For our experiments, we implemented each of the programs under test both directly in our assembly and in C. The C programs are compiled down to Nios-RISC executables with "-O2" and emulated with Altera's instruction set simulator (ISS). Our assembly is mapped to our FPGA implementation to ensure the correctness of our design, and is then run through our instruction set simulator

to gather dynamic instruction counts. Figure 2.11 presents the results of those experiments.

In terms of static code size, our new ISA is very close to the Nios-RISC ISA. However, the dynamic instruction counts vary substantially. Programs such as the CSMA-CD finite state machine and AES have numerous table look-ups where each look-up requires a full table iteration. As a result, these have a very large dynamic instruction count in comparison to the general purpose ISA. On the other hand, bubble sort, which also requires array accesses, is fairly efficient because both the Nios and our ISA loop over the entire array N^2 times. Any inefficiency there is owing to instructions that were executed but not written back because their predicates were false. Finally, RSA and md5 have very little in the way of predicated instructions, and both comprise mainly of ALU instructions. For these applications, the number of executed instructions is very close to the Nios. While our assembly is unoptimized and opportunities for optimization abound, the main point of this is to show that it is indeed possible to constrain a processor enough that all information flow is apparent at the gate level, yet still maintain enough programmability that programs can be mapped to it without an unmanageable amount of overhead.

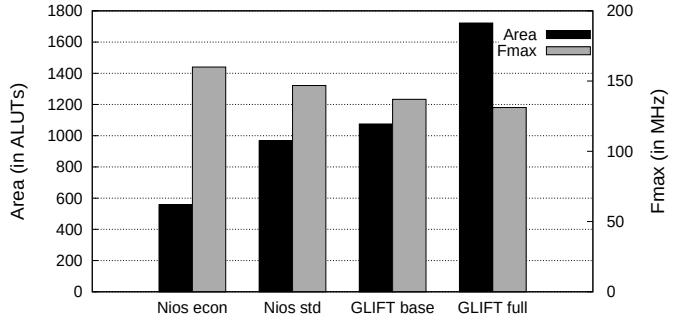

Figure 2.10: Quantifying the area and timing overhead of gate-level information flow tracking. The left Y-axis compares the number of FPGA logic cells required to implement a basic GLIFT processor (which implements the ISA but doesn't include the shadow logic) and a full information flow tracking GLIFT processor to two general purpose micro-processors by Altera, while the right Y-axis shows maximum frequency achieved by each.

2.5 Conclusions

At the end of the day, our new microprocessor is bigger, slower, harder to program, and computationally less powerful than a traditional microcontroller architecture. But what this architecture does for the first time is provide the ability to account for *all* information flows through the chip. It is impossible for an adversary, through clever programming, carefully crafted input, or even the use of covert or timing channels, to ever cause a resulting data element to be marked as "trusted" when in fact it was derived in any way from untrusted data. This is accomplished by tracking the flow of information at the level of gates, where timing signals, predicates, the bits of an address, even the internal results

of logical operations all look like signals on a wire, and all of them are tracked by augmenting those structures using our GLIFT logic transformations. When critical or sensitive operations need to be performed, a co-processor augmented with these abilities could be an attractive option.

We devise a flow tracking logic for simple gates that considers the *effect* of inputs on outputs while propagating taint directly from the truth tables of those gates, and propose a sound composition rule to generate shadow logic for more complex structures. We then show that gate level information flow tracking, when directly applied to a traditional microprocessor, quickly points out many subtle information flows that might be hidden by the ISA abstraction, and at the very worst, lead to a quick explosion of untrusted state. We then go on to describe an architecture that removes these problems while still retaining sufficient programmability to allow it to handle a variety of small but critical tasks. Finally, by implementing a prototype and automatically augmenting it with our information flow tracking logic, we quantify the extra area/delay cost of such flow tracking over a general-purpose micro-controller. While there are many opportunities to further optimize both our architecture and our application kernels, the techniques presented here show that it is indeed possible to track information flows through a programmable design.

Kernel	Description	Static Instruction Count (NIOS)	Static Instruction Count (this work)	Dynamic Instr. Count (NIOS)	Dynamic Instr. Count (this work)	Percentage of Instr. w/ true predicates
FSM	CSMA-CD state machine with with 6 states and 4 inputs. Many table lookups	123	190	441	3322	68%
Sort	Perform bubble sort on a fixed size list of integers	26	21	20621	30358	66%
RSA	Montgomery multiplication and exponentiation from RSA public key cryptography	256	143	44880	39297	84%
AES	Block Cipher, involves extensive table lookups and complex control structures	781	1100	12807	1082207	79%
Md5	Core of the cryptographic hash function, involves mostly ALU and logical operations	769	1386	1226	1431	100%

Figure 2.11: A comparison of the static and dynamic instruction counts for several application kernels on our proposed ISA and an equivalent traditional RISC style architecture (the Nios). While the static instruction counts are comparable, applications that require many irregular accesses to arrays (such as indirect table look-ups) require many more instructions to select out those values.

Chapter 3

Theoretical Foundations of Gate-Level Information Flow Tracking

When designing and implementing a new piece of hardware, the security ramifications of low-level design decisions can be complicated and confusing. Hardware developers typically work from a high level functional specification, such as an instruction set architecture (ISA) manual, a mathematical model, or a functional simulator. These abstractions are necessary steps in the process of refining a design, but gloss over many implementation complexities. However, when the hardware is being built as part of a high assurance design, these complexities are absolutely critical in determining the trustworthiness of the end design.

The problem with high-level hardware specifications is that they often ignore the predictors, caches, buffers, timing variations, and undocumented and unspecified instruction behaviors that are necessary to complete a real system. These

structures complicate higher level information flow evaluations because they can be used to covertly transmit information between security domains [58], to infer the values of secret keys [10, 39], and to subvert documented protection mechanisms [64]. Even worse, many times these details are just ignored.

The goal of this work is to assist developers in creating hardware and firmware with well-defined and statically verifiable information flow properties, and to do so at the lowest digital level of refinement – the level of logic gates. By verifying the properties at this level we are sure to capture the low level timing, implicit flow, and potentially covert channels that are often ignored by higher level analysis.

Given a hardware/firmware design, and an information flow policy (defined over the inputs and outputs of the design), one might ask "will this design always conform to policy". Our approach to the problem is to "convolve" a hardware implementation (e.g. a USB controller) with a circuit formulation of the policy (e.g. Bell-LaPadula) in a way that it creates a new circuit with a single Boolean output with the following properties: If that circuit can be satisfied (i.e. a sequence of input exists that will cause the circuit to output a 1 instead of a 0) then there *may* be a violation, and the details of the satisfying input will lead a designer to the problem; If that circuit can be proven to be unsatisfiable (through enumeration or otherwise) then we know that the hardware implementation will always conform to the policy. Since the analysis is a conservative approximation to the satisfi-

ability problem, i.e. it may return a violation when the circuit is really secure, it is therefore polynomial in the size of the input design. At the same time, our technique requires the design to carefully manage the flow of information so that even a conservative approximation allows designs to pass the verification.

This new analysis circuit, which will only be used for static analysis and never physically implemented in hardware, can then be analyzed by more traditional verification tools. However, constructing an efficient analysis circuit with the above listed properties is difficult: the analysis should ignore inputs and states that are irrelevant to information flow analysis, it should scale to reasonably complex designs, and it should be able to be efficiently and automatically generated for arbitrary designs and information flow policies.

To create this circuit, we start with the actual design to be analyzed, flatten the entire design down to a set of interconnected gates, registers, and memories (the hardware), a set of initial conditions on those memory bits (the firmware), and a set of inputs and outputs with known security labels. Using a set of rules based on the policy lattice, we create our new analysis circuit by replacing the individual gates of the original design with new abstract gates that operate on both abstract values (e.g. $\{0, 1, *\}$: zero, one, and "unknown") and concrete security labels (e.g. trusted and untrusted). A policy violation is then a simple

logical statement over this new domain (e.g. the output at this output port is untrusted).

Specifically, in this chapter:

- We introduce **$*$-logic** (star-logic), a new method of *statically* verifying *arbitrary information flow policies* that works at the gate-level. By carefully designing a hardware-firmware system to work in conjunction with this verification method, all of the states of the resulting design can be efficiently explored.

- We prove that the $*$-logic based abstract execution is sound and that static information flow policy analysis under abstraction is at worst conservative. We also present an algorithm to track security labels through a given boolean circuit for an arbitrary lattice of labels.

- We implement $*$-logic technique as a tool that extracts a gate-level description from a high-level hardware design, converts it into an abstract form that can be efficiently analyzed for policy violations, and verifies compliance using standard design tools. We evaluate the practicality of $*$-logic using an USB bus master controller modified to time-multiplex a single shared bus among mixed-trust I/O devices.

After a discussion of the related work, Section 3.2 describes our verification technique and how it fits into existing design flows. We then formalize these ideas and prove that our verification is sound in Section 3.3 and extend the analysis to a general lattice in Section 3.4. Finally, in Section 3.5 we present the results of the tool's application over the entire system, and conclude with Section 3.6.

3.1 Motivation

Systems that operate our cars, control our hearts, and are used in avionics, military, and banking require security assurances far above the norm. Automotive systems have been shown to be vulnerable to attacks [46, 77], and Koscher et al [42] show how an attacker who is able to infiltrate an Electronic Control Unit (ECU) can leverage this ability to completely circumvent safety-critical systems. Medical devices such as a modern implantable defibrillator is also shown to be vulnerable to unauthorized communication, potentially harmful device reprogramming, and unauthorized data extraction [32], threatening patients' health and safety conditions. Further, in the current design of the Boeing 787 aircraft system, the trusted aircraft control network shares the physical ARINC bus with the untrusted passenger network [26]. The ability to protect the integrity and

confidentiality of certain critical programs will form a crucial capability on which future high assurance automobile, medical, and avionics devices will be built.

Information Flow Policies: Information flow analysis targets security properties such as confidentiality and integrity. These properties can be modeled with an information flow control lattice ($\mathcal{L}$, $\sqsubseteq$), where $\mathcal{L}$ is the set of security labels and $\sqsubseteq$ is the partial order indicating relative trust or secrecy of the labels [25]. Security *labels* (such as trusted and untrusted, or secret and unclassified) are assigned to each input and output in the system, and we seek to determine whether it is possible under any scenario for the information flow policy to be violated (e.g., secret data flowing to an unclassified output). This model of security can be used to determine if policies such as non-interference [31] (where untrusted information must never affect trusted outputs) and Bell-LaPadula [16] (where secrets can never leak down, but unclassified information can be read up) are enforced.

While strict adherence to information flow policies alone is not sufficient to ensure the trustworthiness of a system, information flow policies are certainly necessary. For example, in the case of cryptographic operations, while there is clearly information flowing from the key to the encrypted data, we also need to show the key is not flowing anywhere else (e.g., through a timing channel [39]).

Our Threat Model: We consider a system with a set of labeled inputs (e.g. trusted and untrusted) and a set of labeled outputs – for example a software

router in a mixed-level trust network running on board an aircraft. The system is composed of a set of different software modules, some of which are known at analysis time and some of which are not, all running on the same processor. The attacker is assumed to have complete control of all untrusted inputs to the device. The attacker is further assumed to have full control over the subset of software running on the system that is unknown at analysis time. Given an information-flow policy defined over the input and output from this device (described as a lattice over the labels), the attacker attempts to manipulate the inputs, outputs, and unknown software in such a way that the policy is violated. In our examples in the paper, this would mean a transfer of information from an untrusted (or secret) input to a trusted (or unclassified) output. This threat model includes timing channels, implicit flows, storage channels, and any other digital forms of information flow, but does not include the use of physical phenomena such as EM emission or power draw. A system is said to be *strictly* enforcing a policy if it can be shown that the policy can *never* be violated regardless of the actions of the attacker.

Formal Verification of Systems: One very popular approach to reducing state to be verified is to analyze a higher level model of the system. Many formal methods have been proposed towards this end. The seL4 project verifies all of the C code for an entire microkernel [37], while Verve [80] moves the verification

another step down to the assembly code, and verifies the safety of the operating system using a typed assembly language. The FLINT project even aims at providing a framework for certifying machine code by producing a proof for an executable program [27]. However, all the above work verifies functional correctness, type safety and memory safety of the operating system: none of them is able to detect covert channels and malicious implicit flows beneath the system, i.e. in the hardware.

Extending the formal methods to handle the hardware channels present in real systems has proven quite difficult. Gianvecchio et al. [30] propose an entropy-based approach to detect covert timing channels based on the observation that the creation of a covert timing channel has certain effects on the entropy of the original process. Hu et al. [74] describe a solution called fuzzy time to reduce the bandwidths of covert timing channels by making all clocks available to a process noisy. Techniques for general black-box mitigation of timing channels are also well studied along with a general class of timing mitigators that can achieve any given bound on timing channel leakage, with a trade-off in system performance [12]. Karger, in a retrospective on the VAX-VMM security kernel [36], explains the difficulties of modeling hardware covert channels at higher layers and mentions that while they attempted formally address these issues, in the end these "were done on an informal basis by engineers by closely studying system design" and

that "timing channels proved a much more serious problem ... because many of them were inherent in the underlying hardware". In the end their solution for timing channels was to "fuzz" sources of clock information and thus lower the bandwidth of these channels.

Formal Methods for Information Flow Security: While information flow tracking has been proposed at many levels of the computing hierarchy, from virtual machines [34], high-level languages [62, 56, 14] and compilers [79, 47, 15] to binary analysis tools [7, 20, 28] and even hardware-assisted information flow tracking systems [67, 71, 24, 72, 61], formal approaches for the same have been confined to language-level and operating system-level proposals. Approaches that operate at the language level can even track implicit flows due to branches and loops that introduce observable variations in a program execution. Since code that is never executed can leak information (by the absence of its execution), some secure languages eliminate conditional behavior from the program code (either entirely [49] or based on confidential or untrusted conditionals [62]). A complementary approach to the above techniques is to control information flow through operating system abstractions such as processes, pipes, file systems, etc. [59, 44, 4, 13].

However, none of these approaches track covert flows through architectural features that are hidden beneath the hardware-software interface (i.e., the processor's instruction-set architecture), and the timing channels created through

shared architectural resources like caches, branch predictors, functional units or even processor cores. Determinator, for example, uses instruction counts as an indirect method of measuring time [13]. Such covert channels have received isolated attention, for example methods to build secure caches [76] or branch predictors [8], but general design methods and tools are lacking. Further, without an automated method of formal analysis, some proposals were subsequently found to be vulnerable [40] and then fixed [41].

Information Flow Tracking at the Gate Level: While the approaches discussed above are very valuable, many of the most subtle and exploitable information channels require that we peer far below even architecture-level specifications of a design to where the timing of the machine is actually well defined, i.e., the level of logic gates. Higher-level infrastructures cannot provide a strong guarantee that the full system *implementation* adheres to desired access control and information-flow policies. For this reason, we build upon recent work on Gate-Level Information Flow Tracking (GLIFT) [70]. Based on the observation that *all* digital information flows are a function of *logical information flows* through the gates and wires in a circuit, GLIFT shows that a gate-level description of a processor can be automatically augmented with shadow logic gates that dynamically track the flow of information through the processor and can identify information leaks through explicit, implicit, and even timing channels. Execution Leases [69]

then builds upon this idea to present an architecture where the ISA allows programmers to explicitly control information flows with a form of hardware-bounded function call.

Limitations of Dynamic Gate-Level Information Flow Tracking: One downside of this approach is that shadowing every gate in the circuit adds a considerable hardware overhead over the original logic (up to 3X [69]). This analysis logic costs money to design, verify, and fabricate, takes up area in the final implementation, and uses more power during operation. None of these costs are acceptable in the embedded domain.

The other, more significant, downside is that the dynamic techniques can only identify information leaks on specific executions that require the entire system to be fully specified. Embedded system designers would often like to statically analyze a hardware/software system and determine if it conforms to a specific policy, such as non-interference, even when certain components are neither known a priori nor are trustworthy. In contrast to dynamic tracking methods, our technique allows properties of gate-level descriptions of systems to be verified *statically* even when parts of the system are not known.

Finally, prior works on Gate-Level Information Flow Tracking formulate the label propagation algorithm informally for a two-level lattice. We generalize the

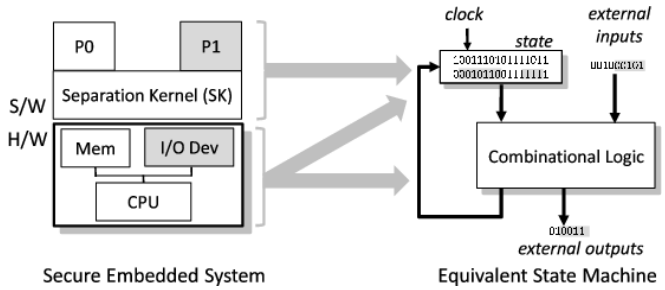

Figure 3.1: A typical embedded system can be flattened into a giant state machine whose state comprises of all software (including kernels and processes) and all hardware state (such as general purpose register-files, internal pipeline registers etc.). The state machine also includes some combinational logic that uses the current state and external inputs to generate next state and outputs. Analyzing information flows for all possible states reachable by this state machine allows us to verify the entire system with respect to a security policy.

algorithm to propagate labels for an arbitrary lattice so as to enable its use in realistic systems that implement Multiple Independent Levels of Security (MILS).

Finally, the techniques we propose for embedded systems make information-flow analysis an integral and automated part of the design flow. Without a method of providing feedback to the hardware designers about problematic information flows, we are stuck in a rut trying to model designs and argue that they are secure, rather than making information flow verification a measurable design constraint from the very beginning.

3.2 $*$-Logic (Star Logic)

In this section we present an overview of our verification method and show how it fits into existing hardware design flows. Given a system in which the hardware

and software cooperate to enforce a specific policy, we want to verify that the measures taken by a small, known portion of the software will be sufficient to prevent unauthorized labels from ever appearing at memory locations and output ports. As Figure 3.1 shows, at the gate-level, all digital hardware systems look relatively similar – a set of internal state (in the form of registers and memory), inputs and outputs (on I/O ports), and some combinational logic that reads the state and inputs and produces the next state (back to registers and memory) and the output (to I/O ports). At the hardware level any software that is running on this system is simply a set of bits in the memory. While certain bits of the state (e.g., the separation kernel) need to be concrete values, we want to verify the final properties of the system for all possible settings of the other bits (e.g., processes and external inputs).

3.2.1 High Level Description of $*$-logic

Figure 3.2 describes how our technique leverages existing design flows to aid in the verification of information flow properties. A user begins with a full digital design of a system (or a portion of a system), usually in some form of HDL. Those regions of the system that are not complete, for example particular regions of memory corresponding to as-of-yet unwritten software, or certain external components of hardware, are filled in with simple placeholder values which will be

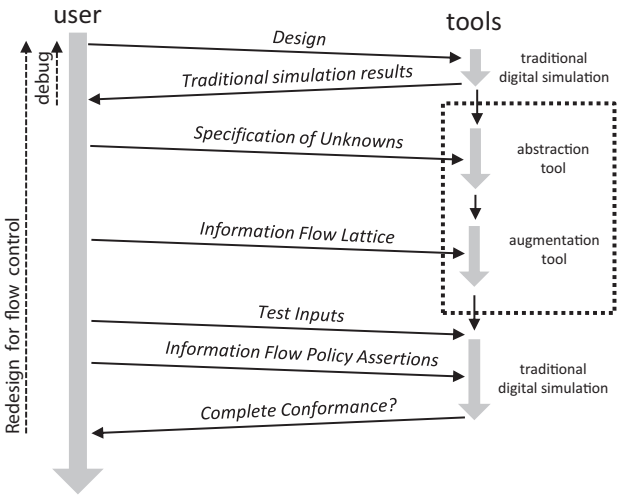

Figure 3.2: Our toolchain for verifying information flow properties of embedded systems. Once the design has been debugged using conventional tools, our abstraction and augmentation tools create a new design that operates on security labels and unknown values in addition to traditional digital values. This augmented design can then be simulated using standard hardware simulators to generate output labels. These are then compared with labels specified by the desired information flow policy to determine if the design conforms to the policy.

replaced for analysis. This base HDL description is compiled down to a gate-level description by existing design tools and tested through traditional methods.

The first step of the tool, abstraction, takes two inputs: the results of the above design synthesis and a specification of which bits are "unspecified" (represented as $*$) and hence could be either a 0 or 1. The output of this first step is one abstract state, in the form of a digital logic design that operates on an encoding of $\{0, 1, *\}$, that represents all possible initial states the concrete system can be in.

The second step in the tool, augmentation, takes two inputs as well: the abstract design from the first step, and an information flow lattice (such as Trusted $\sqsubset$ Untrusted), that specifies a set of labels and implies rules for how they they are to be propagated. This second tool then convolves the lattice with the abstract design to create a new design that operates on both abstract values and labels. The augmented logic, again being a hardware design itself, can then be simulated using existing hardware synthesis and simulation tools.

The final step in the process is to exhaustively simulate the resulting augmented system to check that it conforms to a specified information flow policy. To specify information flow policies the user has to initialize the inputs, outputs, and memory of the system with security labels such as secret/unclassified or trusted/untrusted that are allowed to be assigned to each bit. If an output of the system

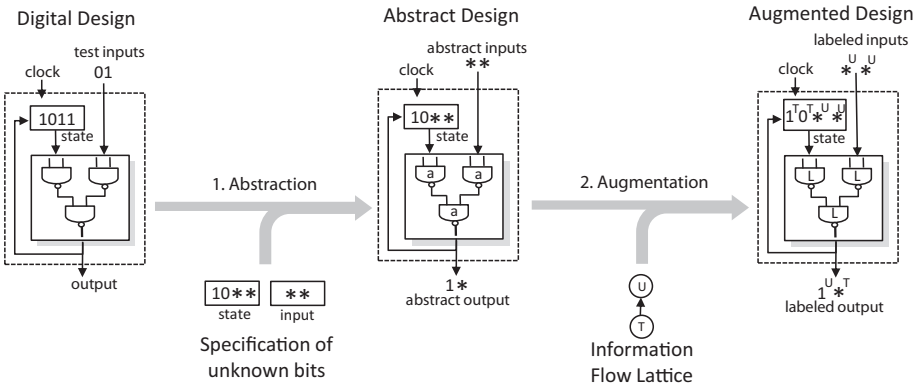

Figure 3.3: Given a concrete design to be verified and a specification of system state and inputs whose values are unknown, Step 1 of our technique automatically generates an *abstract representation* of the design that covers all possible concrete executions. In Step 2, the technique augments the abstract system with the state and logic to track the flow of security labels. The resulting *augmented system* is then synthesized and simulated using hardware design tools such as ModelSim and Altera Quartus. Augmented logic uses security labels specified by the user for inputs and memory bits and generates output labels that are compared with the desired output labels as specified by the information flow policy.

(or some state bit) is found to have an illegal label (e.g. trusted output is labeled untrusted), then the system under test has to be modified for correct information flow control. In this way dataflow assertions (such as "no classified information should egress the system via port 2") can be checked as standard logic assertions ("the label bit for out-port 2 should never be set to True"). Covering the set of logic states possible to ensure that this assertion holds is made far more tractable by the abstraction from step 1, and because the concretely specified system that comprises the trusted computing base only has a practically enumerable number of states (a scenario very common to high assurance systems).

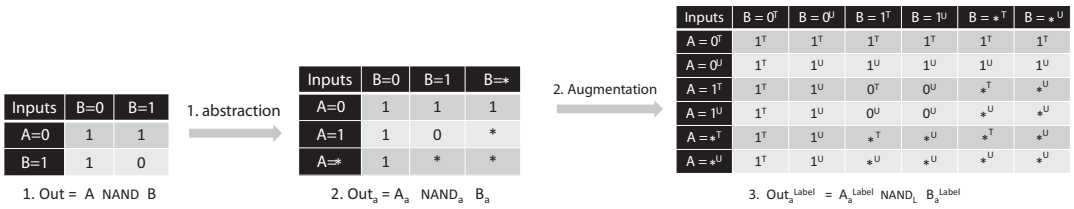

Figure 3.4: Generating information flow tracking logic for a 2-input NAND gate when the inputs can be unknown (∗): Figure shows the truth tables for a NAND gate (1), an abstract NAND gate (2), and the augmented NAND gate (3) that can track security labels trusted (T) and untrusted (U) through the abstract NAND gate. Each NAND gate in the original circuit will be replaced by an augmented NAND, which will be composed similar to the original NAND gates to create the complete augmented circuit.

3.2.2 Abstraction and Augmentation Details

Figure 3.3 shows a more detailed view of the two steps in our toolchain. We begin with the specifics of creating an abstract circuit that represents all possible concrete executions.

Step 1: Abstract Representation of a Digital System: To define an abstract system, we have to map both the concrete bits and logical functions to their abstract counterparts. To map concrete bits, the intuition is to map all the unknown bits in the given digital system to a ∗ in the abstract system. The bits in the concrete system whose values are known take the same values in the abstract system as they have in the concrete system.

To map the concrete logical operators such as NAND into the abstract domain, we create an abstract-NAND (NAND_a) that can operate on inputs valued 0, 1, or ∗. When all inputs are known, NAND_a functions just like the concrete NAND. If

any inputs are unknown ($*$), NAND_a determines if the $*$ inputs cause the output to be a $*$. Instead of simply marking an output as unknown when an input is $*$, we can use the precise definition of NAND to decide whether an $*$ input affects the output. In particular, if one of the inputs to a 2-input NAND is 0, then its output will be a 1 regardless of the other input value. This intuition can be captured in the definition of NAND_a so that given two inputs valued 0 and $*$, it still outputs a 1. Being precise in defining NAND_a and not outputting a $*$ whenever one input is a $*$ prevents the uncontrolled spread of $*$-values in the system.

Step 1 in Figure 3.4 shows the NAND_a logic described completely as a logical function derived from the definition of the concrete NAND. Each cell in the truth table of NAND_a is generated using multiple cells in the truth table of the NAND. For example, when $A = 0$ and $B = *$ for NAND_a, we look up cells in the NAND truth table with $A = 0, B = 0$ and $A = 0, B = 1$. Since the output of NAND is unchanged ($Out = 1$) for both values of B, B being $*$ is ineffective and the output of NAND_a is also a 1. On the other hand, when $A = 1$ and $B = *$, NAND_a's output is a $*$ since the value of B affects the value of the NAND's output.

This algorithm to generate an abstract logical operator can be generalized to any logical function, and we use it to generate abstract counterparts for small logical functions such as 2-input AND, OR, and MUX. However, since it requires checking whether any $*$ valued input affects the output value, it is exponential in

the number of inputs to the logical function. Thus, given a larger circuit expressed as a composition of smaller functions, we have to create a corresponding abstract circuit compositionally by replacing the concrete functions with their abstract counterparts and retaining the structure of the original circuit. As we will show in the next section, such a composition of abstract logical functions propagates the $*$ values in a sound fashion.

Step 2: Tracking Information Flows through the Abstract System: Consider a single two-input NAND gate. The problem of tracking the flow of information through this NAND gate is to decide, given both its inputs and their security labels, what the value and label of the output is going to be. The inputs can be assigned any values from $\{0, 1, *\}$, while the labels can be either trusted (T) or untrusted (U). Intuitively, information is said to flow from an input to an output if the input has some means of *affecting* the value of the output. This implies that the output of a NAND_a gate should be marked as untrusted only if some untrusted input can affect its value, and should be marked as trusted only if *no combination of untrusted inputs* can affect the value of the output. As compared to previous GLIFT works [70], we have to take *unknown* inputs into account to compute the effect of untrusted inputs.

As observed in Step 1, an input with a value 0 controls the output of a NAND gate irrespective of the value of the other input. This observation allows the

output of a NAND gate to be marked as trusted when one of its inputs is a trusted 0 even when the other input is an untrusted unknown $*^U$. On the other hand, if one input is either 1^T or a $*^T$, the other input can affect the output and its label will propagate to the output's label. This intuition of checking for effective inputs to determine the output's label can be described in terms of an algorithm that checks whether the output of the NAND_a gate changes *for any combination of untrusted and unknown values*, and marks the output as trusted only if none of the combinations alters the output value. This is shown as the second step in Figure 3.4.

Each gate in the original design is replaced with its corresponding augmented gate. These augmented gates are *for the purposes of analysis only*; these will never be actually fabricated, but will only be simulated in various design tools to help aid verification. Using the truth table for a NAND gate (the left-most table in Figure 3.4), we can construct an augmented truth table where the inputs can assume a value that is one of $\{0^U, 1^T, 0^U, 0^T, *^U, *^T\}$ (encoded respectively as $\{000, 001, 010, 011, 110, 111\}$), and the output is computed to be one of the same. The most significant bit in the new tuple of values that the gate will operate on is 0 if the original value is concrete or 1 if the value is unknown. The middle bit is the actual value if known and 1 otherwise. The least significant bit is 0 for U and 1 for T. Thus an n-input m-output digital gate is replaced by a gate with $3n$

inputs and produces $3m$ outputs. These augmented gates are then interconnected just as original gates of the system were.

To summarize, our technique takes a partial description of a system along with an information flow policy that specifies security labels for the initial system state, and statically verifies whether the system conforms to the policy for every possible state the system can ever be in, i.e. for all possible executions of the system. We verify this by representing all possible system states using an abstract system, simulating all possible executions by simulating the abstract system once, and verifying conformance to information flow rules for every abstract state.

3.3 Proof of Soundness

We consider the computing system to be a synchronous state-machine as in Figure 3.1 where the state is updated every cycle by some combinatorial logic. To prove that our technique tracks all flows for the entire execution, it is sufficient to prove that information flows are tracked soundly through arbitrary combinatorial logic operating on arbitrary inputs *for one clock cycle*. By induction on the number of clock cycles, the proof will apply to a system that executes for multiple cycles.

The first step is to prove that the abstraction step accounts for all possible executions of the concrete system that can arise from unspecified state. Figure 3.5 illustrates, for a system with two bits of state, the intuition behind why our abstraction is sound.

We can describe the current state of a particular concrete execution as a *configuration* that maps concrete state bits to values; we can then describe the states for all possible concrete executions as a set of configurations. We abstract these concrete configurations into a representation where each bit whose value can be both 0 and 1 across all configurations is treated as $*$, while bits that have a fixed value of 0 or 1 retain their value in the abstract domain. For example, a set of concrete configurations {01,11} yields an abstract configuration {$*$1} while {01,10} yields {$**$}. To prove that this abstraction is sound, we have to show that this abstract configuration represents a set of concrete configurations that is larger than or equal to the initial concrete configuration. In this case, while {$*$1} concretizes precisely into {01,10}, {$**$} represents the configuration set {00,01,10,11} and thus clearly over-approximates {01,10}.

More formally, we use the framework of abstract interpretation to prove the soundness of our abstraction mechanisms. The technique itself is fairly standard; the novelty lies in choosing an abstraction that captures only enough information to perform useful information flow analysis while also letting it scale to larger

designs. An object in the abstract domain is a compact representation of a set of concrete objects (or configurations). An *abstraction* function α maps sets of concrete objects to abstract objects and its inverse *concretization* function γ maps abstract objects into a set of concrete objects. By showing that α and γ form a *Galois Connection*, we can reason about properties of concrete objects by manipulating their corresponding abstract objects. To establish a Galois connection, α and γ have to fulfill the following three conditions. Representing concrete objects as $\sigma \in \Sigma$, abstract objects as $\sigma_a \in \Sigma_a$, and a set of concrete objects as $\overline{\sigma}$:

- α and γ are monotonic
- $\gamma \circ \alpha(\overline{\sigma})) \supseteq \overline{\sigma}$
- $\alpha \circ \gamma(\sigma_a)) \sqsubseteq_a \sigma_a$

Once a Galois Connection has been established, we can compute the abstract semantics systematically using only the concrete semantics and the Galois Connection. In particular, an abstract operator f_a defined as $\alpha \circ f \circ \gamma$ can be proven to be a sound abstraction of a given concrete operator f (Lemma 4.42 in [54]), and we use this result to compute abstract counterparts for a NAND gate.

The second step in our proof is to show that the algorithm we employ for propagating labels through combinational circuits conforms to the formal notion of non-interference, as defined in prior works on language-based information flow

security. By first creating an abstract circuit and then tracking information flows through the abstract circuit, we are able to guarantee that in the end, our technique is able to track all flows for all possible system executions.

3.3.1 Proof of Soundness of Abstractions

We will begin by formally defining the concrete and abstract domains and some helper functions, and use these to establish a Galois Connection between the domains. Once established, the Galois Connection will provide us with an abstract NAND operator that is provably sound with respect to the concrete NAND.

The concrete domain is defined as follows. Let c_i be a concrete boolean variable that represents one bit of state (memory or register), external input, or output of the digital system under test, and C be the set of all c_i. We define σ as a store that returns the current value of a concrete variable, i.e. $\sigma : C \mapsto \{0, 1\}$, and represents an object in the concrete domain. Entries in σ look like $[c_1 = 0], [c_2 = 1]$ and so on. The *collecting semantics* for the concrete domain is defined as the set Σ of σ generated by all possible executions of the concrete digital system. $\mathcal{P}(\Sigma)$ together with the subset operator $\subseteq$ form a complete lattice, called the *concrete lattice*.

The abstract domain is defined in a similar fashion. Let a_i be the abstract counterpart of c_i in the abstract domain and A be the set of all a_i. We define σ_a as a

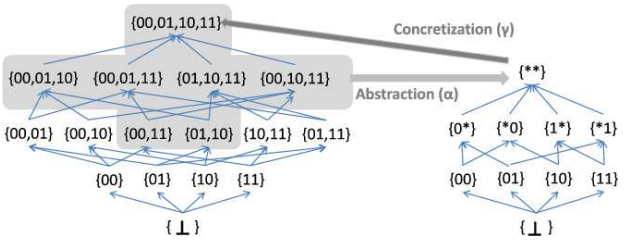

Figure 3.5: Figure shows how a set of concrete objects are mapped conservatively to an abstract object for a 2-bit system. A concrete object {00,01} represents all possible values concrete bits are allowed to have for some system, and is mapped to an abstract object {0,∗}. In the figure, the shaded concrete objects all map to the abstract object {∗,∗}, which when concretized yields a superset of all the shaded objects. The abstraction is sound because abstracting some concrete objects followed by concretizing the resultant abstract object will always yield a more conservative set of concrete objects.

store that returns the current value of an abstract variable, i.e. $\sigma_a : A \mapsto \{0, 1, *\}$, and Σ_a as the set of all possible σ_a. The ordering relation on abstract stores is $\sqsubseteq_a$. $\sigma_a 1 \sqsubseteq_a \sigma_a 2$ iff $\forall i$, either $\sigma_a 1(i) = \sigma_a 2(i)$ or $\sigma_a 2(i) = *$. Σ_a and the ordering operator $\sqsubset_a$ also form a lattice, and $\bigsqcup_a$ is used to represent the join operator in the abstract lattice. Also, note that throughout this section we let the subscript i range over all entries in a concrete or abstract store.

To help map sets of σ into the abstract domain, we define a projection function on σ, $\sigma(i)$, that returns the entry for the i^{th} variable . Another projection function $\sigma(c_i)$ extracts just the value of the i^{th} variable. To map *one* concrete object σ into its corresponding abstract object σ_a, we define a function β as follows.

Definition 1. (Beta function) *β is defined as $\beta(\sigma) = \sigma_a$, where $\sigma_a(a_i) = \sigma(c_i)$, $\forall i$.*

Definition 2. (Abstraction function) $\alpha : \overline{\sigma} \mapsto \sigma_a$ *is defined as* $\alpha(\overline{\sigma}) = \bigsqcup_a(\beta(\sigma_i)),\ \forall \sigma_i \in \overline{\sigma}$.

A helper function δ is used to concretize one entry in σ_a into a set of entries belonging to concrete stores.

Definition 3. (Delta function) $\delta(\sigma_a(i)) = \{[c_i = 0], [c_i = 1]\}$ *if* $\sigma_a(a_i) = *$, $\{[c_i = \sigma_a(a_i)]\}$ *otherwise.*

Since abstract variable with value $*$ concretizes into both 0 and 1, concretizing a 0|1-valued abstract variable always yields a subset of concretizing an abstract variable with value $*$:

Lemma 1. $\delta([a_0 = 0]) \subset \delta([a_0 = *])$ *and* $\delta([a_0 = 1]) \subset \delta([a_0 = *])$.

Definition 4. (Concretization function) $\gamma : \sigma_a \mapsto \overline{\sigma}$ *is defined as* $\gamma(\sigma_a) = \times(\delta(\sigma_a(i)), \forall i)$, *where* $\times$ *is the cartesian product operator.*

Lemma 2. α *is monotone, i.e. for sets of concrete objects* $\overline{\sigma_1}$ *and* $\overline{\sigma_2}$, $\overline{\sigma_1} \subseteq \overline{\sigma_2} \Rightarrow \alpha(\overline{\sigma_1}) \sqsubseteq_a \alpha(\overline{\sigma_2})$.

Proof. $\overline{\sigma_1} \subseteq \overline{\sigma_2} \Rightarrow$ there may be a σ, s.t. $\sigma \notin \overline{\sigma_1}$ and $\sigma \in \overline{\sigma_2} \Rightarrow \bigsqcup_a(\beta(\sigma_{1i}),\ \forall \sigma_{1i} \in \overline{\sigma_1}) \sqsubseteq_a \bigsqcup_a(\beta(\sigma_{2i}),\ \forall \sigma_{2i} \in \overline{\sigma_2})$, since $\bigsqcup_a$ is monotonic $\Rightarrow \alpha(\overline{\sigma_1}) \sqsubseteq_a \alpha(\overline{\sigma_2})$ (by definition of α)

□

Lemma 3. *γ is monotone, i.e. $\sigma_{a1} \sqsubseteq_a \sigma_{a2} \Rightarrow \gamma(\sigma_{a1}) \subseteq \gamma(\sigma_{a2})$.*

Proof. $\sigma_{a1} \sqsubseteq_a \sigma_{a2} \Rightarrow$ there may be an a_i, such that $\sigma_{a1}(a_i) = 0|1$ and $\sigma_{a2}(a_i) = * \Rightarrow$ from Lemma 1, $\delta(\sigma_{a1}(i)) \subseteq \delta(\sigma_{a2}(i)) \Rightarrow \times(\delta(\sigma_{a1}(i)), \forall i) \subseteq \times(\delta(\sigma_{a2}(i)), \forall i) \Rightarrow \gamma(\sigma_{a1}) \subseteq \gamma(\sigma_{a2})$.

□

Lemma 4. $\gamma \circ \alpha(\overline{\sigma}) \supseteq \overline{\sigma}$

Proof. Let $\overline{c}$ be the set of concrete variables c_i such that $\exists \sigma_1, \sigma_2 \in \overline{\sigma}$ where $\sigma_1(c_i) = 0$ and $\sigma_2(c_i) = 1$. Also, let $\sigma_a = \alpha(\overline{\sigma}) = \bigsqcup_a(\beta(\sigma_i))$. Then $\gamma \circ \alpha(\overline{\sigma}) = \gamma(\sigma_a) = \times(\delta(\sigma_a(i)), \forall i)$. It is clear that this cartesian product will generate $2^{(|\overline{c}|)}$ concrete objects, while the initial set of concrete objects $\overline{\sigma}$ may only have a subset of these combinations (the join of whose abstractions creates a $*$ in the abstract state). Thus, $\gamma \circ \alpha(\overline{\sigma}) \supseteq \overline{\sigma}$.

□

Lemma 5. $\alpha \circ \gamma(\sigma_a) \sqsubseteq_a \sigma_a$

Proof. We show the slightly stronger condition that $\alpha \circ \gamma(\sigma_a) = \sigma_a$. $\alpha \circ \gamma(\sigma_a) = \alpha(\ \times(\delta(\sigma_a(i)),\ \forall i) =$

$\bigsqcup_a(\beta(\times(\delta(\sigma_a(i)),\ \forall i))) = \sigma_a$. Since the cartesian product creates all possible concrete states, the join of their abstractions results in the initial abstract state.

□

Theorem 1. (Galois Connection) $\Sigma < \alpha, \gamma > \Sigma_a$ *form a Galois Connection.*

Proof. Follows from the definition of Galois Connection and Lemmas 2,3,4 and 5. □

We have established a Galois Connection between the concrete boolean domain and an abstract domain. To complete the mapping from concrete to the abstract domains, we define an abstract transfer function NAND_a to soundly capture the effect of a given concrete transfer function NAND. We use NAND since any arbitrary combinational circuit can be represented as a composition of NAND gates.

The Abstract NAND gate: The concrete semantics of a NAND gate can be formalized through a truth table that represents the usual NAND logic function. The collecting semantics of the NAND gate can then be defined as $\text{NAND} : \mathcal{P}(\mathcal{B}) \times \mathcal{P}(\mathcal{B}) \mapsto \mathcal{P}(\mathcal{B})$ by applying the concrete NAND gate pairwise to each member of the input set to generate the output set. Here, since the concrete domain is defined in terms of Σ, the collecting semantics are expressed as $\text{NAND} : \Sigma \text{ x } \Sigma \mapsto \Sigma$. The most precise abstract operator that is sound with respect to NAND follows from the Galois connection:

Theorem 2. *(Soundness of Abstract NAND)* $NAND_a : \sigma_a \times \sigma_a \mapsto \sigma_a$ *defined as* $NAND_a = \alpha \circ NAND \circ \gamma$ *is a sound approximation of NAND.*

The procedure to compute NAND_a presented in Figure 3.4 closely follows the mathematical definition of NAND_a above, where given a truth table for a NAND gate, a new truth table is generated to encode the NAND_a function. First the abstract inputs are concretized (so that a $*$ input generates both concrete 0 and 1). Then the concrete NAND is applied to all the input sets, so that the output set will include the effect of all the generated concrete inputs. Finally, the concrete output set is abstracted to yield the abstract output bit. This abstraction step effectively ensures that, if the generated concrete inputs cause the output to be both 0 and 1, then the abstract output will be a $*$. Note that the signature of NAND_a uses σ_a instead of Σ_a, as NAND_a operates on individual abstract states instead of sets of abstract states.

Finally, any combinational circuit can be expressed as a partially ordered sequence of 2-input NAND operations.

Theorem 3. *(Soundness of Composition) For every concrete trace* $NAND^i(\sigma_0), i \geq 0$*, there exists an abstract trace* $NAND_a^i(\sigma_{a0})$*,*

$i \geq 0$*, such that* $\forall i$*,* $\alpha(\sigma_i) \sqsubseteq_a \sigma_{ai}$ *where* $\sigma_i = NAND^i(\sigma_0)$ *and* $\sigma_{ai} = NAND_a^i(\sigma_{a0})$*.*

Proof. Given a Galois Connection $\Sigma < \alpha, \gamma > \Sigma_a$ and an abstract operator NAND_a that is sound w.r.t. the concrete operator NAND, the proof is by strong induction on the length of the trace.

□

A similar proof applies to the truth tables of other abstract logic functions such as 2-input AND, OR, and MUX, and we use such verified modules in our prototype tool. We have shown that the abstract machine represents the entire set of feasible executions, because for each output and for every clock cycle, the abstract logic reads in, operates on, and updates the abstract state and inputs/outputs. In the next section, we show how tracking information flows for this abstract representation allows us to formally prove non-interference for the entire family of executions that the abstract exection represents.

3.3.2 Proof of Tracking Non-Interference

In this section, we prove that our technique verifies non-interference for the abstract system. The technique associates security labels with each bit of abstract state to create augmented states, and introduces new logic to track labels through the abstract circuit. An example of such augmented logic for an abstract NAND gate is generated as the result of Step 2 in Figure 3.4.

More precisely, each abstract store σ_a is transformed into an augmented store σ_l that maps variables to values in $\{0, 1, *\} \times \{T, U\}$. The given combinational logic functions map σ_l to a new value σ'_l, and is treated as a sequence of 2-input NAND operations on σ_l. For the purposes of this proof, we include every wire in the circuit as a part σ_l, so that σ_l includes all inputs to intermediate NAND gates.

We also introduce the notion of T-equivalence of two augmented stores, such that $\sigma_{l1} \sim_T \sigma_{l2}$ if both σ_{l1} and σ_{l2} agree on all trusted values. Our non-interference theorem based on prior work on type-based information flow security [65]. It formalizes the intuition that two stores that begin by agreeing on all trusted state, after executing the same abstract operation, will still agree on all trusted state. Effectively, the untrusted inputs had no affect on the trusted state. Further, operating at the gate-level ensures that we explicitly track all information flows, including timing channels [70].

Theorem 4. *(Non-Interference) Given an augmented logic function f_l, and stores σ_{l1} and σ_{l2}, $\sigma_{l1} \sim_T \sigma_{l2} \Rightarrow f_l(\sigma_{l1}) \sim_T f_l(\sigma_{l2})$*

Proof. The proof uses strong induction on the number of levels in the partial ordering of NANDs. The initial stores σ_{l1} and σ_{l2} have state/external input values derived from the user specification, and the output bits set to $*^U$.

Base Case: For n = 1, the system is a two input NAND_l gate generated after Step 2 in Figure 3.4. Since the stores are initially T-equivalent, and only the output is assigned a value by the NAND_l gate, T-equivalence of σ'_{l1} and σ'_{l2} depends on whether the outputs get identical values in σ'_{l1} and σ'_{l2} *whenever either one is labeled trusted*. If both *out*s are labeled untrusted, even then $\sigma'_{l1} \sim_T \sigma'_{l2}$. Using $in0$ and $in1$ to refer to the inputs of the NAND gate, which could come

from either state or external inputs, and *out* to refer to its output, the proof that the final stores σ'_{l1} and σ'_{l2} are T-equivalent has three cases:

(1) both $in0$ and $in1$ untrusted $\Rightarrow \sigma_{l1}(out) = \sigma_{l2}(out) = (1|*)^U$, i.e. some untrusted value (from NAND$_l$ truth table) $\Rightarrow \sigma'_{l1} \sim_T \sigma'_{l2}$.

(2) both $in0$ and $in1$ trusted: $\sigma_{l1} \sim_T \sigma_{l2} \Rightarrow \sigma_{l1}(in0) = \sigma_{l2}(in0)$ and $\sigma_{l1}(in1) = \sigma_{l2}(in1) \Rightarrow \sigma_{l1}(out) = \sigma_{l2}(out)$ (and *out* is trusted, from NAND$_l$ truth table) $\Rightarrow \sigma'_{l1} \sim_T \sigma'_{l2}$.

(3) one of the inputs (e.g. $in0$) is trusted and the other ($in1$) untrusted: $\sigma_{l1} \sim_T \sigma_{l2} \Rightarrow \sigma_{l1}(in0) = \sigma_{l2}(in0) \Rightarrow$

(Case A) $in0$ is 0: $\sigma_{l1}(out) = \sigma_{l2}(out) = 1^T$ (from $NAND_l$ truth table) $\Rightarrow \sigma'_{l1} \sim_T \sigma'_{l2}$.

(Case B) $in0$ is 1: $\sigma_{l1}(out) = \sigma_{l2}(out) = (1|*)^U$, i.e some untrusted value (from NAND$_l$ truth table) $\Rightarrow \sigma'_{l1} \sim_T \sigma'_{l2}$.

Inductive step: Assume that the stores σ_{l1} and σ_{l2} are T-equivalent for partial orders of NANDs with levels $\leq$ n. Thus an $(n+1)^{th}$ level NAND gate receives input stores that are T-equivalent. Since the proof for the base case works for arbitrary values of the initial stores, following the same reasoning as the base case for the $(n+1)^th$ NAND gate, $\sigma'_{l1} \sim_T \sigma'_{l2}$ after the (n+1)th step.

□

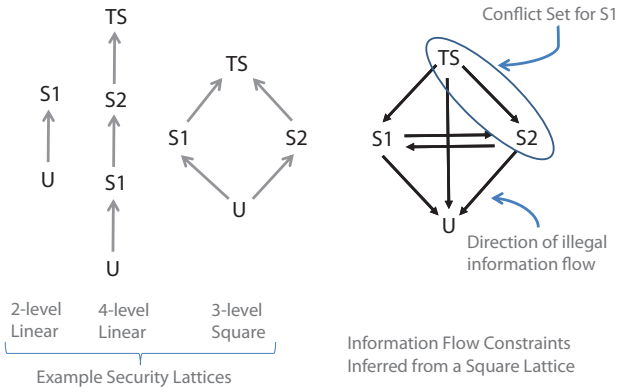

Figure 3.6: Security labels expressed as lattices: Figure on the left shows three example lattices, two linear and one square, that can be created using four labels, Unclassified (U), Secret (S1 and S2), and Top Secret (TS). Figure on the right shows the graph of information flow constraints obtained from a square lattice. The graph shows, for example, that information should not flow from labels S2 and TS to S1 in order to ensure secrecy.

3.4 Information Flows through a Lattice

Thus far, we have explained the $*$-logic technique using a simple, two-level lattice, e.g. `Trusted` $\sqsubseteq$ `Untrusted`. In this section, we present an algorithm to generalize gate-level information flow analysis to handle *arbitrary information flow policies*. While a simple system may only have two security levels, many systems have more complex information flow policies such as `Unclassified` $\sqsubseteq$ `Secret` $\sqsubseteq$ `Top Secret`. The main challenge in extending a two-level information flow analysis to an arbitrary lattice is in propagating a label when the input labels are non-comparable with each other, e.g. `Air force` and `Navy` in the lattice

`Unclassified` $\sqsubseteq$ `Air force`, `Navy` $\sqsubseteq$Top Secret. We present a method that tracks labels more precisely than conservatively assigning the Least Upper Bound of the input labels to the output.

Figure 3.6 shows a few example security lattices on the left, where labels Unclassified (U), Secret (S1 and S2), and Top Secret (TS), are arranged in either a linear or a diamond lattice. Conventionally, given two labels in a lattice, the output is labeled conservatively with the Least Upper Bound of all the input labels. This assumes that all inputs could be potentially effective. *Precision* in label propagation, on the other hand, requires that the output have the *least conservative* label that does not allow any illegal information flows.

Our insight for propagating labels precisely through a boolean function is to use the input values and the boolean function itself to determine a *Candidate Set* of labels that are all information flow secure assignments, and then choose the least conservative label from this set. As an example, in order to be information flow secure when an output is labeled $S1$, we have to check whether the only inputs that can affect the output should have labels that are dominated by $S1$ in the lattice. In case there is more than one non-comparable label in the Candidate Set, we show that it is information flow secure to choose any one of them to be propagated.

To describe how we determine candidate labels, we first define the *Conflict Set* of a label. The conflict set for a label includes labels that are either more conservative or labels that are mutually unordered with the label under consideration. Figure 3.6, on the right, shows a directional graph where the edges represent constraints on information flows for a square three-level lattice. In this graph, we refer to the set of labels that have edges directed towards a label as its conflict set (for example, S2 and TS for S1 in Figure 3.6). The edge from S2 to S1 indicates that S2 should be isolated from S1, and the edge from TS to S1 indicates that no information should leak down from TS to S1. For the purposes of non-interference, once an output is assigned a label, no input with a label belonging to the assigned label's conflict set should be able to affect the value of the output.

Label-propagation algorithm for a general lattice: Algorithms 1 and 2 describe the complete procedure. The algorithm takes as inputs the logic function and the lattice of labels, and has to generate a *shadow logic function* that tracks information flow through the given logic function. This requires enumerating the entire truth table for the shadow logic function, and for each row of the shadow truth table, assign a precise yet safe security label from the lattice to the output.

In the first step, the algorithm generates inputs for the truth table of the shadow logic function by iterating over every possible assignment to the function's inputs and their labels.

Algorithm 1 This figure shows the main algorithm for propagating labels through "F" when the labels belong to a general lattice. For each row of the shadow truth table, the algorithm works by determining the lowest possible label such that no combination of inputs with a label above or incomparable with the chosen label can affect F's output.

```
procedure propagateLabel
input F : combinatorial logic function
input Labels : set of all labels
input Lat : lattice representation of all labels
input X : set of inputs to F with values in 0,1
input L : set of input labels with values in Labels
output Sh_F : truth table for shadow F
XSet ← Set of all possible assignments to X
LSet ← Set of all possible assignments to L
{Shadow truth table requires all combinations of
# X ∪ L}
for each XRow ∈ XSet do
  for each LRow ∈ LSet do
    Candidate_set ← φ
    {All labels are candidates for output label of
    the row}
    for each label ∈ Labels do
      {C : Set of conflicting inputs for label}
      C ← φ
      for each l_i ∈ LRow do
        if Lat.conflictsWith(l_i,label) then
          {x_i is the input corresponding to label l_i}
          C ← C ∪ {x_i}
        end if
      end for
      {Check if C affects F(X_row)}
      if not isAffectedBy(F, X_row, C) then
        Candidate_set ← Candidate_set ∪ {label}
      end if
    end for
    Sh_F_row ← Lat.ChooseMin(Candidate_set)
    Print X_row, T_row, Sh_F_row
  end for
end for
end procedure
```

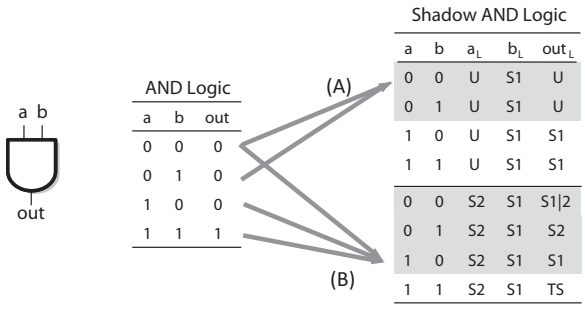

Figure 3.7: Tracking Information Flow through a 2-input AND Gate: Figure shows truth table for the AND Gate (left) and a part of its shadow truth table (right). The shadow truth table shows the interesting cases when both the inputs a and b have different labels (e.g. $a_L = U$ and $b_L = S1$). For the top half, each row of the shadow table calculates the label for the output (out_L) by checking whether the conservative input b can affect the output *out*. This requires checking *out* for both values of b in the table on the left. The gray arrows indicate the rows that have to be checked for each row on the right. The bottom half has two unordered labels, and so it has to check both labels as candidates for the output, and look up three rows from the AND truth table.

The second step executes for each row in the shadow truth table. The algorithm begins by considering *every* label in the lattice as a potential candidate label for the output. For this, the algorithm computes the current label's conflict set and uses these to find inputs that have labels belonging to this conflict set. The key idea is to check whether any combination of inputs that have conflicting labels can affect the value of the output. If the output is affected by some combination of such inputs, then the current label under consideration is *not* a valid candidate output label. The algorithm then moves on to examine the next label, until, in the worst case, the most conservative label is added to the candidate set.

Algorithm 2 This function is used in Algorithm 1, to checks if the value of a combinatorial function F is affected by a given subset of its inputs

```
procedure isAffectedBy
input F : combinatorial logic function
input X_row : set of input values
input C_row : a subset of inputs
func_row ← F(X_row)
{Toggle each combination of subset's elements, and}
{Check if F's output changes}
for each Comb ∈ power set of C_row do
  Comb_inv ← {c̄_i}, where c_i ∈ Comb
  func_inv ← F(Comb_inv ∪ (X_row − Comb))
  if func_inv ≠ func_row then
    return 1
  end if
end for
return 0
end procedure
```

Note that the most conservative label is always a candidate label, as information can flow from all labels to it (i.e. its conflict set is null). This ensures that the algorithm is guaranteed to assign at least one label for the output for every row in the shadow truth table. Having considered all labels, the algorithm will output a candidate set of labels that are all safe to be assigned to the output. Also, note that the order in which labels are considered is not important.

In the final step, once a candidate set of labels is found, the algorithm will assign the output label most precisely by choosing a label from the candidate set that is the least conservative (or is lowest in the lattice). This choice depends on the two conditions below:

Handling Totally Ordered Labels: If one label in the candidate set is totally ordered and lesser than all the other candidates (i.e. there is a unique lowest candidate label), assign it as the output label.

Handling Mutually Unordered Labels: If there are multiple, mutually unordered labels in the candidate set (e.g. S1 and S2) that are lower than all other labels in the candidate set, then it is safe to choose either one as the output label. We will analyze this case in more detail.

Analysis of the Algorithm: For most cases, there is one label that is the lowest among the candidate labels. Multiple incomparable choices emerge when multiple non-comparable labeled inputs have no effect on the output. This occurs for example when both inputs a and b are 0 and have labels S1 and S2 from a square lattice (as in the first row of the lower half of the shadow AND truth table in Figure 3.7). For this situation, neither S1's conflict set by itself nor S2's conflict set by itself affects the value of the output, and hence both S1 and S2 could be assigned to the output label legally and belong to the Candidate Set.

Conventionally, given two inputs with labels S1 and S2, the output would be marked as Top Secret. Following the informal descriptions presented in prior dynamic gate-level information tracking [70], one may be tempted to label the output as Unclassified, since neither S1 nor S2 by itself affects the output. How-

ever, the *combination* of S1 and S2 definitely affects the output, and hence the informal treatment in [70] does not extend to a general lattice.

We propose that selecting any one of the lowest candidate labels as the output label, even though they are mutually unordered, ensures security while at the same time maintaining precision. Security is maintained because we have checked that the input's conflict set is ineffective, while not using Top Secret allows the output to remain at a lower security level.

To further understand why choosing any one label is secure, consider the following example. Add a new label S3 to the square lattice such that S1 dominates S3 while S2 is incomparable with S3. The logic function to be shadowed has two steps: in the first step, S1 and S2 inputs produce an intermediate output, which is later AND-ed with S3-labeled input to create the final output. The tricky case occurs when all three inputs individually do not affect the output, for example when the functions at each step are 2-input AND gates, and all three inputs are 0s. When we choose S1 after the first step, $0_{S1}\&0_{S2} = 0_{S1}.0_{S1}\&0_{S3} = 0_{S3}$. When we choose S2, $0_{S1}\&0_{S2} = 0_{S2}.0_{S2}\&0_{S3} = 0_{S3}$ or 0_{S2}. Finally, we end up with the output label being either a precise S3 or an imprecise (but safe) S2. In either case, the output was not labeled conservatively as Top Secret.

Finally, we would like to note that the algorithm presented here is not optimized for efficient execution time. Rather, it is deliberately independent of

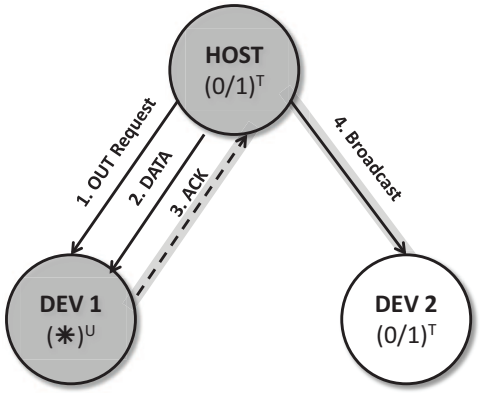

Figure 3.8: Information leaks in a USB system: Once the Host controller receives an ACK from the untrusted device, its state is untrusted. When it broadcasts next, the untrusted information spreads to all trusted devices on the bus.

performance optimizations such as traversing the lattice from bottom to top, and emphasizes how the security property is enforced.

3.5 Experimental Analysis

We will now apply $*$-logic to verify whether a shared Universal Serial Bus (USB) follows information flow policies.

3.5.1 Experimental Setup

Our particular test scenario works to demonstrate that we can in fact guarantee non-interference between devices on the USB bus. Specifically, our experimental setup consists a completely specified Host (with each bit $(0|1)^T$)) and 2 Devices. Device 1 is untrusted and unspecified and Device 2 is trusted and specified. We

want to guarantee that all unintended information flows are contained *for all possible values of untrusted inputs and state*, i.e. for any communication between Device 1 and the Host there should not be any untrusted information flowing to Device 2.

For representative information flow policies, we choose three security lattices. The first lattice L0 is the simplest two level lattice such as `Trusted` $\sqsubseteq$ `Untrusted`. The second lattice (L1) is a 4-level linear lattice (`Unclassified` $\sqsubseteq$ `Secret 1` $\sqsubseteq$ `Secret 2` $\sqsubseteq$ `Top Secret`, and the final one is a square lattice (L2) with four labels and 3 levels (`Unclassified` $\sqsubseteq$ `Secret 1`, `Secret 2` $\sqsubseteq$ `Top Secret`). These lattices cover the cases where labels are both strictly ordered and partially ordered, and also the case where there are at least three independent security levels, as required by many military applications.

Methodology

Our verification toolchain can analyze hardware designs written in behavioral Verilog or VHDL so that hardware designers can use their tools of choice for design entry. Verilog/VHDL designs are then synthesized using Synopsys Design Compiler into a gate-level description (netlist) using the `and_or.db` library. The result of this synthesis is a netlist that consists of just AND, OR, and NOT gates along with registers and memory. This netlist is input to our abstraction tool,

which replaces gates and bits of the netlist with their abstract counterparts and outputs the abstract netlist. The abstract netlist is then input to our augmentation tool that generates information flow tracking logic for the abstract design to create the final netlist. As both the final augmented design and the intermediate abstract design are full hardware system designs in and of themselves, each can be simulated using existing hardware synthesis and simulation tools such as ModelSim or Altera Quartus. Figure 5.8 shows the results of this synthesis process for the USB host and devices for each of the three tested lattices. We obtained these area and synthesis numbers using Synopsys Design Compiler and targeting the SAED 90nm technology library [1].

To conduct an experiment, we first initialize the system with the Master's state machine being instantiated as a set of abstract bits and logic, represented as 00 and 01 for concrete 0 and 1 respectively, and the shadow state bits marked as trusted (0). Device 1 is unknown and untrusted, hence each device bit is 11 to represent $*$ and each of its shadow bits is 1 to represent untrusted. Device 2 is known (so as to carry out a valid USB transaction during the test, but this is not required for the verification) and trusted. The information flow policy is encoded as the satisfiability question: "Does the shadow bit for the data to a trusted device ever become 1, i.e. untrusted?". To decide this, we simulate the

USB	Area (number of gates)				Verification Time (sec)			
	Base	L0	L1	L2	Base	L0	L1	L2
TDMA Host	692	6242	9392	11852	1.3	11.01	14.23	17.53
Device	472	4282	6580	8140	1.2	7.79	10.19	11.96

Figure 3.9: Verifying USB Host and Device Controllers for three label lattices: Size of each controller grows with lattice complexity (L0 to L2), but these augmented designs are for verification purposes only. It takes less than 30 seconds to synthesize and simulate the controllers through one time schedule of the TDMA schedule super-imposed on the shared bus.

abstract machine representation in ModelSim and stop when we have completed one loop of the USB traffic scheduler.

3.5.2 Experimental Results

We executed our test scenario, where a trusted known Host communicates with an unspecified and untrusted Device 1. Specifically, we have the Host perform a write transaction with the Device which consists of first sending an OUT request packet, followed by data, and then completed by a handshake acknowledgement packet from Device 1. This test scenario was executed and untrusted information from Device 1 does in fact flow to Device 2 even though they are not physically on the same bus. Since Device 1 is untrusted, the Host's entire state becomes completely untrusted when it attempts to receive an acknowledgement from Device 1. Subsequent transactions between the Host and Device 2 cause the Host's now untrusted state to flow to Device 2, making Device 2 untrusted as a result.

To solve this problem, it is critical to restore the Host back to a trusted state prior to communicating with Device 1. To do this, we implement a time-division multiple accesses (TDMA) scheme enforced by the Host following a *trusted* schedule. This TDMA based scheme arbitrates between untrusted and trusted states for Device 1 and Device 2 respectively. If the time slot expires when executing in an untrusted state, the state is restored back to a trusted one so that communication between Device 2 can be completed without any untrusted information flows.

Simulating the design for a complete TDMA time slot shows that in fact all unintended information flows are eliminated, and that no information leaks from Device 1 to Device 2. We simulate a complete transaction, where the Host completes a full round of communication with Device 1 with all transmitted values from Device 1 to the host being unknown. Once this transaction completes, the Host returns back to a trusted state. Thus on a subsequent transaction with Device 2, the data bit has a shadow value of 0 (trusted) and thus no information from the untrusted Device 1.

In the end, for a system with one Host and 2 Devices, we were able to verify the Host for one particular time schedule in less than 30 seconds for all lattices. In the process, we are only required to specify 692 of the total 1636 gates in the

system. Further, the amount of concretely specified state does not increase with the number of devices.

3.5.3 Discussion

As shown by our evaluation, one of the main advantages of $*$-logic is that it integrates directly within existing hardware design flows. The hardware designer has to only initialize the augmented design with some known firmware and the security labels, and after simulating the augmented system, check to ensure that no trusted output or memory region is labeled as untrusted.

$*$-logic can potentially scale well with increasing design sizes, as the size of the augmented design only increases linearly with the increasing design size. This linear growth is possible because $*$-logic relies on a very coarse abstraction, i.e. a bit can be 0, 1, or $*$, and tracks information flows at a very coarse granularity. Due to this, it requires the designs themselves to very carefully manage the flow of bits, and some such design techniques have been demonstrated by prior work in dynamic GLIFT [70]. Such techniques, proposed to control the flow of untrusted information, can also be used to manage the flow of unknown information, and in the end design a system that meets information flow policies even in the presence of unknown and untrusted bits.

Our primary future work is to integrate $*$-logic with more sophisticated verification tools that rely on finer grained abstractions (e.g. that understand integer arithmetic) to prove that a *set* of designs are secure. For example, $*$-logic requires instantiating a bus controller with a particular time slot for verification purposes. With finer grained abstractions, we could prove that the controller is secure for *any time slot value*.

3.6 Conclusions

Many high assurance systems are built around small components that are responsible for maintaining the integrity or confidentiality of the overall system. In this paper we describe how arbitrary information-flow properties of such hardware-firmware components can be statically verified using a set of digital design transforms and traditional design tools. Towards this end, we propose a new verification technique that abstracts out all parts of a given system that is irrelevant for information flow security, enumerates all states of this abstract design, and verifies that a trusted signal is never affected by untrusted ones.

Embedded systems are trusted by people to do everything from stopping their cars to controlling the beating of their hearts, yet all too often these systems are designed with security as an afterthought at best. Our hope is that by transforming

security constraints into functional verification constraints these types of problems will be more understandable and tractable to practicing hardware/software engineers. In this paper we have taken a tangible step towards this vision with respect to information flow security, however this cannot be the end of the story. More work is needed to understand how cryptographic and other label-modifying functions can be properly integrated, how these hardware/software-level methods can be more suitably married to the many powerful language level techniques already known, and how engineers may most effectively be informed of security issues early in their design process, just to name a few. However, even with these open problems, we believe the tools developed herein are useful in and of themselves as demonstrated by our work verifying a USB bus controller.

Chapter 4

Execution Leases: A Hardware-Supported Mechanism for Enforcing Strong Non-Interference

We introduce *Execution Leases*, an architectural mechanism that makes information flows explicit to the programmer, including timing, covert and implicit flows through control/architectural state. The basic idea behind a lease is that control of a portion of the machine is given over to an untrusted entity for a fixed amount of time and within a fixed range of addresses. After the lease expires, control is yanked back to the trusted code and any remnants of the untrusted actions are purged from the critical machine state such as the PC (registers and main memory are not part of the critical machine state and retain their values and their security labels even after a lease expires). The hard part is performing this operation with reasonable overheads and in a way that can be demonstrated to be

correct through inspection of the gate-level implementation alone. Because of the relative freedom provided by Execution Leases (as opposed to the past approach of ensuring there is never *any* way for untrusted or sensitive code to effect the program counter), the resulting code can be a 100x faster in some cases, several factors smaller, and far easier to program. Specifically, our contributions include:

- The introduction of Execution Leases, both as a programming model in the abstract and as implemented by a specific ISA for high assurance systems.
- New methods for verifiable architectural information-containment by design, including a description of the various microarchitecture modifications needed to bound information flows in our processor.
- The evaluation of a complete Execution Leases implementation, including a new Lease-based ISA, a small language and compiler that target this ISA, a fully synthesizable prototype, a complete gate-level information flow analysis of the final design, and results from experiments with several hand-written applications.

We begin with the specifics of the architecture, along with details of its implementation and application, in Sections 4.1 and 4.2. Following this, we describe experimental results in Section 4.3 and conclude with Section 4.4.

4.1 Architecture

To understand the reasons behind Execution Leases, we begin by describing the many ways in which information can leak in a traditional processor, and the lengths that the original GLIFT processor went to in order to prevent them.

4.1.1 The Problem with Overprotecting Critical State

Constructing an efficient architecture that can strongly contain the flow of information, yet still maintains a good level of programmability is difficult, and the philosophy for dealing with this problem in the original GLIFT work was simply to ensure that critical machine state could never become tainted. While this sounds straight forward, it is quite a bit harder than it sounds. It means that the architecture has to be constructed in such a way that it is *impossible for any data* in the system (which could then turn out to be tainted) to *ever* effect the program counter, the instruction memory, or the address of a store. If any of these were to be tainted, the entire state of the machine would quickly (in one or two cycles) end up marked as tainted, and there would be no way to undo that damage.

As an example: the original GLIFT architecture ensured that it was impossible for the program counter to ever be influenced by the the result of some

computation (and thus risk being tainted). This, of course, means no conditional jumps of any sort, and in fact ensures that programs will always execute a set fixed number of instructions at every invocation. While the machine would no longer be Turing-complete, in many cases the complete lack of conditional jumps could be compensated for by predication. Because predication transforms control dependencies into data dependencies, almost all of the instructions in the original GLIFT-enhanced architecture could be executed conditionally without ever effecting the program counter (with the obvious exception of jumps).

Likewise, the architecture had to prevent the execution of indirect loads and stores. Consider the information flow in the statement $\mathrm{M}[x] = 1$. In this statement, there is clearly a flow of information from x to $\mathrm{M}[x]$, but there is also a much more subtle *implicit* flow of information from x to $\mathrm{M}[y]$ where $x \neq y$. Why? Because by observing that $\mathrm{M}[y] \neq 1$, we have now gained some information about x. If a store is to be executed and the target address of the store is tainted, information flows from that store the *every single piece of memory in the system* (in other words every possible value of y). If we think about the flow of information at the gate level, this becomes very clear. The tainted bits of the address flow into the memory decoder, and as a result all of bit-lines are tainted. When the write actually happens, we have to assume that any of those bit-lines could have been active and thus the write could have happened to any of the possible memory

addresses. To deal with this problem, the original GLIFT architecture prevented any indirect loads and stores, instead enforcing that all of the loads and store used an address that was a constant offset from some untaintable counters (kept untaintable in much the same way as the PC was above).

The final, and perhaps most obvious step to keeping the core architectural state from becoming tainted in the original GLIFT architecture is to never allow the execution of *tainted code*. Tainted code, will always end up tainting the PC, the load and store address, and any other state effected by the execution of that code (i.e. everything), as shown in Figure 4.1.

The ramifications of this philosophy of never allowing any of the processor critical processor state to become tainted are enormous. For example to do a table lookup (a very common operation in AES and many other crypto algorithms), due to a lack of indirect loads, the original GLIFT architecture would have to loop over all of the entries and predicate out all of the loads that were not the one specific index that was to be looked up – a slow and code intensive prospect. It also lead to an inability to have functions, and to bound the effect of an execution of tainted code[1]. The approach we take in this paper is far less constrained, allowing all of this critical state to become tainted over bounded periods of time, while

[1]If you are considering a confidentiality policy where *secrets* are tainted rather than untrusted data, this ability would be particularly useful because it means that code itself could be secret and there would be no way to learn either what the code is or the results of its computations without observing bits marked as tainted (secret)

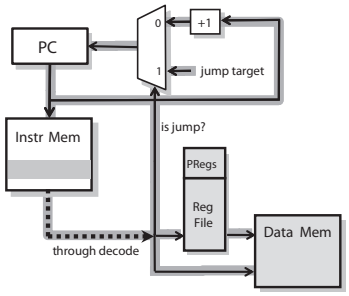

Figure 4.1: Figure shows the basic GLIFT processor running tainted code. The taint (in gray) spreads to the entire system state and makes it practically useless to track information flows.

always keeping the *minimal trustworthy control* necessary to return the machine to a fully untainted state.

4.1.2 Bounding and Cleaning up Tainted State with Execution Leases

To understand how an Execution Lease helps to solve this problem, let us first consider the execution of arbitrary tainted code. This case, where the bits of the actual instruction are tainted, is the most difficult to bound. An untainted function `foo` (e.g. some trusted function), wishes to call a tainted function `bar` (e.g. some arbitrary code). On a traditional machine, this could be implemented with a call and return. The problem is that once the PC jumps to tainted code, everything that code does is tainted, even the eventual return instruction.

Instead we need a way to jump into that tainted code such that a) we can get back to `foo` without learning *anything* about what happened inside `bar`, and b) we need a way to bound *all* the state that can be changed by `bar` so that `foo` can't learn about `bar` by observing things that `bar` did not do (i.e. bound the implicit flows). Because confidentiality and integrity are different forms of the same problem we can phrase the exact same property by just changing the way "taint" is interpreted after the analysis is complete.

The idea behind an Execution Lease is to grant access to a limited amount of state of the machine (including the PC and a portion of the memory) for a fixed and predetermined amount of time in such a way that i) enforcement of the lease can *never* by affected by tainted data, ii) the *critical* tainted state (e.g. the PC) can be scrubbed leaving no residue of tainted data behind, and iii) that it is clear through a *gate-level analysis* of the flow of information that properties (i) and (ii) hold (e.g. it does not depend on some property of the software or some semantics of some state to show that (i) and (ii) hold). We implement these execution leases with special instructions `settimer` and `setbounds` that enforce a bound on the number of instructions that can be executed before control is restored back to the caller and a bound on the accessible memory region respectively. However, to see why and how these semantics keep provably tight control over the flow information we need to explain the implementation of the mechanisms behind these semantics.

4.2 Mechanism

To make sure that the called context (the leasee) does not interfere with the calling context (the leaser), an Execution Lease must enforce a bound on the control flow of the leasee. This ensures that control is returned to the leaser in a manner that is in no way dependent on the leasee. In contrast, during a typical function call, the callee determines when (and if) to return to the caller. Additionally, the leaser must enforce a bound on the address space accessible to the leasee to prevent information from being written explicitly throughout the entirety of the machine. Finally, we need to ensure that both control and address bounding can be performed *without ever making an architectural decision based on the taint values*. This is a subtle but very important point. If we use the taint values in determining whether to "admit" or "deny" a particular action, the fact that the status of that action (admit/deny) is visible to the architecture implies that a dangerous flow of information has taken place. In such a case it may be possible to, for example, try writing to particular addresses to see if those writes are permissible, thus learning something about the values written there. In a very real sense, if the architecture is not separated from the GLIFT logic, the GLIFT logic becomes *part of* the architecture logic, and would then be subject to same potential for covert channels and implicit flows as any other

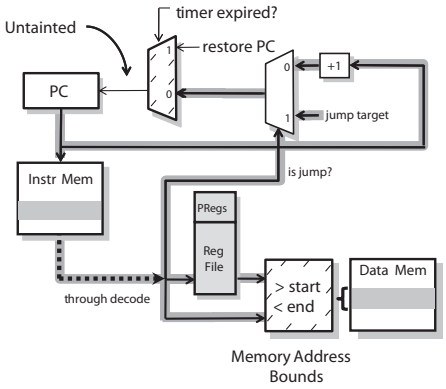

Figure 4.2: Execution Lease Architecture: Lease logic (dashed) bounds tainted programs in both time and space, and prevents the entire system state from becoming tainted. PC is restored to an untainted restorePC value when an untainted timer expires. Lease logic is also used to bound the memory regions the tainted code can access.

architecture logic. Later in this section, we show how this tangling of GLIFT logic and architecture could happen when propagating information flow for an untrusted store instruction. Instead, we need to build an architecture that handles space and time isolation both cleanly (so we can see it to be true at the gate level) and inherently (to avoid the tangling data and taint bits).

4.2.1 Inherent Enforcement of Time-Bounds

Instead of a call-and-return, we can ensure that control will be restored to the leaser context using a timer. In essence, one leases the program counter out to the leasee (which may or may not be tainted and where that taint might either indicate secret or untrusted code) for a fixed amount of time. Once the timer

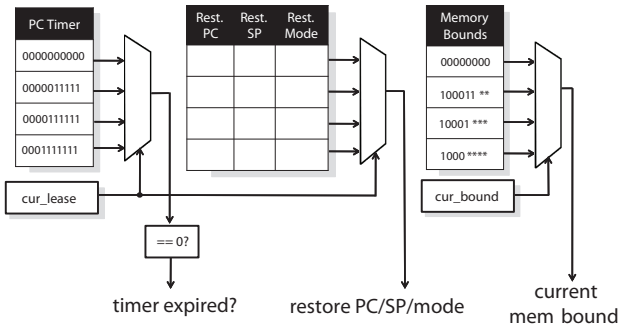

Figure 4.3: Figure shows the information that has to be stored as part of a stack of successively nested execution leases. Note that a restore SP (here, the *cur_lease* and *cur_bound* registers are the stack pointers) is stored with each stack because we do not want to compute the next SP from the current (possibly tainted) SP when a lease expires.

expires, control is automatically restored back to a return PC value that was provided by the leaser when it invoked the lease. Figure 4.2 shows the Lease architecture, and a scenario where untainted code leases the CPU to some tainted code. The timer value itself and the restore PC are untainted, and when the timer expires, a MUX is used to reset the PC to the restore PC. Correspondingly, the GLIFT-logic observes that the MUX output is dependent solely on untainted values (i.e the old tainted PC has no effect), and marks the PC as untainted. Of course nothing is ever this simple, and here the complexity lies in the fact that we need to support *multiple nested leases* to support multi-level procedure calls.

The need for multiple nested leases naturally suggests maintaining a stack of lease records that stores the time the lease is active for and the PC value that the control must return to when the lease expires. We have to implement a stack that

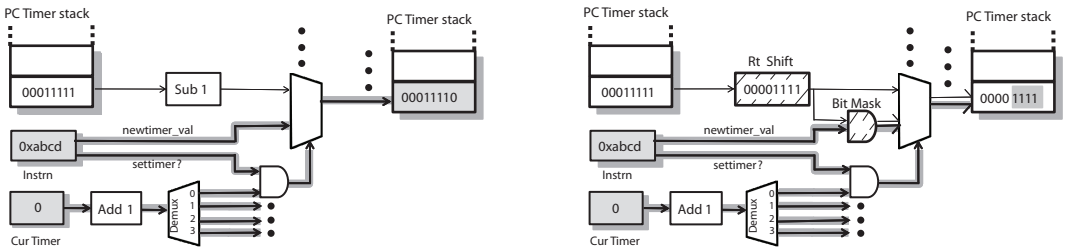

Figure 4.4: Preventing taint explosion through timers: Figure shows two possible timer implementations in a situation where some tainted code is executing under time bounds set by untainted code. On the left, using an intuitive timer implementation, the taint (gray lines) spreads through the logic and marks the previously untainted timer value as tainted (shaded gray). However, the figure on the right shows that by using a bit-mapped timer encoding and by masking off the leading 0-valued bits from the tainted new timer value, we can ensure that the tainted select input to the MUX can only choose between two identical untainted values for the leading 0-valued bits. This makes it explicit at the gate level that the tainted new timer cannot affect the existing lease timer bounds.

stores these attributes, but where the information flow containment is inherent in its gate-level implementation.

With each lease entry in the stack, its leaser's location on the stack is also recorded as part of the lease entry (*restoreSP* in Figure 4.3). The *restoreSP* thus carries the taint of the leaser, and this allows the Lease-CPU to pop leases by setting the *cur_lease* register to its *restoreSP* value without having to compute the next *cur_lease* value from its current value following the usual stack semantics. If we always use the current value of *cur_lease* to compute the next, once the *cur_lease* register is tainted, it will prevent itself from ever being reset to untainted.

To implement successively nested timers, we encode the timers as a bit-vector where each bit represents a minimum time unit. For instance, a timer of value 00...0111 will execute for three time units. Decrementing these timers then requires shifting the register to the right once every time unit with a 0 entered at the MSB. Nesting of successive timers is enforced at the bit level by using the 0s in the right-shifted current timer value as the prefix of the next timer. Figure 4.4 (right side) shows how this mechanism provides gate-level guarantees as opposed to an intuitive scheme that used subtractors to decrement the timers (left side in Figure 4.4). Using the intuitive scheme, an untrusted select input to a MUX decides the next timer value from among the decremented current timer or a new timer value provided by the `settimer` instruction. Even though the decremented timer is trusted, its bit-values could differ from the new, tainted timer value, and because the select itself is tainted, GLIFT logic will mark the MUX output as tainted. As a result, even though we can manually observe that the intuitive implementation is functionally correct, its semantics are not *inherent* in its gate and bit-level implementation. Using our scheme, masking off the trusted, leading bits (0s) ensures that a tainted select chooses between two trusted 0s for the leading bits of the next timer value. As a result, the 0s in a timer value can be shown to increase monotonically until it expires completely. On each cycle,

the processor logic detects if the current timer has reached 0 (expired) and if it has, the processor's PC is assigned the current restore PC from the stack.

The timers are bit-encoded in such a way that a few very small sizes are supported in addition to the largest function being covered. In general, since leases require the timing behavior of functions to be specified statically, there is going to be a correlation between the timer encoding and the execution time overhead as compared to a general-purpose version of the program, and as our application suite grows, developing encodings with a wide range will become more important. Since very small leases are often used for small functions and indexing into arrays, in our prototype, we chose to assign two bits each for time granularities of 4 and 32 instructions, and four bits each for 256, 2K, 4K, and 8K instructions. This simple encoding allows lease durations from 4 to 58440 instructions, and is sufficient to cover our application suite.

Finally, we note that the stack of timers come into play only when at least one lease has been set. The processor begins execution in GLIFT mode in a base context that has no corresponding timer, which allows for trusted programs that execute in never-ending loops. We expect that the processor will begin execution in trusted mode in this base context, and because of this the *first lease entry on the PC stack* is expected to always be trusted. We have already discussed mechanisms used to implicitly reset the taint values for important processor state

like the PC, (and thus the Instruction word) and the *cur_lease* register. Now we discuss our mechanisms to precisely enforce memory bounds for loads and stores, and the reason behind a power-of-2 aligned memory bounds field.

4.2.2 Inherent Enforcement of Memory Accessibility

Consider a scheme for enforcing memory bounds that allows a store to go through to memory only if it is within the specified bounds, and some tainted code executing a store instruction. One intuitive option to build such a memory controller would be to use comparators to check if the store address is within bounds, and forward the store instruction to memory only if it is. Figure 4.5 (left) shows how GLIFT logic will propagate the taint through such a bounds-enforcing logic. Since the address itself as well as the memory's chip-enable is tainted, GLIFT logic for the memory decoder marks *all* the wordlines as tainted.

Instead, in our architecture (right side of Figure 4.5), memory bounds are stored in ternary format where trailing "∗"s represent the desired memory bounds (for e.g. setting bounds register to 10** enforces bounds from 1000 to 1011). A memory controller then composes the address that is actually sent to memory by taking the high bits that are set to either 1 or 0 from the memory bounds register and concatenating the lower bits from the incoming address generated by (potentially untrusted) code. Through such a concatenation, the address sent to

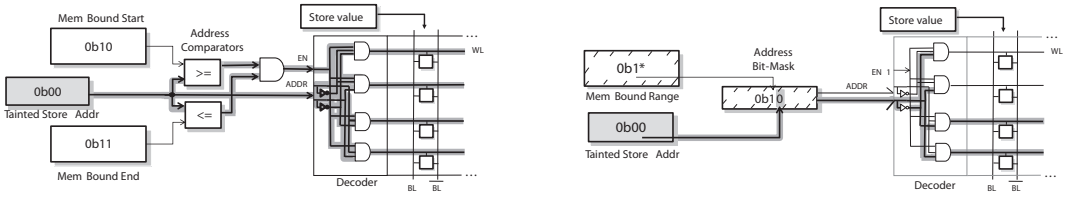

Figure 4.5: Figure shows (on the left) the problem of implementing memory bounds in a naive fashion, where a tainted address causes all the Word Lines and thus the entire memory to be marked as tainted. On the right, with a bit-masked memory address bound, only the currently accessible memory range is marked as tainted by the GLIFT logic.

memory will always be within the bounds, and the isolation is handled cleanly. Further, this concatenation will create a new address that is partially untainted (the bits that came from the untainted bounds) and only partially tainted (the remaining lower bits extracted from the tainted address). By sending this address and its taint simply to the GLIFT-generated shadow memory decoder, the shadow AND-gates inside the decoder will automatically taint only the address range indicated by the tainted "$*$" bits. Thus, information flow containment through leases is made *inherent* at the bit level.

To ensure that a `setbound` instruction creates successively nested memory bounds, the bounds are stored as a combination of address and its mask and are thus restricted to be power-of-2 aligned. Each successive bound is composed using the current bounds' most significant unmasked bits and concatenating the remaining bits from the setbound instruction. In the next section, we show how memory bounds are used in our benchmark programs for function calls as well as

for making protected indirect memory accesses. Since these happen independently, we provide independent stacks for the PC and memory bounds (i.e. memory and PC bounds have independent timers). Further, we realized that allowing for two concurrent memory bounds makes it convenient for programs to share memory though global memory regions while also working in their local frames. As a result, we have *two* stacks that enforce two concurrent memory bounds in addition to the stack of PC timers.

4.2.3 Executing General Purpose Code

We can use the gate-level timing and memory bound guarantees that leases offer to execute general purpose code, specifically conditional jumps and indirect memory accesses, within a lease. For the leasee to be able to use general-purpose instructions, the leaser must set the *Mode* bit for the lease, indicating a general-purpose lease. The mode in which the current lease is executing (GLIFT or General Purpose) forms part of the current context (as shown in Figure 4.2). The leaser's mode is also stored along with each entry in the PC stack so that the *current_mode* register can be restored to a trusted leaser's mode when a lease expires. An interesting feature of our architecture is that once a lease is set in general purpose mode, no further leases can be set until the lease expires. Leases can only be set in GLIFT mode to ensure that the fixed size lease stack

cannot be used to leak information covertly, while within a general purpose lease, conventional call-and-return semantics can be used to implement functions.

Conditional jumps help performance because they can allow the lease time to be limited to the maximum of the two conditional paths at an $if - else$ branch instead of executing both sides of the branch. Similarly, indirect memory accesses allow programs to index into arrays arbitrarily without having to iterate over the entire array and predicating out the desired index. In the next section, we present the Lease ISA and how it can be used to implement a high level language.

4.3 Evaluation

Now that we have described the basic microarchitectural structures in our Execution Lease prototype, we provide more details of how these structures are exposed to the programmer and compiler. To demonstrate that such an approach can lead to a correct and relatively easy to program secure microcontroller, we have built a fully synthesizable prototype instantiated on an FPGA and a compiler that translates high-level constructs like functions, loops, and array accesses into machine code with lease instructions, jumps and indirect memory accesses. In this section we present performance and area results for this prototype and describe

several of the more important features of the design through code examples and comparisons to the original GLIFT work.

4.3.1 A New ISA for Execution Leases

The original GLIFT architecture uses predication to prevent tainted data from ever affecting the PC, while a special `countjump` instruction allowed a fixed number of unconditional jumps to support fixed-length loops, and special `load-looprel` and `store-looprel` instructions allowed programs to access a fixed range of memory addresses in the loop (by using an immediate value for the base address).

In contrast, our Execution Lease ISA allows the caller of the lease to set explicit bounds on the range of memory addresses that the callee is allowed to access (using `setmembound-hi` or `setmembound-lo`), and the time the callee will execute for (using `settimer`). The `setbounds` instructions set the address bounds for a given time duration and along a given power-of-2 aligned boundary, while the `settimer` instruction sets a timer, the general-purpose mode, and automatically stores a PC to be restored when the lease expires.

Function calls in the new assembly: We have chosen to implicitly record the third instruction after `settimer` (i.e. $PC + 3$) as the restore-PC, allowing for one instruction in the middle for an unconditional `jump` to the callee and another

instruction that spins in an infinite loop to wait out any remaining time on the lease if the callee finishes early. The address of this infinite-loop instruction is suggested as a return address to the callee as part of the calling convention. Note that even if the callee disregards the return PC and is still executing some code when the lease timer expires, the PC will be yanked back to the restore PC that was recorded when the lease timer was set. The return PC is suggested so that if a lease duration is longer than required, the callee need not be concerned with waiting out their leases or completing the lease precisely without spilling over into others' code. If the time is insufficient, leases will still ensure that the effects of an unusual code-path do not propagate outside the current time and space bounds.

A function call looks like the snippet of assembly code in Figure 4.6 (that calls an I^2C bus initialization function).

Accelerated array accesses: The `setbounds` and `settimer` instructions are also used to enable indirect memory accesses (used for accessing array elements). Since the GLIFT ISA allowed only direct memory addressing, indexing into an array required using a loop that iterated over the entire array. The chosen address was accessed by using a predicated load (or store) that is set to True at the desired index.

For example, consider a code snippet of the `SubBytes` function for an implementation of the AES [23] encryption algorithm in the original ISA (Figure 4.7).

```
#[Pred] Instruction Operands
[1] setboundg    1000000**, 72  #bounds for global memory
[1] ...                         #copy args, ret-addr to callee function
[1] setboundl    1010100***, 72 #bounds for local stack frame
[1] settimer     1, 72          #timer := 72. mode := 1 (general purpose)
                                #restorePC := PC + 3
[1] jmp          i2c_init       #jump to callee function
label13:                        #function returns here: an infinite loop
[1] jmp          label13        #to wait out remaining lease time
[1] load-direct  r0, 0x29f      #arrive here when lease expires
```

Figure 4.6: Figure shows the assembly instructions generated to implement a lease called by a programmer in the high-level language. The `setboundg` and `setboundl` instructions set the address bounds for all subsequent memory accesses for the next 72 instructions. The `settimer` instruction then initializes the next lease with a mode (general-purpose or GLIFT), a timer and a restore PC, and the following unconditional jump sets the PC to that of the callee function. The PC is expected to return to `jmp label3`, where it will spin until it is restored to the `load-direct` instruction once the PC timer has counted 72 instructions. Note that the actual function required 42 instructions, and the corresponding local and global timers would be 43 and 50 respectively. Using our bit-encoding of the timers, the timers are set to 72 instructions.

The function substitutes the value in the `state` matrix with values in the `SBox`. The code in Figure 4.7 loads the value in the `state` matrix (which in this example is stored at address starting at 0x100) and every element serves as an index to the `SBox` and is substituted by the value in the `SBox` (which is stored at address 0x300). The `SBox` is a 256 entry table, correspondingly the `countjump` instruction 0x05 loops back 255 times just to read a single value from the SBox table [70].

In Lease ISA, the `setbound` instructions set a memory access bound for a limited amount of time. In Figure 4.7, the lease lasts 1 instruction which allows the

C code	GLIFT - Base Asm	Lease Asm
state[i] = sbox [state[i]]; /*sbox: int [256] */	0x00 [1] load - looprel R0 := [0x100 + C0] # R0 = state[i] 0x01 [1] init - counter C1 := 0 # start the loop. j = 0 0x02 [1] cmpeq P1 := C1, R0 # if (j == R0) 0x03 [P1] load - looprel R1 := [0x300 + C1] # R1 = SBox [j] 0x04 [1] increment - counter C1 := 1 # j++ 0x05 [1] countjump 0x02, 255 # loop back 255 times	[1] load -indirect R0 := [0x100 + R2] [1] setmembndlo 00000011***, 1 [1] load -indirect R1 := [0x300 + R0]

Figure 4.7: Execution Leases allow indirect memory accesses within bounded memory regions. In comparison, the base GLIFT ISA performs table lookups by iterating over the entire array and predicating out the desired index. For many cryptography algorithms, table lookups are numerous and each base GLIFT lookup adds performance overhead in proportion to the table sizes.

program to perform a bounded indirect memory access (as shown in an unoptimized snippet from AES in Figure 4.7). In fact our compiler conservatively inserts bounds before every memory access, but merges adjacent leases of `setbounds` that have the same address range and creates a longer lasting lease.

Executing General Purpose Code Safely:Leases allow us to execute *conditional jump* instructions safely. By enforcing a tight time and memory bound, a lease ensures that there is no untracked side-effect of the general purpose code. As mentioned in Section 4.1, general purpose code cannot set any further leases. Lease instructions (that set timers and bounds), if executed in general purpose mode, will not commit and will be equivalent to a no-op. In this mode, functions can use the conventional memory-based stack to implement function calls, and the ISA includes an instruction to load the current PC into a register that is used to compute return addresses for callee functions. Finally, the instruction set includes the usual 2-operand single-destination ALU-ops to execute arithmetic, shift, and

Mode	Instruction	Information Flow Properties
GLIFT only	setTimer	The lease caller's taint value is assigned to the callee's lease state (like timers, restore PC, restore SP etc). Since the cpu starts off with trusted code, the first timer on the stack is always trusted (i.e untainted).
	setBound g/l	
General Purpose only	jump If RegZero	If jump target or condition Reg is tainted, the PC is marked tainted (following which, all registers and the entire leased memory becomes tainted too).
	jump To RegValue	
Both Modes	load Immediate	The destination register gets the taint value of the PC
	load/store Direct	The destination gets the taint value of the PC or - d with that of the source memory address.
	load PC To Reg	Taint value of PC is assigned to destination Reg.
	load/store Indirect -global	A tainted memory address will mark the entire leased memory as tainted.
	load/store Indirect -local	
	countJump	The PC gets the taint value of the instruction word.
	directJump	
	and, or, not, xor, shl, shr, add, sub, cmpeq, cmplt	Destination reg untrusted if either operand is untrusted. For AND and OR, the trust value is tracked precisely

Figure 4.8: An overview of the ISA of our prototype architecture, and the information flow tracking policies that are extracted from the actual logic level implementation.

compare instructions. Table 4.8 shows the complete list of instructions supported by our Execution Lease ISA.

4.3.2 A Prototype Processor that implements Execution Leases

We have built a prototype microcontroller that implements all the mechanisms described in Section 4.1 to support execution leases. The Lease CPU is a 5-cycle, in-order, unpipelined processor with 64Kb of Instruction and Data Memory, 8 general purpose registers and 8 registers to store the loop counters (that count down the number of iterations for `countjmp` instructions).

Most importantly, the CPU maintains three independent lease stacks (one each for the PC, global and local memory bounds) with each stack allowing upto

8 nested leases. Each lease stack includes registers to store the timers, the return stack pointers for the entry, the current stack pointer, and the restore PC and restore mode for the PC stack, and memory bounds in ternary format for the global and local memory stacks.

The Lease CPU is implemented in Verilog as a structural composition of gates and instantiations along with registers for processor state and block RAMs for Instruction and Data Memory. This structural composition of logic allows us to automatically extract the gate-level taint tracking logic for the entire processor by shadowing all registers and wires, and connecting together the shadow gates and modules in the same manner as their original processor counterparts, resulting in a full shadow machine that operates on taint bits instead of data.

We have used Altera's QuartusII v8.0 to synthesize the Glift-Lease CPU onto a Stratix II FPGA with synthesis settings that balance both area and timing. These area and timing numbers are then compared against the basic Glift CPU presented in [70], and with Altera's Nios RISC processor. We chose Nios-standard core as a point of comparison as it has a simple ISA, is reasonably well-optimized, and is targeted to the same family of FPGAs.

Figure 4.9 shows the area and timing results from synthesizing all the processors under test. This includes the Nios processor as a general purpose baseline, the basic Glift processor and its version augmented with shadow logic as the Glift

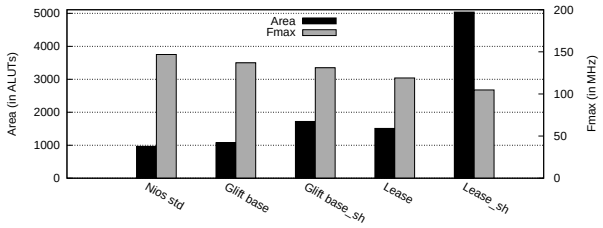

Figure 4.9: Quantifying the area and timing overhead of Execution Leases in a Glift CPU. The left Y-axis compares the number of FPGA lookup tables(ALUTs) required to implement a Glift Lease processor with that for the Glift base processor and a general purpose RISC micro-processor by Altera. The graph also shows the number of lookup tables to implement versions of both Glift processors augmented with shadow logic for information flow tracking. The right Y-axis compares the Fmax for same set of processors.

baseline, and the Lease CPU (with and without shadow logic) as the processors under evaluation. The left Y-axis shows the area in number of ALUTs (black bar) while the maximum operating frequency values are presented on the right Y axis (gray bar). One important result is that in absolute terms, all these processors are very small in size. Even on the outdated Stratix II EP2S15F67263 FPGA that was used for evaluation, the smallest Nios processor required only 5% of the combinational ALUT resources, while the largest Lease CPU required 35% of the ALUTs. The largest FPGAs are over 10X larger than EP2S15F67263, and so the area of these CPUs is probably of little concern.

The second result is that in relative terms, the Lease CPU requires 50% more ALUTs than the Glift baseline CPU. Augmenting this Lease CPU with the information flow tracking logic adds an additional 2X ALUTs. This jump from the Lease processor to "Lease_sh", its shadowed version, is much larger than the 70%

increase from Glift base to Glift base_sh. Not surprisingly, the main overhead of implementing leases arises from the MUXes and deMUXes required to read and write back to the set of 8 timers, restore PC, restore SP and other lease state. The auto-generated shadow MUXes then account for the overhead of the Lease_sh CPU. Since the lease behavior of a program is known statically, compiler optimizations could potentially bring down the number of hardware stack entries required.

The Fmax of the synthesized CPUs, on the other hand, show that the Glift and Lease CPUs at 120-130 MHz are only slightly slower than the Nios processor at 160 MHz. This difference in frequency is mainly because Glift and Lease processors support barrel shifts while Nios supports only 1-bit shifts. With 1-bit shifts, the Lease CPU can also operate at 160 MHz. In relative terms, the synthesized Lease_sh CPU at 108 MHz also runs slightly slower than the Lease CPU (120 MHz).

4.3.3 Programming with Execution Leases

To code up our benchmarks, we have designed a simple high level language with constructs that capture the functionality of the Lease CPU. The lexical and grammar definition of Lease compiler is generated using Antlr, and all other parts of the compiler are written in Java. We demonstrate various aspects of our language

through an encryption library modeled after the privilege separated OpenSSH. Our library uses an I^2C bus interface for I/O and implements a public-key RSA encryption function to exchange symmetric keys and followed by AES for all subsequent communication. This library models a scenario where the I^2C bus drivers are untrusted (e.g. imported as a commercial binary) and also implemented as a general purpose program with a potentially unbounded loop. The aim is to completely isolate the untrusted drivers and prevent them from ever *affecting* the encryption functions or accessing any information illegally (for e.g., the symmetric key for a transaction).

The `lease` statement is the most important new idea of this language. Its syntax is: **lease**(*timer*, *memorysize*, $Function(arg0, ...)$, *returnvalue*); and it allows a programmer to lease a bounded area of memory, jump to a location, and execute a bounded number of instructions. Each function also has an *execution mode*, GLIFT or general purpose, which has to be specified statically.

Estimating Execution Time: The *timer* argument to a lease statement conveys the number of instructions that the $jumpToFunction$ is allowed to execute. If left at 0 for a GLIFT mode function, the compiler can automatically fill the timer value by statically analyzing the $jumpToFunction$ and computing the total time required to execute both sides of conditional branches, fixed size loops, and some extra instructions to pass in arguments, set appropriate memory

bounds, and retrieve the return value. Further research is required to make the compiler capable of estimating execution times for general purpose functions and provide useful suggestions to the programmer of the GLIFT mode caller function. On one hand, many security functions that operate on streaming data are easy to estimate, as they already take a fixed amount of time (e.g. AES, RSA, md5 etc). In cases where the function is accessing some peripheral device (or in general waiting for some asynchronous communication) the lease bounds will be governed by system-level timing constraints. For instance, aircraft require some critical computation every N ms, and the non-critical functions will work around this constraint.

Setting memory bounds: The *memorysize* is used to bound the local memory accesses $jumpToFunction$ will make, and can either be determined statically for GLIFT mode programs or (as is the case for peripheral interface drivers) be fixed to an arbitrary size based on the system designer's discretion. In addition to local memory bounds, global memory bounds have to be set in case the caller wants to allow the callee restricted access to some additional region of memory. For instance, the I^2C transmit function can be allowed access to the encrypted message in addition to its local frame. The callee can use the `load/store-global` instructions to access these out-of-frame addresses. One concern might be that putting arrays into power-of-2 aligned memory regions might prove prohibitively

hard for the programmer. While more research is definitely required to place arrays in the most compact manner possible and to minimize data movement, our compiler has a simple algorithm whereby it determines all the constraint sets for arrays that need to be adjacent, places them along aligned boundaries as far as possible, and moves arrays around for when all constraints cannot be met. The caller has to specify using the `@` symbol (as in `int [2] @arr`) if a function call argument requires to be accessed using the global load/store instructions.

Another very useful optimization the compiler performs relates to the problem of setting the memory bounds before every load or store, especially in loops. For such cases, the compiler begins by inserting a `setbounds` instruction before every memory access, but in a later pass discovers all adjacent setbound instructions that use the same bounds and coalesces them into a single `setbound` instruction with the common bound.

Execution Modes: Every function declaration requires an $@GLIFT$ or $@General$ prefix that states the mode the function will execute in. In Glift mode, the Lease ISA only allows a *countjump* instruction to jump to a PC a fixed number of times. The language thus supports fixed size loops of the form `for i in range(start,end,step) { block }`. The usual $if - else$ statements are compiled down to predicated blocks of code, and function definitions and statements resemble similar constructs in other imperative languages. In General-purpose

mode, conditional jumps are allowed, and if-else statements need not both execute, but the lease instructions (`settimer` and `setbounds`) are no longer available. General purpose functions can use the conventional in-memory stack to execute function calls, and use general-purpose registers as stack and frame pointers.

In Figure 4.10, we show a snippet from our encryption library. Execution begins in the main() function in GLIFT mode. This sets a lease for initializing the I^2C bus driver, transmitting the start signal and device address, and reads in input from the serial bus. In the example, we show the serial clock (SCL) and data (SDA) bus lines using the memory mapped addresses (`scl_da[]` and `scl_da_in[]`). Once the input has been read in, either RSA or AES is invoked conditionally, but since the function is in general-purpose mode, it only needs to set a lease of $max(aes, rsa)$ (unlike the Glift-base assembly that would require both AES and RSA to execute on every iteration). It is interesting that even though the I^2C receive function has a loop that queries a device for an ACK and can do so indefinitely, once the lease expires, control will be restored to the main function.

End-to-end property: This example shows how leases can be used by programmers to explicitly manage the flow of all tainted information through the CPU and memory, and thus ensure the *integrity* of some critical function (the encryption library) in the presence of untrusted functions (the bus drivers). Considered

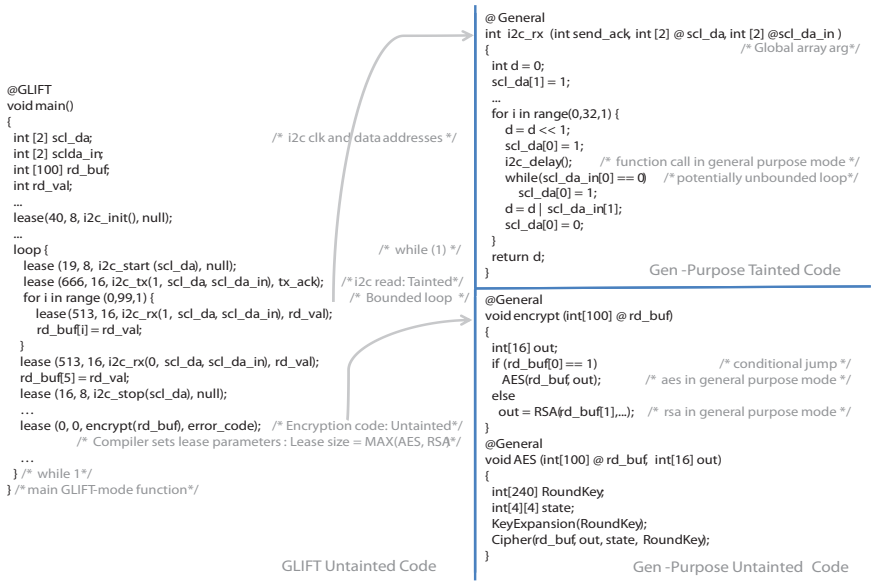

Figure 4.10: Programming using Execution Leases: Execution begins in trusted (untainted) GLIFT mode. Function calls can be made to general-purpose functions that operate within a fixed time and memory lease. Within a lease, general purpose code can execute conditionals (without predication), potentially unbounded loops, and communicate through protected memory regions, while the CPU implementation guarantees that no tainted code can ever interfere with out of bounds memory or execution time.

in a *secrecy* context, leases could be used to show that the only transfer of information between a secret library and the unclassified bus drivers is the encrypted buffer.

4.3.4 Quantitative Differences in the Resulting Code

While Execution Leases provide the program with a completely new ability, the option to provably contain the flow of information even when tainted *code* is executing (where that code is tainted either because it is secret or it is untrusted), it

also provides quantifiable differences in the performance of applications. We compare the execution time of several different kernels running on the NIOS processor, on the original GLIFT microprocessor, and on the Execution Lease machine. The NIOS code was generated from gcc with level 2 optimizations enabled. The code targeting the original GLIFT machine is hand written assembly. The code targeting the the Execution Lease machine is generated by our custom compiler which performs no optimization other than the lease bound merging discussed above.

As can be seen in Figure 4.11, the static code size between the different machines are all relatively close (for example MM is 83 instructions instead of the 73 from in the original GLIFT machine) with only a few glaring exceptions (AES is 2X bigger and BSort is 5X bigger). However, even though AES is 2X bigger, it is approximately 68X faster using Leases instead of the original GLIFT ISA. BSort is also interesting because, on the original GLIFT machine, this was really the *only* practical way to do sort – because random indexed lookups took O(n) time instead of O(1). On that machine, bubble sort takes O(n^2) time while merge sort takes on the order of O(n^3) time. On our new architecture, merge sort would take O($n$log(n)) time which means that even though bubble sort is larger and takes longer, in fact "fastest worst-case sort" would be a more appropriate benchmark. Measured against the general purpose NIOS CPU, the dynamic instruction counts

in a Lease-CPU are comparable, mainly because most of the security kernels are very regular and easy to estimate tightly.

In terms of performance it is worth noting that excluding the two *best* performing applications, the average speedup is still 32%, while the two best performing applications (FSM and AES) each are running 8.1X and 68X faster respectively. FSM and AES are each so much faster because they are dominated by table lookups, FSM to find the next state, and AES to perform the sbox operation. While this shows the potential of Execution Leases to drastically reduce the time to perform some of the most fundamental computations, Leases allow us to compose together larger programs such as the encryption library that would be extremely hard without function calls and extremely slow without table lookups.

4.4 Conclusions

Architectural support for *Execution Leases* has the potential to be a powerful new tool for the designers of high assurance systems. The idea that execution resources are leased out to regions of code with fixed bounds on the time and memory addresses is both a model of execution that a programmers can understand, yet can be implemented in such as way that safety is verifiable all the way down to the gate level. In implementing a full prototype of an Execution Lease

Kernel	Description	Static Instruction Count (NIOS)	Static Instructions (original GLIFT)	Static Instructions (Execution Lease)	Dynamic Instructions (NIOS)	Dynamic Instructions (original GLIFT)	Dynamic Instructions (Execution Lease)
FSM	CSMA-CD state machine with with 6 states and 4 inputs. Many table lookups	123	190	130	441	3322	410
BSort	Perform bubble sort on a fixed size list of integers	26	21	126	20621	30358	43518
RSA	Montgomery multiplication and exponentiation from RSA public key cryptography	256	143	95	44880	39297	26329
AES	Block Cipher, involves extensive table lookups and complex control structures	781	1100	2113	12807	1082207	15785
Md5	Core of the cryptographic hash function, involves mostly ALU and logical operations	769	1386	951	1226	1431	1012
MM	Matrix Multiplication	108	73	83	9043*	17035	10094

Figure 4.11: A comparison between the static instruction counts and dynamic instruction counts between a traditional processor (the Altera NIOS), the original GLIFT based microcontroller, and our Execution Lease based processor. As we have not implemented a multiply instruction yet in our prototype, the Matrix Multiply result for the NIOS does not make use of native multiply instructions either. Excluding the two *best* performing applications, the average speedup is 32%, with the two best performing applications (FSM and AES) each running 8.1X and 68X faster respectively.

machine, we came upon several challenges. For example, execution leases can themselves invoke further leases and the processor has to enforce that successive leases are nested in both time and address space. Bounding all memory accesses within the latest bounds requires constructing the memory address in a unique manner from the bounds and the incoming address, while also requiring a special tag memory that can mark a range of addresses as tainted in parallel. However, we describe a series of techniques for overcoming these challenges, and we show that the resulting machine is both far easier to program and capable of executing far more powerful code than prior GLIFT based processors (it is not limited to

GLIFT-ISA from [70]). While there is still more work to do in further refining the ISA, optimizing the hardware implementation, fleshing out the language features, and improving the code generated by the compiler, even now the Execution Lease machine is the only design we are aware of that is capable of demonstrating the non-interference of two or more general purpose software components all the way down to the level of gates.

Chapter 5

Making Gate-Level Verified Systems Practical

Systems requiring the highest levels of trust are often designed and built using waterfall principles, modeled at a high-level in a theorem proving system, coaxed through said theorem proving system by hand, documented to an tremendous degree throughout the entire process, and then evaluated by a trusted third party or evaluation board [3]. It is estimated that the entire process costs over $10,000 per line of code [6] and takes over 10 years to complete [4]. In the end, there is more evaluation of the *development process* than the final *end artifact*, and formal properties are shown to hold only for hand-written high-level models of the system rather than for the actual implementation [33]. The ultimate goal of our work is to create full system implementations (including both hardware and software) with security properties that can be directly measured and verified.

The restrictive systems presented in previous chapters leave numerous challenges with regards to the creation of real systems. The hard nature of an Execution Lease programming model does not naturally support performance enhancing micro-architectural features such as caches, pipelining, branch predictors, or TLBs because of the timing variabilities they introduce; it lacks sufficient support for software behaviors that cannot be bounded and divided into fixed regular sized chunks of work, it does not provide any easily verifiable mechanisms by which different trust domains can communicate safely; and it makes handling the inherently dynamic nature of I/O very difficult. Furthermore, information flow security is provided using additional *analysis logic* that adds substantial area-delay overheads to the deployed system. In this chapter, we present our experiences building a full system that removes all of the above restrictions yet is still verifiably information flow secure, i.e. conforms to a specified information flow policy.

Our method for overcoming this challenge is two-fold: first, we create a thin skeleton of hardware, that when configured and operated by a small piece of software, describes a *minimal* functionality with which the information flow of the rest of the machine can be governed. **In essence, this minimal slice of the hardware is to the entire processor core, as a microkernel is to a full operating system**. This strict structure is then enhanced by a runtime system that delivers more dynamic or non-information-flow-critical services (e.g.

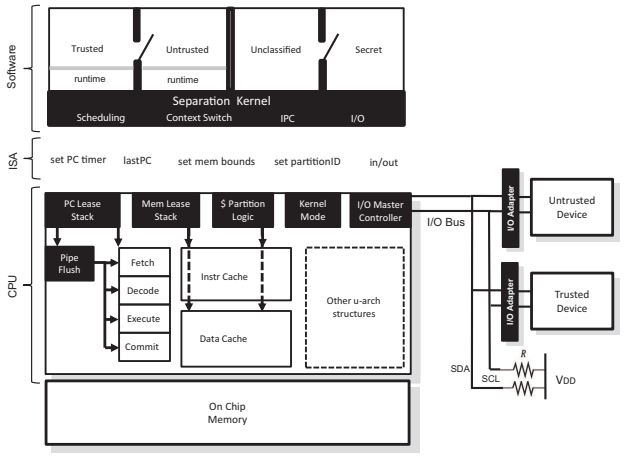

Figure 5.1: The proposed architectural skeleton (shaded black in the CPU) that allows explicit software control over the entire processor state. The processor includes dynamic micro-architectural features such as caches and pipelining. This hardware skeleton is used by a separation kernel to manage execution time, memory, and I/O devices among multiple security partitions. We also introduce trusted adapters for secure I/O. Here, an I^2C master controller on the CPU manages a shared bus among off-the-shelf I^2C devices with different trust levels. In the end, we verify that the hardware and kernel together enforce a desired information flow policy such as non-interference.

communication interfaces and context swapping). Because these operations are decoupled from the information flow properties of the skeleton, they do not add complexity to the verification of the system as a whole.

To make this idea more concrete, consider the process by which a context is saved and restored. To be a correct context switch, the registers and PC (along with other things) need to be saved off to a region of memory, and then restored when the process is scheduled again. However, to verify the *information flow*

properties of such a system, we need only to verify that the context is saved and restored in a way that does not leak data and violate policy.

To demonstrate these principles we have created a synthesizable full-system prototype, complete with a pipelined CPU, a micro-kernel that enables isolation and communication by explicitly controlling all micro-architectural state, and an I/O subsystem that allows off-the-shelf I^2C devices to be connected to a single shared bus. Our system can provide caches, pipelines, and support for the micro-kernel in only 1/4th the area and with double the clock frequency as more restrictive prior work. Finally, for a system of size 50K logic gates (approximately) and with only 3264b out of 133kB state specified concretely, we can statically verify that the entire hardware-software stack conforms to a specified information flow policy all the way down to its gate level implementation.

5.1 A Secure Architectural Skeleton

Our architectural skeleton, working in conjunction with the micro-kernel, must deliver each of the following capabilities in a way that can be verified to be side-channel free.

1. **Verifiable Common Case Optimizations in the CPU:** Techniques such as caches and pipelining are taken for granted in the non high-assurance

systems but pipeline stalls and cache evictions can easily introduce side-channels. While countermeasures for these side-channels exist, we must be able to formally prove their absence.

2. **Verifiable Context Switches, Scheduling, and Communication in the kernel:** Because timing channels are part of our threat model, the micro-kernel, working in conjunction with the skeleton, must have a way to bound the behavior of a software partition. Furthermore, it must have a way to save and restore process contexts without leaking information about those contexts, to schedule these processes or partitions at arbitrarily fine granularities, and to allow inter-partition communication in a tightly-controlled manner.

3. **Verifiable I/O:** We must be able to construct a system that is able to communicate with the outside world. In particular we must allow software partitions measured access to I/O, and this access must be able to exploit simple off-the-shelf I/O protocols. While both authentication and physical attacks are beyond the scope of this paper, we must ensure that, if necessary, information flow can be limited to a subset of the parties connected on the I/O network.

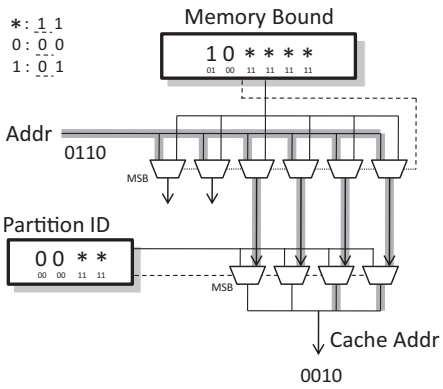

Figure 5.2: Implementing caches: The processor-generated memory address is first masked off used the memory bounds register. This creates a physical address that is within the currently active execution lease bounds. Most significant bits of this address are then further masked off using the trusted partition ID register to generate the address for the cache. As a result, information flow from a potentially untrusted memory access is limited to the currently enabled portion of the cache. Since trusted bits from the partition ID register control the MUXes, information flow control can be verified precisely.

Understanding our approach to the problem can be difficult at first because our design method and verification method are intimately linked. **The verification method works because our designs exhibit incredibly tight control over the flow of information, and our design method is useful because designs can be verified easily.** Later, in Section 5.2, we will describe how we verified a specific incarnation of our system, but for now we can think at a high level about the two primary methods of managing the information flow in our skeleton.

The first method for information flow control is to ensure that certain critical portions of the machine and kernel are always kept with high integrity, i.e. trusted.

If we can verify that the system will never breach this invariant, then these critical bits can be used as the root of trust for the rest of system.

The second method is carefully time-multiplexing the rest of the state between multiple security levels. There are two parts to this second method. Because these time multiplexed bits, for example the hardware's program counter or the kernel's current process ID, will change labels over time, we must bound the effect that these bits can have on the system. Then, after we have finished a unit of time, we must always be certain to "clean-up" any of these bits remaining in multiplexed parts of the system (as controlled by trusted bits).

In short, our minimal skeleton working with the separation kernel ensures that "trusted bits stay trusted" and that "untrusted bits always get cleaned up". Both these methods can then be verified to be implemented correctly through a gate-level analysis. Cleaning up untrusted bits in a verifiable manner is best understood using a multiplexer (MUX): if the select input to a MUX is trusted, and it selects a trusted value, the result can be trusted no matter what that actual value is, even when the other input is untrusted. A MUX can thus be thought of as a gatekeeper for trust, and is used to implement logic that resets critical system state to a trusted value or masks out untrusted signals. Together, both these methods allow, for example, the kernel bits that store the partition schedule

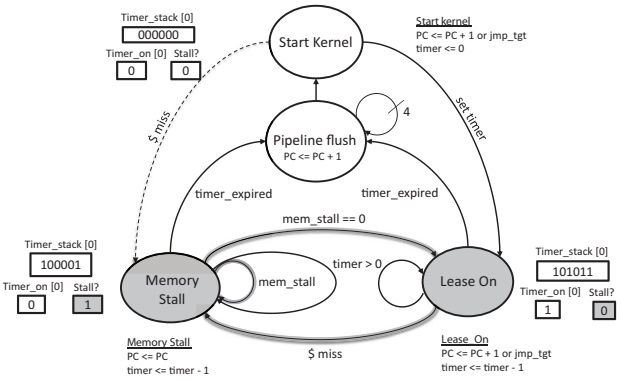

Figure 5.3: Secure Pipelines: The state machine shows the Program Counter update logic for a pipelined CPU. From all untrusted states (in gray), there is a transition to a trusted state that is triggered by a trusted timer, and hence the PC is always reset verifiably. The pipeline flush requires 4 cycles since our prototype CPU has a 4-stage pipeline. The dashed line from start to memory stall is to indicate that while a memory stall is possible, kernel code ensures that no memory access misses in the cache.

to always stay trusted and control the MUXes to reset the program counter from an untrusted state.

In the rest of this section we will show how the combination of bit-level isolation and trusted time-multiplexing allows us to first implement a skeleton that addresses the four requirements listed above and then formally verify that the entire system conforms to a desired information-flow policy.

5.1.1 CPU: Using Caches, Pipelines, and Other Micro-architectural Structures

Caches. Side-channels through caches have been shown to leak information about private keys [9, 76]. These attacks exploit the ability of an attacker to learn the memory accesses of a secret process by first filling the cache, yielding to the secret process, and then inspecting which of its own memory accesses miss in the cache. The fundamental problem is that the cache controller uses both secret and unclassified information to decide which cache lines to evict. Counter-measures to this attack include pinning secret lines in the cache [76] (which was first shown to be vulnerable [40] and then fixed [41]), and proposed secure caches may still be vulnerable to collision-based timing-driven attacks [40].

To implement an information flow secure cache, we design the cache controller to only use trusted values to control the cache contents: i.e. by *partitioning* the cache or by *clearing* the cache based on trusted parameters after untrusted or secret code has finished executing. This ensures that untrusted information is confined to its partition, while trusted information can flow to untrusted partitions (or unclassified to secret).

To implement partitioned caches that are verifiably isolated at the bit-level, we allow only power-of-2 aligned cache partitions that are configured by the kernel

through a `partition ID` register (as shown in Figure 5.2). The partition ID register represents currently enabled cache partition(s) and uses two bits for each bit of the cache address. Of these two bits, if the MSB is 1 the cache address uses the processor-generated memory address else the LSB of the partition ID register is used. For example, for a 4b cache address, a partition ID of 00_00_11_11 implies that the two most significant bits of the cache address will be 00, and the two lower bits will be used from the address generated by the processor. The partition ID register set to all 1s will indicate that the entire cache is available for use. The kernel sets the partition ID register before jumping to untrusted code using the instruction `set_partitionID immediate`. The cache controller also communicates with a memory controller in case of a read-miss or write-evict, and squashes an outstanding memory request when the execution time slot for some untrusted code ends.

This cache controller logic can be verified to be secure at the bit level because the MUXes that select the final cache address are controlled by the trusted partition ID register (Figure 5.2). While prior work has proposed that unclassified code pre-load AES tables into locked cache lines or clear the entire cache after secret execution [41], we are able to implement the mechanism and verify it at the gate-level.

Pipelining. Pipelines are challenging to implement in an information flow secure manner since they introduce unpredictable dynamic behavior through memory stalls, branch prediction, register forwarding etc. Our key insight is that such dynamic behavior can be allowed as long as untrusted programs' effects do not spill over into trusted space and time slots.

Figure 5.3 shows the state machine that controls the program counter (PC) in our CPU. The state machine begins in a `start` state where no PC lease is currently on and trusted kernel code is expected to execute. It can set a timer and transition to the `lease_on` state. This state is typically used to run untrusted or secret programs, but the transition to this state is based on a trusted jump instruction. On a cache miss, the state machine can transition to `memory_stall` state. From an untrusted instruction, this transition will be untrusted and the `memory stall` state will be untrusted too. When the PC timer expires, however, both `lease_on` and `memory_stall` states transition to a sequence of 4 states, one for each stage of the pipeline, where the PC is first restored to a trusted value (and then incremented each cycle). The logic to implement this sequence is hardwired instead of using the jump instruction because general purpose registers may themselves be untrusted at the end of a lease to untrusted code. Since the restore PC is stored in the trusted lease unit, computing the next PC from

this maintains the PC as trusted. Further, since the lease timer has expired, no instructions are committed during this sequence of states.

Other micro-architectural features can also be employed safely through a combination of trusted partitioning and time-multiplexing. While we do not implement a branch predictor or prefetcher, implementing these would require their state to be either partitioned using trusted parameters or flushed at the end of every security context by the kernel.

5.1.2 Micro-Kernel: Context Switches, Scheduling and Communication

A kernel *partition* encapsulates all computational resources required by a security level, comprising of time, memory, and optionally I/O interfaces. A portion of instruction and data memory are reserved for each security level, and when a partition is actually scheduled to run, it gains control over part of the machine such as execution units and register files for a trusted amount of time. To prevent information leaks to untrusted programs, kernel parameters such as time and memory slots allocated to each partition and the overall number of partitions depend only on trusted constants assigned at boot time. As a result, the kernel scheduler implements a statically determined schedule which can act as a coarse-grain first level scheduler, while each partition implements a second-level

scheduler to optimize performance within their own time bounds (as proposed before for real-time [60] and highly secure [37] systems).

Precise Context Switches: The kernel ensures continuous unbounded operation by saving and restoring user programs' state on every context-switch. To support precise control over timing in the presence of caches and pipelines, we introduce two unique features in our micro-kernel. The kernel explicitly manages all micro-architectural state in the processor, e.g. through the partition ID register to enforce cache partitions, and has perfectly imperturbable execution time for each kernel function, e.g. by never having a memory access miss in the cache.

To demonstrate these ideas, we step through the context switch routine that is triggered each time a set timer expires. After a pipeline-length delay to flush the entire pipeline state, the first kernel instruction to commit is a `set_partitionID immediate`. This activates the kernel partition which stores complete context information for all partitions. The kernel then stores general purpose registers in trusted addresses specifically earmarked for the partition's context. This is possible since the number of partitions is a trusted kernel parameter. The kernel also stores the last PC that entered the commit stage when the timer expired, accessing it through the (`last_PC Mem[reg]`) instruction. Once the current context is saved, the kernel loads the general purpose registers for the new partition. It then sets the cache partition available for the next partition (using `set_partition`),

sets memory bounds (`set_membounds global/local` and finally sets a time limit for the new partition to execute (`set_timer`). The `mode` argument to `set_timer` indicates whether the new context will execute in kernel or user mode (since the partitionID register can only be set in kernel mode). Finally, once the new partition's context is loaded and the bounds are set, the kernel loads the user-space PC and jumps to it.

Since the kernel has its own reserved partition in the cache, each of its memory accesses is a cache hit, and because there are no branches in its code, the kernel never has a pipeline stall. Thus context switches always complete within a fixed execution time. Further, the kernel explicitly controls the state of all micro-architectural features, using partitionID register for the cache, and relying on the processor skeleton to reset the state of the pipeline, memory controller and the I/O bus-controller when a lease timer expires. Finally, while not implemented in our prototype, the kernel can optionally save and restore the entries on the lease stack. Saving the lease stack for each partition allows schedulers that run inside partitions to use different scheduling granularities than that of the kernel, without being aware of the time bounds that they themselves run within.

Fine-grained Scheduling. We propose a novel hardware stack implementation that allows execution time and memory bounds to have arbitrary durations and yet is verifiably secure at the bit level. The time and space bounds for each

partition are stored in a hardware stack in the CPU skeleton. This stack has to verifiably ensure that code inside a partition cannot over-write its own lease bounds, i.e. the current stack pointer should never affect any stack entry below the top of stack. The implementation in Execution Leases [69] protects lower stack entries by bit-encoding the timer values (e.g. each bit represents 32 cycles), and constructing each timer entry to have less time units than the previous entry. This restricts scheduling granularities to be aligned with the bit-encoding and introduces artificial performance overheads.

We provide a more flexible stack implementation by encoding the safety property in the logic for the *stack pointer* instead of the timer values (Figure 5.4). By choosing the stack pointer as the critical state, we free up the timers themselves to be assigned arbitrary values. We encode the stack pointer so that each bit corresponds to a stack entry, and on every clock cycle, the value assigned to each bit of the stack pointer is predicated upon *all* the lower stack pointer bits being true; i.e. a stack entry is valid only if all timers lower than itself are still active. Thus while information clearly flows from lower stack pointer bits to higher ones, higher bits never affect the lower ones. Our two information flow control techniques can be observed here: while timer[0] is permanently trusted, the rest of the stack is time-multiplexed between both security levels.

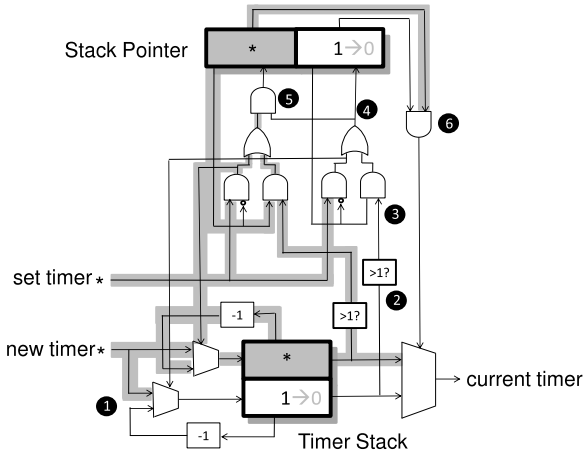

Figure 5.4: Implementing flexible timers with bit-level isolation. Isolation: Untrusted code (in gray) does not taint the lowest stack entry (1), and when the timer expires, the `current timer` value becomes trusted (path 2-3-4-6). A timer, when it expires, resets all timers above itself (5) and thus enforces nested leases. Flexibility: Instead of timer values, the `stack pointer` is encoded to ensure bit-level isolation. Hence the actual timers can be assigned arbitrary values.

We modified the `set_timer` instruction to receive arbitrary values as the time limit, and added a `mode` bit that a caller can use to specify whether the callee executes in kernel or user mode. This bit is set to 0 by the kernel when it schedules a user-space code in order to protect the partition_ID register. Our CPU implementation has three separate lease stacks of 2 entries each, one for execution time (PC) and one each for local and global memory bounds.

Read/Write Protection: Information flow policies such as non-interference [31], Biba [17], and Bell & La Padula [16] allow information to flow along one direction in a security lattice, e.g trusted parameters can be read but not tampered with. In order to support security policies that allow uni-directional informa-

tion flow, we modify the lease implementation to support read-only, write-only, or read-write control over memory bounds. In contrast, the original implementation of Leases allowed complete access to memory regions that are specified as part of the lease and could only support isolation in memory. The kernel uses the instruction `setmembound global/local, timer, addr_range, RO/WO/RW` to specify whether the memory region is Read-Only, Write-Only, or Read-Write, and the memory controller enforces these permissions at run-time.

To enforce isolation, we specify non-overlapping memory bounds for the partitions[1]. To implement communication, one option is for two communicating partitions to share a memory region. This requires the partitions to be initialized so that communicating partitions are adjacent. Since our CPU implements two distinct memory regions per partition, one partition can share memory areas with a maximum of four other partitions to facilitate zero-copy communication. For more complex communication patterns, the microkernel has to move data into partition "inboxes".

[1]Embedded systems typically operate exclusively on physical memory, eliminating channels associated with both aliasing and dynamic allocation

5.1.3 I/O: Using Off-The-Shelf Protocols and Devices Securely

We complete our embedded system by implementing an I/O protocol to connect the CPU and separation kernel system to peripheral devices. I^2C is a serial two-wire bus protocol that is commonly used in many embedded systems, for e.g. to configure RF tuners, video decoders and encoders and audio processors. It is also present on chipsets designed by Philips, National Semiconductor, Xicor, Siemens, and many others [35]. We show, for the first time, that it is possible to implement a provably secure I^2C master controller that can then interface with commodity I^2C devices.

Implicit information leaks occur when, for example, the I^2C bus master first communicates with an untrusted slave, and then with a trusted slave. The master's current state will first depend on information received from the untrusted slave, and appear as a covert channel underneath the ISA to software, where the untrusted slave affects the timing of trusted communication. At the gate-level, this implicit flow takes the form of an explicit ACK message from the untrusted slave to the trusted master's state machine, causing the master's state machine to be labeled untrusted. Thus even if the bus master seems to be behaving "cor-

rectly" and the devices are not snooping on the bus, there are still information flows between devices on the bus.

Trusted Bus Adapter: To restrict these implicit flows, we propose to overlay a time division multiplexed (TDMA) schedule over an I^2C bus, and introduce *adapters* to connect external devices to the shared system bus (Figure 5.1). Each adapter's time slot is a trusted kernel parameter, and the adapters enforce that for each slot only the currently addressed device has access to the bus while the remaining adapters disconnect their corresponding slave devices. When a lease timer expires, the bus master state machine is reset to a trusted state to eliminate the implicit flows mentioned earlier. We implement the adapters to not only impose a TDMA schedule on the bus but also conform to the I^2C specification. The adapters do so by driving the clock signal to a device low when its slot has expired, and since the I^2C protocol doesn't rely on wall-clock time, the devices hold on to data until the I^2C clock goes high again. As a result, we can use *unmodified I^2C-compliant devices* in our I/O subsystem.

The CPU - I/O Interface: The I^2C Bus Master state machine is implemented in hardware as part of the CPU. Programs in each partition (device drivers local to each partition) use two instructions `in dest_reg, dev_addr` and `out dev_addr, src_reg` to transfer a 32b value between a register and an I^2C device that the partition is allowed to address. Since peripheral devices are typically

slower than a CPU, a device driver can use the instruction `i2c_on dest_reg` to record into `dest_reg` whether the I^2C bus is currently in the middle of a transmission.

We have described a system that manages the flow of unknown and untrusted bits such that arbitrary untrusted computation is tightly bounded by trusted bits. The challenge now is to verify that an implementation of the high-level description correctly enforces an information flow policy.

5.2 Results

Figure 3.2 shows our toolchain to analyze hardware designs written in behavioral Verilog or VHDL (so that hardware designers can use their tools of choice for design entry). Verilog/VHDL designs are synthesized using Synopsys Design Compiler into a gate-level netlist using the `and_or.db` library. The result of this synthesis is a netlist that consists of just AND, OR, and NOT gates along with registers and memory. This netlist is input to our abstraction tool, which replaces gates and bits of the netlist with their abstract counterparts and outputs the abstract netlist. The abstract netlist is then input to our augmentation tool that

generates information flow tracking logic for the abstract design to create the final netlist. Finally, the augmented design is simulated using hardware synthesis and simulation tools such as Altera Quartus.

5.2.1 CPU Implementation

This section presents implementation details of our CPU (*Star-CPU*) and compares its functionality and area-delay with prior work. We implemented the Star-CPU in Verilog, generated a gate-level netlist using Synopsys Design Compiler, and synthesized this using QuartusII v9.1 with Altera EP2S15F48C43 FPGA as the target device. The Star-CPU pipeline is single-issue, executes in-order, and has 4 stages (fetch, decode, execute, and commit/write-back). It has 8 general purpose registers, a mode bit to indicate kernel/user mode, and a partition ID register to record the current security context. The memory hierarchy includes a 2kB direct-mapped data cache, and 64kB each of instruction and data memory. The data cache is implemented on the FPGA using comparator logic and registers and requires one cycle if a memory access is a hit, while the memory is implemented using on-chip block RAMs that take two cycles to service a memory request. To emulate memory access latency in an ASIC implementation of the system, the memory controller is implemented to introduce an additional delay of 100 cycles. Without micro-architectural features such as branch predictors,

TLBs, and Out-of-Order execution, the Star-CPU pipeline stalls on each cache miss and requires the compiler to ensure that a register used in a conditional jump instruction has the desired value at least 4 instructions before it is used.

Area-Delay Comparison. Figure 5.6 quantifies the size and performance advantages of the Star-CPU against the Execution Lease CPU and against the Star-CPU with dynamic GLIFT logic (Star-GLIFT). The Star-CPU provides caches, pipelining, and kernel support beyond the Lease CPU in equivalent area and clock-frequency, and provides static security guarantees compared to Star-GLIFT in almost 1/4 the logic, 1/2 the memory, and 2X the clock-frequency.

The Star-CPU's base functionality is implemented in 5756 ALUTs (Adaptive Look-Up Tables in an Altera FPGA, where 1 ALUT corresponds very approximately to 9-12 gates), and while the base functionality in the Lease CPU requires only 1511 ALUTs, it requires 5040 ALUTs when the dynamic analysis logic is factored in [69]. Thus the Star-CPU replaces analysis logic overhead with a cache and pipeline logic. In terms of performance, the Star-CPU and Lease CPU have similar frequencies (99 MHz vs. 104MHz), but the unpipelined Lease CPU only commits one instruction every 5 cycles. Further, without a cache, every memory access in the Lease CPU goes to main memory.

Comparing the verified Star-CPU to Star-GLIFT, we observe that the Star-GLIFT CPU requires 23,956 ALUTs for logic and 2×133kB for state and state

Instruction	Description
set_timer R1, R2, R3	Set PC lease. Arguments R#: register or immediate. R1: timer, R2: restore PC, R3: kernel/user mode
set_membound R1, R2, R3	Set local or global memory bounds. R1: memory range, R2: timer, R3: read/write mode
set_partitionID Immediate	If mode == kernel, then partitionID = Immediate
last_PC [R1]	Mem[R1] = PC in commit stage when the last timer expired
jgtz /jump R2, R1	Jump if R1 >= 0 or unconditionally. PC = R2 or Memory[R2]
load/store/mov R2, R1	Immediate and register direct addressing modes
add,sub,lsh,rsh and,or,not,cmplt R1, R2, R3	ALU instructions. Register arguments
in/out R1, dev_addr	Read and write to I^2C transfer register
io_on R1	R1 = 1 if I^2C transaction is ongoing
no-op	No-op instruction

Figure 5.5: Figure shows the ISA for the Star-CPU

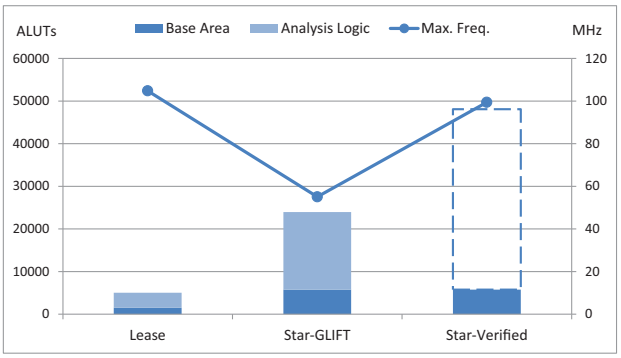

Figure 5.6: Area and Frequency comparison among secure CPUs.

labels, whereas the Star-CPU only requires 5756 ALUTs and 133kB for state. Adding dynamic tracking logic for the complex control logic of the CPU introduces substantial delays and reduces the maximum operating frequency of the Star-GLIFT CPU to 55MHz (from 99MHz for the verified Star-CPU). In summary, the verified Star-CPU provides better functionality than the Lease CPU, and static verification in comparison to Star-GLIFT CPU with much lesser area and delay.

5.2.2 Kernel Implementation

Our full-system prototype is representative of a high assurance avionics system [2]. Each trust domain in the system is assigned a partition, and a microkernel manages these partitions' access to the CPU, physical memory, and peripheral devices. The kernel implements an ARINC 653 scheduler (a standard in avionics systems) which requires that all partitions be statically defined at compile time in the form of a *major frame period* that repeats forever. Within a major frame, the schedule specifies one or more execution time slots for each partition, leaving the partitions free to implement standard, priority-based schedulers within their time slots.

Specifically, our prototype instantiates 4 partitions: the first partition to run programs responsible for controlling trusted avionics functions, the second parti-

tion for untrusted programs such as passenger internet and non-critical diagnostics, the third partition for a trusted cross-domain guard responsible for one-way communication among the above two partitions, and the fourth partition reserved for trusted kernel functions that require hardware access such as a context switch. To effect a context switch, the kernel partition gets one time slot after each of the other partitions' slots. In actual systems, the partitions are sized so that they meet hard real-time guarantees demanded by critical application; we opt for arbitrary durations for each partition in order to demonstrate how to verify non-interference between the trusted and untrusted partitions.

To verify non-interference, we instantiate the trusted kernel scheduler and the context switch partition with concrete values, while the other three partitions are instantiated as unknown ($*$). The cross-domain guard partition overlaps a read-only memory region with the trusted avionics partition and a write-only partition with the untrusted partition. This ensures that information can only flow in one direction from trusted to untrusted. Note, however, that cross-domain guards can also be required to impose restrictions on the *type* of information that can be transferred. Enforcing such rules requires verifying the guard program logic using alternative formal verification techniques; bit-level information flow analysis is too coarse-grained to provide such guarantees.

```
Context Switch 1: // from partition 0 to 1
    // save current state: 10 cycles
    set_partitionID  KERNEL_ID;
    store  ctxt_arr [PID0][0],  gen_reg [0]; …
    store  ctxt_arr [PID0][7],  gen_reg [7];
    last_PC  ctxt_arr [PID0][8];

    // restore incoming state: 9 cycles
    load  gen_reg [0],  ctxt_arr [PID1][0]; …
    load  gen_reg [7],  ctxt_arr [PID1][7];
    set_partitionID  PID1;
```

```
Kernel Scheduler 1: // uses reserved registers R1 and R2
    // set bounds and timers for partition 1: 10 cycles
    load R1, PID1_BOUNDS_G; load R2, PID1_TIME_G;
    set memboundsg R1, R2, PID1_RW_G;

    load R1, PID1_BOUNDS_L; load R2, PID1_TIME_L;
    set memboundsl  R1, R2, PID1_RW_L;

    load R1, PID1_TIMER; load R2, Context Switch2_PC;
    settimer  R1, R2, USR_MODE;

    jump  ctxt_arr [PID1][8];
    // returns to context switch2 () in kernel mode
```

```
Context switch time = 2 x pipeline depth + context save + context restore  + kernel scheduler   = 37 cycles
```

Figure 5.7: Kernel scheduler and context switch functions in assembly. Security policies are expressed through the values of partition parameters for memory and time bounds. The functions are small since the ISA and CPU are designed for information flow control.

The scheduler is small; the trusted, concretely specified scheduler and context switch code only requires **87** assembly instructions. This is primarily because the ISA in Figure 5.5 is explicitly designed for information flow control. The time required to switch contexts is an important performance metric for a kernel. In our system prototype, it takes the kernel partition **37 cycles** to switch one partition and schedule another: 4 cycles each to flush and re-fill the pipeline, 19 to save and restore partition ID, general purpose registers and the last executed PC, 10 to set new memory and time bounds and to jump to the PC for the restored context.

5.2.3 I/O Implementation

The I^2C devices and adapters are also processed using the verification flow mentioned above (note that the CPU and kernel can be verified independently of the I/O by treating the I/O interface in the CPU as trusted and unknown). Our experiment use a single master and three slaves connected to the bus using

adapters. Each adapter synthesized individually requires 49b of state and the slave requires 21b. We wish to verify that information does not flow from the untrusted slave to any other device, and set up a test where the master is trusted and known, communicates with an untrusted slave that is unknown ($*^U$). The I^2C bus has two trusted slaves, one specified and one unknown.

5.2.4 Verification Results

We simulate the augmented designs of the CPU and the I/O system for one loop of the kernel scheduler, come back to the initial state, and verify that all security labels for memory and outputs follow the desired policy for every state of the system. Figure 5.8 shows a screenshot of verifying that the I^2C adapter state is reset to trusted once an untrusted time slot has ended. We simulate a complete I^2C transaction, letting transmitted data values be unknown, and verify that no untrusted value ever appears on the adapter outputs to the trusted slave devices. This experiment used 3 slaves and 3 adapters with a total of 184 state bits, and if we assume that the hardware modules' contributions are proportional to their individual sizes, our technique can verify the 184b system by specifying ∼128b concretely and evaluating all combinations of the rest in a single execution of the augmented system.

The $*$-logic verification technique scales to handle large embedded system designs, as shown in Figure 5.6. Of the total 133kB state for the Star-CPU, only 3264b are required to specify the micro-kernel's scheduler, context switch code, and partition bounds. The verification scales because its complexity grows linearly with the size of the design under test, as each module is replaced by its augmented module.

Total verification time includes the time for both synthesizing the design and simulating it for a kernel scheduler loop. The augmented logic takes considerably longer to synthesize as compared to the basic design under test. The augmented Star-CPU, with 48093 ALUTs, required 14 hours to synthesize with QuartusII v9.1 as compared to just 7 minutes for the basic design with 5756 ALUTs. Once synthesized, simulating one loop of the kernel scheduler only takes a few seconds. All measurements were made on a 1GHz AMD Athlon 64 X2 Dual Core Processor with 1MB cache and 2.7GB RAM.

In the future, we will integrate $*$-logic with other formal techniques that can work with richer abstractions. While such techniques do not readily scale to large systems, these can complement $*$-logic to verify that the system is secure for a set of implementations or kernel parameter values instead of one specific implementation.

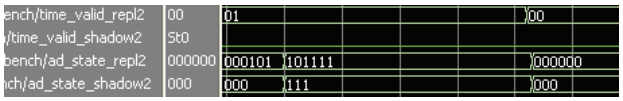

Figure 5.8: Figure shows how to check for a safe reset of the I^{2}C adapter to a trusted state (based on a trusted signal `time_valid`). The Modelsim simulation waveform begins with the adapter in an untrusted time slot (`time_valid = 1`). During the untrusted communication, the adapter's state bit stays unknown (as indicated by the MSB of `ad_state` being 1) and untrusted (`ad_state_shadow` = 1) until the time slot expires (indicated by the `time_valid` signal). At that point, the adapter's state machine is reset to a trusted state (indicated by the `ad_state_shadow` signal going low).

5.3 Conclusions

Embedded systems are trusted by people to do everything from stopping their cars to controlling the beating of their hearts, yet all too often these systems compromise strong security for rich functionality. We have shown, for the first time, that complete statically verifiable information flow security is compatible with the convenience of continuous, unbounded operation and dynamic optimizations – even when we consider timing channels and other hardware/software leaks as part of our threat model. Our system is designed around an architectural skeleton that allows a micro-kernel to safely multiplex mixed-trust programs on the hardware, and can do so in $1/4^{th}$ the area and with double the clock frequency as more restrictive prior work. These advances are due in part to the development of a tool that can statically verify information flows through full systems at the bit level, allowing us to verify a 133kB system by specifying only 3264b concretely,

and leaving behind the hardware dynamic flow tracking considered in prior work. While more work is required to examine the broad applicability and scalability of this approach, by implementing and verifying a full-system prototype (including a CPU, a micro-kernel, and I^2C based I/O) we have demonstrated that a useful balance between flexibility and hardware information leakage is not only possible but can even be relatively efficient.

Chapter 6

Conclusion

We conclude this dissertation by summarizing our key contributions and discussing the utility and trade-offs in gate-level information flow analysis.

Information flow security is a very versatile security model and captures the requirements of many high assurance systems where it is important to preserve the confidentiality of secret information or the integrity of critical system programs. In addition to information flow control, and not addressed in this dissertation, systems rely on encryption to preserve confidentiality while using an unclassified communication channel or even declassification functions to make some aggregated or outdated secret information available to an unclassified observer.

This work is a significant step towards realizing truly information flow secure systems. While information flows through application level logic and even operating system abstractions have been well-studied in the past, in the presence of a deliberately malicious program, identifying and closing *all* information leaks – be

they through timing of observable events or through obscure low-level hardware state – becomes critical to achieve high assurance. This dissertation presents the *first* method to provably analyze all digital information flows and then presents system level abstractions for realistic embedded systems to be built and verified as being completely information flow secure.

6.1 Contributions

Our first insight is that analyzing information flows at the lowest level of digital abstraction is the key to accounting for all explicit, implicit, and timing channels. By operating at the gate level, the entire computing system is brought under the purview of the information flow analysis and even leaks through obscure state or timing become observable to the security analysis.

A further insight, that makes this gate-level analysis become practical by not categorizing every mixed-trust system as being insecure, is recognizing that we can be more precise in propagating security labels through gates than for a generic system with mixed trust inputs. For example, labeling the output of an AND gate as trusted if a trusted input is 0 makes the information flow analysis precise and thus automatically recognize when a multiplexer is being used to reset some state back from untrusted to trusted. As a result, systems designed around a small

trusted kernel can be verified automatically if the kernel relies on multiplexers to safely manage the remaining state among various trust levels.

To complement this verification technique, we present two system-level abstractions that can be used to design and implement complete systems. The first abstraction – Execution Leases – is for the system programmer, where the hardware exports precise control over all information flows to the system programmer through specific instructions for bounding the space (all physical registers and memory) and time that are visible to the currently executing code. Using these `set_timer` and `set_memory_bound` instructions, a caller program can set up precise space-time sandboxes for the callee program to run in. The second abstraction is for the hardware designer to construct a system that allows performance enhancing features while still not leaking information through timing channels. We propose a small, trusted Architectural Skeleton that is used by a software micro-kernel to manage the entire remaining system state. The micro-kernel can use this skeleton to either partition all performance enhancing micro-architectural features such as caches and branch predictors or to overwrite all untrusted information before using the shared state for trusted programs.

In the end, we propose that a system designer use these abstractions to design the hardware-software root of trust and then verify the final gate-level implementation automatically using the $*$-logic tool. This would ensure that the formal

model of information flow security is applied systematically to the entire system state and the final bit-level system implementation conforms to a desired high-level security policy like non-interference. In the rest of this section, we discuss subtler issues that arise when using gate level analysis to verify practical systems.

6.2 Discussion 1: Utility of $*$-logic Verification Technique

$*$-logic is well suited for many high assurance systems, such as a router in an aircraft or a controller in automobiles, where the security requirements and the root of trust are known statically to a system designer.

A typical Integrated Modular Avionics system has to multiplex functions ranging from on-board diagnostics to navigation and control to passenger internet to possibly even some confidential data collection on one shared compute and communication infrastructure. The root of trust in such a system is some trusted program such as a real time operating system like Integrity or VxWorks running on a commodity CPU such as PPC 750, and this root of trust is responsible for enforcing a statically defined round-robin schedule, preserve the integrity of the navigation and control programs, and maintain the confidentiality of mission-related data. To verify such a system, the system designer has to concretely

specify its root of trust (including the kernel and the processor) and label rest of the state as unknown. The designer also specifies some parts of the state as untrusted (the passenger internet related programs), some other state as classified (programs related to a confidential mission), and even some state as "untrusted and unclassified" such as the on-board diagnostics that a ground mechanic can access for maintenance but not to sabotage the overall aircraft or to leak classified information. In effect, the entire system state has three additional labels (unknown, untrusted, and secret) to represent the three types of information that has to be tracked explicitly. Given such a specification, $*$-logic will work very well to verify whether the entire bit-level implementation of the system conforms to a desired security policy such as non-interference.

One important aspect of $*$-logic that concerns large scale system verification is the ability to specify some state as "unknown and trusted". This aspect is useful when a designer responsible for only a part of the system wishes to verify her subsystem for information flow security. For example, the I/O master controller may not be known to the person verifying the rest of the CPU and micro-kernel. In such cases, the designer can label the I/O controller as trusted and known, design her subsytem to be independent of the rest at the bit level, and then verify that the kernel and CPU are non-interfering for all possible values of the trusted

but unknown I/O controller state. Thus the system can be verified to be secure at the bit level in an incremental or distributed manner.

Another practically beneficial aspect of GLIFT is that the analysis allows a system to be verified (or augmented with dynamic analysis logic) even when a hardware or software design vendor only provides a gate-level netlist or binary to their customers. So for a designer assembling a System-On-Chip using multiple third-party hardware designs or a system with third-party software for less critical tasks (such as a web server), GLIFT can still guarantee security for the final system implementation. In contrast, techniques that analyze the source code statically (aur augment the code under test with dynamic analysis code) are unsuitable when systems are implemented using off-the-shelf netlists and binaries.

Weaknesses: $*$-logic requires trusted system constants that are required for information flow control, such as the execution time schedule of different trust domains, to be known concretely. Ideally, we would like the system to be verified as secure independently of the exact schedule values, but $*$-logic requires these values to be concrete since these are used as the select signal to multiplexers that in turn reset system state back to trusted values. Due to the simple three-level abstraction used for each bit (0, 1, or $*$), the analysis cannot reason about integer arithmetic and recognize that a $*$ counter value when decremented is bound to reach 0. Hence, to be verified as secure, the microkernel has to be specified

concretely including all system constants. If the kernel parameters such as the schedule is changed, the designer has to re-verify the system. In order to resolve this issue, further work is required to integrate $*$-logic with static analyses that can reason about higher levels of abstraction such as integers.

6.3 Discussion 2: Higher Level Design with Gate-level Verification

The core idea behind GLIFT is that all information flows are captured when all digital behaviors are observable, for example by representing the entire system design as a synchronous state machine. Both the GLIFT dynamic analysis and verification techniques use this insight to directly analyze a gate level system description. An alternative method for analyzing information flows would be to work at the level of a higher level hardware description language (HDL) instead of a gate-level description.

The benefit of working at the HDL level is that all flows can still be captured but the designer can use higher level language features such as procedural blocks, if-else and case statements, or even state machine level descriptions, instead of designing hardware tediously using only combinatorial logic. Using a compiler that verifies information flow properties through an HDL design, the designer

gets quick design time feedback and can iterate amongst different system design options that balance performance and area overheads while being verifiably secure. Further, the compiler can employ advanced program analysis techniques and speed up the design process through precise debugging hints. In contrast, the current analysis tool requires the design to be synthesized and simulated before the security vulnerabilities can be analyzed. The second version of our toolchain, which takes high level HDL desriptions as input and uses Synopsys Design Compiler to generate a gate-level description, allows for convenient design but still leaves debugging as a hard problem in case information flow violations are found.

In our design experience, we have found two design patterns (either assigning state and signals statically amongst trust domains or leasing them to untrusted domains with a trusted reset) to be used in different combinations for all the designs we implemented. As a result, we have begun looking into providing these patterns as a domain specific language over a popular HDL like Verilog and thus to aid the design of verifiably secure hardware designs.

As our first step we have designed Caisson [50], a domain specific language that combines the state machine abstraction for hardware designs with insights from type-based techniques used in secure programming languages. Specifically, Caisson allows a designer to represent the Execution Lease mechanism as a state machine where the trusted kernel state (the "master" state) can schedule the

machine-state bits among different trust domains (each in their own "slave" states) with no bits of information leaking between any pair of slave states.

This first version of a secure HDL proved to be useful because once the design patterns are formalized, the language allows a designer to quickly iterate through multiple system designs. For example, we designed a CPU where the pipelined stages are not leased for a longer length of time, but are scheduled among different trust domains on a per-cycle basis. In this design, every pipeline stage alternates between executing trusted and untrusted inputs and storing to trusted and untrusted outputs. While such a design is reminiscent of how many multi-threaded systems manage individual hardware threads (here, each thread here represents a unique trust domain), such fine-grained sharing of hardware resources in a verifiably secure pipelined processor is a first.

The drawbacks of this HDL based approach is that this does not account for software, such as the kernel, that is responsible for managing multi-trust functionality in most practical systems. Static verification of only the hardware is limited to computing systems that have fixed functionality, and such systems are increasingly becoming rare even in avionics and automobile domains. Further, in many large scale systems, a designer may not have access to all the hardware modules' source code. GLIFT and $*$-logic, on the other hand, can verify gate-level netlists without requiring access to source code. Finally, an HDL level technique

add adds the synthesis tool into the trusted computing base. This is a significant drawback for systems that have to be certified for high assurance use, since it adds the complexity of verifying that the compiler itself does the verification and code generation tasks correctly.

Ideally, we aim to have both ease of design and low cost of verification while being completely information flow secure. Towards this end, we anticipate that a designer will create hardware and software using a high level tool and get quick feedback, and then verify the final gate-level implementation using GLIFT or $*$-logic without having to trust the compiler from high level code to a gate level netlist.

6.4 Discussion 3: On Static vs Dynamic Verification

Fundamentally, complete information flow security precludes conditional actions based on an untrusted predicate (outside of any Lease, that is) as the effects of such conditional actions would label the entire system as untrusted. However, complete information flow security can be obtained by systems that require dynamic actions to only be performed by trusted programs. In practice this flexibility is sufficient for systems that rely on a trusted kernel to take interrupts at

arbitrary times, instead of polling for interrupts according to a fixed schedule, and also for systems that require the security label lattice to be updated based on some trusted intructions.

Based on this observation and the two discussions above, the ideal analysis technique would use dynamic analysis to enable maximum flexibility in security policies and system design, while using static analysis to reduce the area and run-time performance overhead of the dynamic analysis. We envisage the system to comprise of hardware that only provides mechanisms for efficient label storage and propagation, e.g. by implementing an efficient label store [72, 68] and programmable label propagation logic as a look-up table [72, 24] respectively, while the software is responsible for setting the labels and policies and taking appropriate action in case of a security violation. To reduce the overhead of dynamic tracking logic, the hardware designer can explicitly specify the Architectural Skeleton (Chapter 5) using an HDL for which precise GLIFT logic is generated automatically, while for the remaining system, an imprecise and conservative but much smaller tracking logic is employed. Further, the hardware designer can specify the programmer visible state (registers and memory) for which the information flow policies actually have to be enforced. Finally, to make the design phase efficient, the designer can use a variant of Caisson that allows her to work at a high level of abstraction and get quick design time feedback. Together, these techniques

will allow the designer to efficiently construct systems that add little additional overhead for tracking and enforcement of security policies, allow operating system and application level security policies to be guaranteed down to the bits, and impose no restrictions on system functionality beyond the ones imposed by complete information flow security.

Bibliography

[1] 90nm generic cmos library , synopsys university program, syn opsys inc., available at.

[2] Arinc 653. http://www.lynuxworks.com/solutions/milaero/arinc-653.php.

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

[4] The integrity real-time operating system. http://www.ghs.com/products/rtos/integrity.html.

[5] This car runs on code. http://news.discovery.com/tech/toyota-recall-software-code.html.

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

[7] 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.

[8] O. Accigmez, J. pierre Seifert, and C. K. Koc. Predicting secret keys via branch prediction. In *Cryptology, The Cryptographers Track at RSA*, pages 225–242. Springer-Verlag, 2007.

[9] O. Acicgmez. Yet another microarchitectural attack: Exploiting i-cache. In *14th ACM Conference on Computer and Communications Security (ACM CCS)*, 2007.

[10] O. Aciiçmez, Çetin Kaya Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In *The Cryptographers Track at RSA Conference*, pages 225–242, 2007.

[11] T. Alves and D. Felton. TrustZone: Integrated Hardware and Software Security, July 2004.

[12] A. Askarov, D. Zhang, and A. 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, New York, NY, USA, 2010. ACM.

[13] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2010.

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

[15] G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for java. In *27th IEEE Symposium on Security and Privacy*, 2006.

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

[17] K. Biba. Integrity considerations for secure computer systems, 1977.

[18] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[19] H. Chen, X. Wu, L. Yuan, B. Zang, P. chung Yew, and F. T. Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. *isca*, 0:401–412, 2008.

[20] J. Clause, W. Li, and A. 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.

[21] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 133–147. ACM Press, 2005.

[22] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.

[23] J. Daemen and V. Rijmen. The design of rijndael: Aes - the advanced encryption standard. 2002.

[24] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *34th Intl. Symposium on Computer Architecture (ISCA)*, June 2007.

[25] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[26] D. 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.

[27] X. Feng, Z. Shao, Y. Dong, , and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI*, pages pages 170–182, 2008.

[28] 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.

[29] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In *Cryptographic Hardware and Embedded Systems*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer-Verlag, 2001.

[30] S. Gianvecchio and H. Wang. Detecting covert timing channels: an entropy-based approach. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 307–316, New York, NY, USA, 2007. ACM.

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

[32] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In

Proceedings of the 2008 IEEE Symposium on Security and Privacy, pages 129–142, Washington, DC, USA, 2008. IEEE Computer Society.

[33] D. Jackson. A direct path to dependable software. *Commun. ACM*, 52(4):78–88, 2009.

[34] T. Jaeger, R. Sailer, and Y. Sreenivasan. Managing the risk of covert information flows in virtual machine systems. In *s. ACM Symposium on Access Control Models and Technologies (SACMAT)*, France, June 2007.

[35] D. Kalinksy and R. Kalinsky. Introduction to i2c. *Embedded Systems Programming*, 14(8), August 2001. http://www.embedded.com/story/OEG20010718S0073.

[36] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the vax vmm security kernel. *IEEE Trans. Softw. Eng.*, 17(11):1147–1165, 1991.

[37] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, New York, NY, USA, 2009. ACM.

[38] P. Kocher, J. J. E, and B. Jun. Differential power analysis. In *Advances in Cryptology*, pages 388–397. Springer-Verlag, 1999.

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

[40] J. Kong, O. Aciiçmez, J.-P. Seifert, and H. Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proc. of the 2nd ACM workshop on Computer security architectures*, pages 25–34, 2008.

[41] J. Kong, J. pierre Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *15th IEEE International Symposium on High Performance Computer Architecture*, 2009.

[42] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Eoxperimental Security Analysis of a Modern Automobile. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[43] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2010.

[44] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. *SIGOPS Oper. Syst. Rev.*, 41(6):321–334, 2007.

[45] L. C. Lam and T. 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.

[46] U. E. Larson and D. K. Nilsson. Securing vehicles against cyber attacks. In *Proceedings of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead*, CSIIRW '08, pages 30:1–30:3, New York, NY, USA, 2008. ACM.

[47] G. le Guernic and T. Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *ACM Conference on Computer and Communications Security (CCS'09)*, 2009.

[48] R. B. Lee, P. C. S. Kwan, J. P. Mcgregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005.

[49] J. Lewis. Cryptol: specification, implementation and verification of high-grade cryptographic applications. In *Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, page 41. ACM, 2007.

[50] X. Li, M. Tiwari, J. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. Caisson: a hardware description language for secure information flow. In *PLDI*, pages 109–120, 2011.

[51] W. Martin, P. White, F. Taylor, and A. Goldberg. Formal construction of the mathematically analyzed separation kernel. *Automated Software Engineering, International Conference on*, 0:133, 2000.

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

[53] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and Visualizing Full Systems with Data Flow Tomography. In *ASPLOS-XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2008.

[54] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[55] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology - CT-RSA 2006*, pages 1–20. Springer-Verlag, 2006.

[56] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003.

[57] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, New York, NY, USA, 2009. ACM.

[58] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, New York, NY, USA, 2009. ACM.

[59] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI 2009)*, Dublin, June 2009.

[60] J. Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.

[61] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. 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.

[62] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

[63] K. Shimizu, H. P. Hofstee, and J. S. Liberty. Cell broadband engine processor vault security architecture. *IBM J. Res. Dev.*, 51(5):521–528, 2007.

[64] O. Sibert, P. A. Porras, and R. Lindell. An analysis of the intel 80x86 security architecture and implementations. *IEEE Transactions on Software Engineering*, 22(5):283–293, 1996.

[65] G. Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 297–307. Springer-Verlag, 2007.

[66] G. Suh, C. O'Donnell, and S. Devadas. Aegis: A single-chip secure processor. *Design and Test of Computers, IEEE*, 24(6):570–580, Nov.-Dec. 2007.

[67] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[68] M. Tiwari, B. Agrawal, S. Mysore, J. K. Valamehr, and T. Sherwood. A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags. In *Proceedings of the International Symposium on Microarchitecture (Micro)*, 2008.

[69] M. Tiwari, X. Li, H. Wassel, F. Chong, and T. Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009.

[70] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[71] 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 *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254. IEEE Computer Society, 2004.

[72] G. Venkataramani, I. Doudalis, Y. Solihin, and M. 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.

[73] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. February 2007.

[74] M. Völp, C.-J. Hamann, and H. Härtig. Avoiding timing channels in fixed-priority schedulers. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ASIACCS '08, pages 44–55, New York, NY, USA, 2008. ACM.

[75] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[76] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput. Archit. News*, 35(2):494–505, 2007.

[77] M. Wolf, A. Weimerskirch, and T. Wollinger. State of the art: Embedding security in vehicles. *EURASIP Journal on Embedded Systems*, 2007.

[78] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *ISSTA*, pages 185–195, 2007.

[79] W. Xu, S. 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.

[80] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *PLDI*, 2010.

[81] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. 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.

[82] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006.