Protection Primitives for Reconfigurable Hardware

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

Doctor of Philosophy

in

Computer Science

by

Theodore Douglas Huffmire

Committee in Charge:

Professor Timothy Sherwood, Chair
Professor Frederic Chong
Professor Ryan Kastner

June 2007
The Dissertation of
Theodore Douglas Huffmire is approved:

________________________________________
Professor Frederic Chong

________________________________________
Professor Ryan Kastner

Professor Timothy Sherwood, Committee Chairperson

May 2007
Protection Primitives for Reconfigurable Hardware

Copyright © 2007

by

Theodore Douglas Huffmire
To David.
The Dream Keeper

“Bring me all of your dreams
You dreamers
Bring me all of your
Heart melodies
That I may wrap them
In a blue cloud-cloth
Away from the too-rough fingers
Of the world.”


Dreams

“Hold fast to dreams
For if dreams die
Life is a broken-winged bird
That cannot fly.

Hold fast to dreams
For when dreams go
Life is a barren field
Frozen with snow.”

Acknowledgements

I would like to thank Cynthia Irvine and Timothy Levin of the Naval Postgraduate School’s Center for Information Systems Security Studies and Research (NPS CISR) for their insightful comments on this dissertation. I also wish to thank Andrei Paun and Jason Smith of Louisiana Tech University for providing me with a Linux-compatible version of Grail+. I would like to thank Janet Kayfetz for helping me to refine my writing and presentation styles. I also wish to thank my committee for providing their comments. Finally, I want to thank my advisor, Tim Sherwood, for his exceptionally helpful feedback, advice, and guidance.

This research was funded in part by National Science Foundation Grant CNS-0524771, NSF Career Grant CCF-0448654, and the SMART Defense Scholarship for Service.
Curriculum Vitae

Theodore Douglas Huffmire

Education

1997  A.B. in Computer Science, Princeton University

Experience

2006 – 2007  Department of Defense Fellow, UC Santa Barbara

2004 – 2006  Graduate Research Assistant, UC Santa Barbara

2002 – 2004  Teaching Assistant, UC Santa Barbara

1997 – 2002  Member of Technical Staff, Epson Palo Alto Laboratory

Selected Publications


Ted Huffmire, Shreyas Prasad, Tim Sherwood, and Ryan Kastner: “Policy-Driven Memory Protection for Reconfigurable Hardw-

Abstract

Protection Primitives for Reconfigurable Hardware

Theodore Douglas Huffmire

Reconfigurable hardware is at the heart of many high-performance embedded systems. Satellites, set-top boxes, electrical power grids, and the Mars Rover all rely on Field Programmable Gate Arrays (FPGAs) to perform their respective functions. Despite the proliferation of reconfigurable devices into critical systems, sound reconfigurable system security remains an unsolved challenge. An FPGA system often has multiple modules (cores) on the same chip that share external resources such as off-chip memory, and these cores operate at different trust levels. While this enables small form factor and low-cost designs, it opens up the opportunity for modules to intercept or even interfere with the operation of each other. Providing a low-cost means to ensure logical isolation of modules is our primary goal, and we will leverage the reconfigurable nature of FPGAs to our advantage in solving this problem.

We propose a novel approach to reconfigurable system security that relies on both static and runtime techniques that work together to isolate the cores. The first element of our isolation strategy is a reference monitor, a runtime mechanism that enforces policies that specify the legal sharing of memory. These policy
specifications are expressed as a formal language, and a compiler translates them to a hardware description that can be directly transferred to an FPGA. Our language is powerful enough to express a variety of classic security scenarios. The second element of our strategy is a static technique that uses physical isolation to prevent unintended information flows by surrounding each core with a “moat” that blocks wiring connectivity from the outside. The third element is the detection of possible covert channels in stateful policies by statically analyzing the policy enforced by the reference monitor. This helps to prevent the use of the reference monitor as a covert channel. The fourth element is to make the construction of policies as accurate as possible by providing the embedded systems designer with a higher-level language for expressing security concepts as well as a set of tools that use formal methods to ensure that a policy under construction is mathematically precise.
# Contents

Acknowledgments vi
Curriculum Vitæ vii
Abstract ix
List of Figures xiv
List of Tables xx

1 Introduction 1
   1.1 The Need for Reconfigurable System Security ............... 1
   1.2 Policy-Driven Memory Protection .......................... 6
   1.3 Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems .................... 8
   1.4 Detecting Covert Channels in Stateful Policy Enforcement Systems 9
   1.5 Expressing Security Policies Precisely ................ 10

2 Reconfigurable Systems 12
   2.1 Introduction ............................................... 12
   2.2 Architecture of a Reconfigurable System .................. 13
   2.3 Reconfigurable Devices and Security ........................ 15
   2.4 Mixed-Trust Design Flows .................................. 17
       2.4.1 Motivating Examples ................................. 19
   2.5 Spatial versus Temporal ................................... 21

3 Policy-Driven Memory Protection for Reconfigurable Systems 22
   3.1 Introduction ............................................... 22
3.2 Protecting Memory on an FPGA ................................................. 27
3.3 Policy Description and Synthesis ............................................. 29
  3.3.1 Memory Access Policy ...................................................... 30
  3.3.2 Hardware Synthesis ......................................................... 36
  3.3.3 Design Flow Details ......................................................... 37
3.4 Example Applications .......................................................... 43
  3.4.1 Access Control List .......................................................... 44
  3.4.2 Controlled Sharing ........................................................... 45
  3.4.3 Chinese Wall ................................................................. 46
  3.4.4 Redaction ..................................................................... 48
  3.4.5 Bell and LaPadula Confidentiality Model ............................... 50
  3.4.6 High Water Mark ............................................................. 51
  3.4.7 Biba Integrity Model ......................................................... 52
  3.4.8 Dynamic Policies ............................................................ 53
3.5 Integration and Evaluation .................................................... 55
  3.5.1 Enforcement Architecture .................................................. 55
  3.5.2 Evaluation ..................................................................... 57
  3.5.3 Synthesis Results ........................................................... 58
  3.5.4 Impact of the Reference Monitor on System Performance ............. 62
3.6 Summary ............................................................................. 63

4 Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems ......................................................... 66
  4.1 Introduction ........................................................................ 66
  4.2 Reconfigurable Systems ....................................................... 69
    4.2.1 Reconfigurable Hardware ................................................ 70
  4.3 Physical Isolation with Moats .............................................. 71
    4.3.1 Building Moats ............................................................... 75
    4.3.2 A Performance/Area Trade-off ......................................... 77
    4.3.3 The Effect of Constrained Routing ................................... 78
    4.3.4 Overall Area Impact ......................................................... 80
    4.3.5 Effective Scrubbing and Reuse of Reconfigurable Hardware ..... 83
  4.4 Drawbridges: Interconnect Interface Conformance with Tracing ......... 87
    4.4.1 Efficient Communication under the Drawbridge Model ........... 89
    4.4.2 Architecture Alternatives ............................................... 92
  4.5 Application: Memory Policy Enforcement .................................. 95
  4.6 Related Work .................................................................... 97
    4.6.1 Reconfigurable Hardware Security .................................. 98
    4.6.2 Covert Channels, Direct Channels, and Trap Doors ............... 101
List of Figures

1.1 Alternative strategies for providing protection on embedded systems 7
2.1 A Modern FPGA-based Embedded System .......................... 14
2.2 Design Flow .................................................................. 17

3.1 Parse tree of the simple access policy .............................. 38
3.2 Expanded parse tree ...................................................... 39
3.3 NFA derived from the regular expression .......................... 40
3.4 NFA converted to a minimized DFA ................................. 40
3.5 Enforcement module ...................................................... 42
3.6 A Chinese wall policy .................................................... 46
3.7 DFA that recognizes legal accesses for the Chinese Wall policy 47
3.8 A redaction architecture ................................................ 49
3.9 Two alternative architectures for the enforcement mechanism 56
3.10 Setup time and cycle time ............................................. 58
3.11 Circuit area versus number of ranges .............................. 59
3.12 Cycle time versus number of ranges ................................. 59
3.13 Setup time versus number of ranges ............................... 60
3.14 Circuit area versus access policy .................................... 61
3.15 Cycle time for each access policy ................................... 61
3.16 Setup time for each access policy ................................... 62

4.2 FPGA fabric and floor plan ............................................. 72
4.3 A simple two-core system mapped onto a small FPGA ........ 74
4.4 Moats on an FPGA with routing restricted to segments of length one and two ......................................................... 75
4.5 Comparison of area for different configurations of routing segments 80
4.6 Comparison of critical path timing for different configurations of routing segments ........................................... 81
4.7 The trade-off between the number of cores, the size of the moat, and the utilization of the FPGA ........................................... 81
4.8 Architecture alternative 1 ........................................... 90
4.9 Architecture alternative 2 ........................................... 90

5.1 A non-trivial cycle ........................................... 113
5.2 A non-trivial cycle where the sender only causes a subset of the transitions ........................................... 114
5.3 A cycle consisting of three nodes ........................................... 115
5.4 A redaction policy with four possible covert channels ........................................... 119
5.5 A dynamic policy that switches between a B&L policy and a Biba policy ........................................... 119
5.6 A transitive property of covert channels ........................................... 120
5.7 A Chinese wall policy that can leak two bits of information ........................................... 121
5.8 Coping with a covert channel in a stateful policy with two states ........................................... 125

6.1 An isolation policy ........................................... 133
6.2 A controlled sharing policy ........................................... 134
6.3 An access list policy ........................................... 136
6.4 A Chinese wall policy ........................................... 137
6.5 A redaction policy ........................................... 138
6.6 A Bell and LaPadula policy ........................................... 139
6.7 A high water mark policy ........................................... 141
6.8 A bible policy ........................................... 142
6.9 A low water mark policy ........................................... 144
6.10 A dynamic policy in which returning to an earlier policy is not allowed ........................................... 145
6.11 A dynamic policy in which returning to an earlier policy is allowed ........................................... 147
6.12 A “toy” isolation policy ........................................... 152
6.13 A Venn Diagram that illustrates the logic behind our scheme ........................................... 155
6.14 An an automated approach to the incremental construction of policies ........................................... 156
6.15 DFA that recognizes the language (A|B|C)* ........................................... 157
6.16 DFA that recognizes the language (C|D|E)* ........................................... 157
6.17 DFA that recognizes the language C* ........................................... 157
6.18 DFA that recognizes legal accesses for a Chinese Wall policy ........................................... 157
6.19 DFA that recognizes illegal access for a Chinese wall policy ........................................... 158
List of Tables

4.1 Reconfiguration Time ........................................... 85
4.2 Comparison of Communication Architectures .................. 94
Chapter 1

Introduction

The mind is not a vessel to be filled but a fire to be lighted
Plutarch (c. 46-127), On Listening to Lectures

1.1 The Need for Reconfigurable System Security

Blurring the line between software and hardware, reconfigurable devices strive to strike a balance between the raw high speed of custom silicon and the post-fabrication flexibility of programmable processors. While this flexibility is a boon for embedded system developers, who can now rapidly prototype and deploy solutions with performance approaching custom designs, in reality this results in a system development methodology where functionality is stitched together from a variety of “soft IP cores,” often provided by multiple vendors with potentially multiple levels of trust. The problem is that, unlike traditional software where
resources are managed by an operating system, soft IP cores necessarily have very fine grain control over the underlying hardware. To address this problem, the embedded systems community requires novel security primitives which address the realities of modern reconfigurable hardware.

Reconfigurable hardware, such as a Field Programmable Gate Array (FPGA), provides a programmable substrate onto which descriptions of circuits can be loaded and executed at very high speeds. Because they are able to provide a useful balance between performance, cost, and flexibility, many critical embedded systems make use of FPGAs as their primary source of computation. For example, the aerospace industry relies on FPGAs to control everything from satellites to the Mars Rover. Their circuit-level flexibility allows system functionality to be updated arbitrarily and remotely. Real-time and military projects, such as the Joint Strike Fighter, make frequent use of FPGAs because they provide both high-performance and well-defined timing behavior, but they do not require the costly fabrication of custom chips.

FPGA technology is now the leading design driver for almost every single foundry \footnote{A foundry is a wafer production and processing plant available on a contract basis to companies that do not have wafer fab capability of their own} meaning that they enjoy the benefits of production on a massive scale (reduced cost, better yield, difficult to tamper with), yet developers are free to deploy their own custom circuit designs by configuring the device in the appro-
Chapter 1. Introduction

appropriate ways. This has significantly lowered the primary impediment to hardware
development, cost, and as such we are now seeing an explosion of reconfigurable
hardware based designs in everything from face recognition systems [71], to wire-
less networks [77], to intrusion detection systems [39], to supercomputers [11]. In
fact it is estimated that in 2005 alone there were over 80,000 different commercial
FPGA designs projects started [64]. Unfortunately, while the economics of the
semiconductor industry has helped to drive the widespread adoption of reconfig-
urable devices in a variety of critical systems, it is not yet clear that such devices,
and the design flows used to configure them, are actually trustworthy.

Reconfigurable systems are typically cobbled together from a collection of ex-
isting modules (called cores) in order to save both time and money. Although
ideally each of these cores would be formally specified, tested, and verified by a
highly trusted party, in reality, such a development model cannot hope to keep
up with the exponential increases in circuit area and performance made possible
by Moore’s Law. Unlike uni-processor software development, where the program-
ning model remains fixed as transistor densities increase, FPGA developers must
explicitly take advantage of denser devices through changes in their design. Given
that embedded design is driven in large part by the demand for new features and
the desire to exploit technological scaling trends, there is a constant pressure to
mix everything on a single chip: from the most critical functionality to the latest
Chapter 1. Introduction

fad. Each of these cores runs “naked” on the reconfigurable device (i.e., without the benefit of an operating system or other intermediate layer), and it is possible that this mixing of trust levels could be silently exploited by an adversary with access to any point in the design flow (including design tools or implemented cores). In an unrestricted design flow, even answering the question of “are these two cores capable of communication” is computationally difficult to answer.

Consider a more concrete example, a system with two soft-processor cores and an AES encryption engine sharing a single FPGA. Each of these three cores requires access to off-chip memory to store and retrieve data. How can we ensure that the encryption key for one of the processors cannot be obtained by the other processor by either reading the key from external memory or directly from the encryption core itself? There is no virtual memory on these systems, and after being run through an optimizing CAD tool the resulting circuit is a single entangled mess of gates and wires. To prevent the key from being read directly from the encryption core itself, we must find some way to physically isolate the encryption engine from the other cores. To protect the key in external memory, we need to implement a memory protection module, we need to ensure that each and every memory access goes through this monitor, and we need to ensure that all cores are communicating only through their specified interfaces. To ensure these properties hold at even the lowest levels of implementation (after all the design tools
have finished their transformations), we argue that slight modifications in the design methods and tools can enable the rapid static verification of finished FPGA bitstreams\(^2\). The techniques presented in this dissertation are steps towards a cohesive reconfigurable system design methodology that explicitly supports cores with varying levels of trust and criticality – all sharing a single physical device.

**Thesis Statement:** While existing security primitives for FPGAs do not provide isolation or protection, mixed trust designs may be supported by exploiting both the spatial nature of computation on reconfigurable devices and the ability to integrate enforcement mechanisms. The primary contributions of this dissertation are:

- A novel language-based scheme for expressing security policies that can be translated directly to a hardware description of a reference monitor, a runtime enforcement mechanism that can be loaded onto an FPGA

- A static technique that uses physical isolation to prevent unintended information flows by surrounding each core with a “moat” in which routing is disabled.

---

\(^2\)bitstreams are the term for the detailed configuration files that encode the exact implementation of a circuit on reconfigurable hardware – in many ways they are analogous to a statically linked executable on a traditional microprocessor
Chapter 1. Introduction

- A static technique of automatically detecting possible covert channels in a stateful policy in order to prevent the use of the reference monitor as a covert channel

- A higher-level language for expressing security policies as well as a set of tools that use formal methods to ensure that a policy under construction is mathematically precise

1.2 Policy-Driven Memory Protection

Figure 1.1 shows three different strategies of providing protection for embedded systems. In an ideal world, every core would run on its own separate FPGA chip, but this is clearly a very inefficient way of separating cores. Instead, we propose a language-based approach that uses a reference monitor to enforce policies that specify the legal sharing of memory. In our design, a memory access policy is a formal description that establishes what accesses to memory are legal and which are not. Our method rests on the ability to formally describe the access policy using a specialized language. We present a compiler through which the policy description can be automatically transformed and directly synthesized to a circuit. This circuit, represented as a bit-stream, can then be loaded into a reconfigurable
Figure 1.1: Alternative strategies for providing protection on embedded systems. From a security standpoint, a system with multiple applications could allocate a dedicated physical device for each application, but economic realities force designers to integrate multiple applications onto a single device. Separation kernels use virtualization to prevent applications from interfering with each other, but they come with the overhead of software and are therefore restricted to general-purpose processor based systems. Our approach to providing protection for FPGA based embedded systems uses a reconfigurable reference monitor to enforce the legal sharing of memory among cores. We also exploit the spatial nature of computation on reconfigurable devices to provide strong physical isolation of cores.
hardware module and used as an execution monitor to analyze memory accesses and enforce the policy.

1.3 Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

A reference monitor must not be subverted or bypassed, and it must not be possible for two cores to establish a direct connection unless that connection is intended by the designer. To provide physical separation of cores as well as the reference monitor and to ensure that every memory access must go through the reference monitor, we propose a security primitive in which every core is surrounded by a “moat” that blocks wiring connectivity from the outside. The only way that a core that is surrounded by a moat to communicate with the outside world is via a “drawbridge,” a narrow interconnect pathway that can be accurately traced. A design can be statically verified to conform to a specification that describes the manner in which cores are connected with each other.

In modern reconfigurable systems, cores communicate with each other via a shared bus. Unfortunately, the shared nature of a traditional bus architecture
Chapter 1. Introduction

raises several security issues. Our drawbridge model also contains a shared memory bus with a time division access. The bus divides time equally among the modules, which makes it harder to use the shared bus as a covert channel. We also describe some of the security challenges of the partial reconfiguration feature of some of the latest FPGA devices, and we discuss how to exploit partial reconfiguration to deal with some extreme circumstances.

1.4 Detecting Covert Channels in Stateful Policy Enforcement Systems

A reference monitor is only as good as the policy it enforces, and some stateful policies may allow a malicious core to use the reference monitor as an unbounded covert channel. We describe an automatic method of detecting such storage channels in policies so that they can be eliminated during the design phase of an embedded system. If modifying the policy is not an option, we demonstrate the use of counters to measure the bandwidth of the covert channel at runtime and take corrective action if needed.
Chapter 1. Introduction

1.5 Expressing Security Policies Precisely

Any security mechanism is only as effective as the policy it is enforcing. If there is a mistake in the policy, the reference monitor will faithfully enforce a flawed policy. We wish to provide assurance that policies are correctly formed. Although our memory access language is highly precise, using it requires some expertise. To reduce the possibility of human error, we need a more abstract, higher-level language for engineers to work with. This higher-level language expresses security concepts such as isolation and controlled sharing, which the compiler accurately translates into our lower-level memory access language. We also provide tools to assist an embedded system designer in the construction of mathematically precise security policies. These tools make use of formal methods to ensure that policies are correctly formed.

Constructing mathematically precise policies is essential to sound security. In order for a policy to be precise, it must accept all behavior which is legal and reject all behavior which is illegal. Constructing policies can be challenging without an automatic way of verifying that the policy reflects the intent of the person creating that policy. Our methods make it possible to determine if there is any conflict between behavior that should be legal and behavior that should be illegal. We propose an automatic method of incremental construction of security policies that
Chapter 1. Introduction

is based on theoretical foundations. If a conflict is found between legal and illegal behavior, the system informs the person constructing the policy of the offending overlapping behavior.

The remainder of this dissertation is organized as follows: In Chapter 2, we provide background on reconfigurable systems. In Chapter 3, we describe a runtime technique that uses a reconfigurable reference monitor to provide policy-driven memory protection. In Chapter 4, we describe a static technique of surrounding each core with a “moat” that blocks wiring connectivity from the outside. In Chapter 5, we describe a static technique for analyzing stateful policies to detect possible covert channels. In Chapter 6, we describe a higher-level language and a set of tools to make the job of expressing security policies as accurate as possible. Finally, in Chapter 7, we conclude and provide some insight on the direction in which this field is likely to head.
Chapter 2

Reconfigurable Systems

To get back my youth I would do anything in the world, except take exercise, get up early, or be respectable.

Oscar Wilde (1854-1900), The Picture of Dorian Gray (1891)

Anyone who has to ask about the annual upkeep of a yacht can’t afford one.

J.P. Morgan (1837-1913)

2.1 Introduction

This chapter provides background on the nuts and bolts of FPGAs. Increasingly we are seeing reconfigurable devices emerge as the flexible and high-performance workhorses inside a variety of high performance embedded computing systems [10, 16, 19, 44, 63, 82]. The power of reconfigurable systems lies in the immense amount of flexibility that is provided. Designs can be customized down to the
level of individual bits and logic gates. They combine the post-fabrication programmability of software running on a general purpose processor with the spatial computational style most commonly employed in hardware designs [19]. Reconfigurable systems use programmability and regularity to create a flexible computing fabric that can lower design costs, reduce system complexity, and decrease time to market, while achieving 100x performance gain per unit silicon as compared to a similar microprocessor [14, 18, 96]. The growing popularity of reconfigurable logic has forced practitioners to start to consider the security implications, yet the resource constrained nature of embedded systems is a challenge to providing a high level of security [49]. To provide a security technique that can be used in practice, it must be both robust and efficient. To understand what is a practical design, we must first examine the architecture of a modern reconfigurable system.

2.2 Architecture of a Reconfigurable System

Field Programmable Gate Arrays (FPGAs) are the most common reconfigurable devices. An FPGA is a collection of programmable gates embedded in a flexible interconnect network. FPGAs use truth tables (known as lookup tables or LUTs) to implement logic gates, flip-flops for timing and registers, switchable interconnect to route logic signals between different units, and I/O blocks (IOB)
for transferring data into and out of the device. A circuit can be mapped to an FPGA by loading the LUTs and switch-boxes with a configuration, a method that is analogous to the way a traditional circuit might be mapped to a set of and or gates. Figure 2.1 shows a modern FPGA-based embedded system.

**Figure 2.1:** A Modern FPGA-based Embedded System: Reconfigurable logic, blocks of SRAM, and hard-wired microprocessors all share the same piece of silicon, and, more importantly, the same off-chip memory. The reconfigurable logic is a fabric of tiny lookup tables and statically scheduled routing hardware that can be configured to emulate almost any possible circuit.

LUTs employ static RAM cells as programming bits. A LUT is an extremely generic computational component. It can compute “any” function; i.e. any n-input LUT can be used to compute any n-input function. A LUT requires $2^N$ bits to describe, but it can implement $2^{2^N}$ different functions. LUTs are limited to a small number of inputs due to the size of SRAM cells as a programming point. A typical LUT has either 4 or 5 inputs, a number based on extensive empirical work
Chapter 2. Reconfigurable Systems

aimed at optimizing physical aspects of the FPGA architecture [7]. An FPGA is programmed using a bit-stream. This binary data is loaded into the FPGA to execute a particular task. The bit-stream contains all the parameters needed such as the configuration interface and the internal clock cycle supported by the device.

2.3 Reconfigurable Devices and Security

FPGAs provide a very important security benefit over ASICs. During the manufacture of an ASIC, the sensitive design is exposed to the risk of theft. Since most foundries are located overseas, this issue concerns the national interest. Trimberger explains how FPGAs transform the problem of trusting the foundry into the information security problem of preventing the design from being stolen from the FPGA itself [92].

FPGAs are a natural platform for performing many cryptographic functions because of the large number of bit-level operations that are required in modern block ciphers. While there is a great deal of work centered around exploiting FPGAs to speed cryptographic or intrusion detection primitives, researchers are now starting to realize the security ramifications of building systems around hardware which is reconfigurable. One major problem is that hardware, not just software, can now be copied from existing products, and there has been a flurry of research
to protect this intellectual property [12, 46, 52] and to secure the FPGA’s program logic update channels [34, 33]. However, few researchers have begun to consider the security ramifications of compromised hardware [31].

It is important to understand the different attacks against FPGAs that are possible in order to develop countermeasures [101]. In a covert channel attack, an observable property such as power consumption is analyzed by a malicious module in order to steal secrets such as cryptographic keys or the bit-stream contained in the FPGA, which is valuable intellectual property [88]. In some systems, the bit-stream can be modified remotely, and authentication mechanisms should be employed to prevent unauthorized users from uploading a malicious design, which could change the intended functionality of the device. Even worse, the malicious design could physically destroy the FPGA by causing the device to short-circuit [31]. Solutions to these problems include encryption [12] [45] [46], fingerprinting [51], and watermarking [52]. While there are a variety of attacks possible, our work is concerned with addressing the problem of memory protection on reconfigurable systems. In particular, in Chapter 3, we present techniques to provide separation while allowing controlled interaction between multiple interacting cores and modules with respect to their use of off-chip memory\footnote{The same approach is applicable to on-chip memory, but we leave this to future work.}. In Chapter 4 we discuss techniques to isolate the cores themselves so that they do not interfere with
Figure 2.2: Design Flow: Distinct cores with different pedigrees and varied trust requirements find themselves occupying the same silicon. Reconfigurable logic, hard and soft processor cores, blocks of SRAM, and other soft IP cores all share the FPGA and the same off-chip memory. How can we ensure that the encryption key for one of the processors cannot be obtained by the other processor by either reading the key from external memory or directly from the encryption core itself?

each other or leak data. In our attack model, there may be subverted modules or remote attacks that originate from the network through I/O, but we assume that the attacker cannot physically modify or monitor the device.

2.4 Mixed-Trust Design Flows

Figure 2.2 shows a few of the many different design flows used to compose a single modern embedded system. The reconfigurable implementation relies on a large number of sophisticated software tools that have been created by many
different people and organizations. Soft IP cores, such as an AES core, can be
distributed in the form of Hardware Description Language (HDL), netlists\(^2\) or a
bitstream. These cores can be designed by hand, or they can be automatically gen-
erated by computer programs. For example, the Xilinx Embedded Development
Kit (EDK) [103] software tool generates soft microprocessors from C code. Accel
DSP [35] translates MATLAB [90] algorithms into HDL, logic synthesis translates
this HDL into a netlist, a synthesis tool uses a place-and-route algorithm to con-
vert this netlist into a bitstream, with the final result being an implementation of
a specialized signal processing core.

Given that all of these different design tools produce a set of inter-operating
cores, you can only trust your final system as much as you trust your least-trusted
design path. If there is a critical piece of functionality, e.g. a unit that protects
and operates on secret keys, there is no way to verify that this core cannot be
snooped on or tampered without a set of isolation strategies.

The subversion of design tools could easily result in malicious hardware being
loaded onto the device. In fact, major design tool developers have few or no checks
in place to ensure that attacks on specific functionality are not included. However,
just to be clear, we are not proposing a method that makes possible the use of
subverted design tools on a trusted core. Rather, we are proposing a method by

\(^2\) Essentially a list of logical gates and their interconnections
which small trusted cores, developed with trusted tools (perhaps using in-house tools which are not fully optimized for performance) can be safely combined with untrusted cores.

2.4.1 Motivating Examples

Consider a system with two processor cores and an encryption core. One goal of our methods is to prevent the encryption key for one of the processors from being obtained by the other processor by either reading the key from external memory or directly from the encryption core itself.

Aviation — Both military and commercial sectors rely on commercial off-the-shelf (COTS) reconfigurable components to save time and money. Consider the example of avionics in military aircraft in which sensitive targeting data is processed on the same device as less sensitive maintenance data. In such military hardware systems, certain processing components are “cleared” for different levels of data. Since airplane designs must minimize weight, it is impractical to have a separate device for every function. Our security primitives can facilitate

---

FPGA manufacturers such as Xilinx provide signed cores that can be trusted by embedded designers, while those freely available cores obtained from sources such as OpenCores are considered to be less trustworthy. The development of a trusted tool chain or a trusted core is beyond the scope of this dissertation.
the design of military avionics by providing separation of modules that must be integrated onto a single device.

**Computer Vision** — In the commercial world, consider a video surveillance system that has been designed to protect privacy. Intelligent video surveillance systems can identify human behavior that is potentially suspicious, and this behavior can be brought to the attention of a human operator to make a judgment. IBM’s PeopleVision project has been developing such a video surveillance system that protects the privacy of individuals by blurring their faces depending on the credentials of the viewer (e.g., security guards vs. maintenance technicians). FPGAs are a natural choice for any streaming application because they can provide deep regular pipelines of computation, with no shortage of parallelism. Implementing such a system would require at least three cores on the FPGA: a video interface for decoding the video stream, a redaction mechanism for blurring faces in accordance with a policy, and a network interface for sending the redacted video stream to the security guard’s station. Each of these modules would need buffers of off-chip memory to function, and our methods could prevent sensitive information from being shared between modules improperly (e.g., directly between the video interface and the network). While our techniques could
not verify the correct operation of the redaction core, they could ensure that only
the connections necessary for legal communication between cores are made.

2.5 Spatial versus Temporal

The goal of spatial computing [20] is to interconnect operations in space rather
than time, exploiting parallelism to achieve high throughput. This is made pos-
sible by the bounty of transistors available on modern devices. Since computa-
tions are divided spatially, efficient communication between computing elements
is needed to exploit spatial locality. This can be achieved by placing frequently
communicating elements closer together in order to reduce the distance along crit-
ical paths. In Chapter 4, we exploit the spatial mapping of applications to the
device to provide isolation of multiple computing cores that reside on a single
device.
Chapter 3

Policy-Driven Memory Protection for Reconfigurable Systems

*Yea, from the table of my memory*
*I’ll wipe away all trivial fond records,*
*All saws of books, all forms, all pressures past,*
*That youth and observation copied there.*

William Shakespeare (1564-1616), Hamlet (c. 1600)

3.1 Introduction

Reconfigurable hardware is at the heart of many high performance embedded systems. Satellites, set-top boxes, electrical power grids, and the Mars Rover all rely on Field Programmable Gate Arrays (FPGAs) to perform their respective functions for everything from encryption to FFT, or even entire customized processors. The bit-level configurability of these devices can be used to implement specific logic circuits that are highly optimized compared to the processing required in a general-purpose CPU. Because the logic of the fabricated device is
reconfigurable, special-purpose circuits can be developed and deployed at a frac-
tion of the cost associated with custom fabrication (e.g., ASIC). Furthermore, the
logic on an FPGA board can even be changed in the field. These advantages
of reconfigurable devices have resulted in their proliferation into critical systems,
yet many of the security primitives which software designers take for granted in
general-purpose processors are simply nonexistent. In this chapter we present a
runtime security primitive that uses a reconfigurable reference monitor to enforce
the legal sharing of memory among multiple applications on a single FPGA de-
vice. Our scheme employs a specialized compiler to translate a memory access
policy specification to a hardware description of an enforcement mechanism that
can be integrated with the computing cores.

Due to Moore’s law, FPGAs today have enough transistors on a single chip
to implement over 200 separate RISC processors. Increased levels of integration
are inevitable, and reconfigurable systems are no different. Current reconfigurable
systems-on-chip include diverse elements such as specialized multiplier units, in-
tegrated memory tiles, multiple fully programmable processor cores, and a sea of
reconfigurable gates capable of implementing significant ASIC or custom data-
path functionality. The complexity of these systems and the lack of separation
between different hardware modules on the FPGA device has increased the possi-
bility that security vulnerabilities may surface in one or more components, which
could threaten the entire device. New methods are needed to provide separation and security in these highly integrated reconfigurable devices.

One of the most critical aspects of separation that needs to be addressed is in the management of external resources such as off-chip DRAM. While a general-purpose processor will typically provide virtual memory mapping primitives such as TLBs that are used to enforce some form of memory protection, reconfigurable devices usually operate in a flat physical address space with a flat program structure (e.g., without underlying operating system support). Lacking these mechanisms, the FPGA environment is assumed to be benign, since any hardware module can normally read or write to the memory of any other module at any time. Whether purposefully, accidentally, or maliciously, destructive interference between cores can result. This situation calls for a memory access policy and related control mechanisms that all modules on chip must obey. In this chapter we present a method that utilizes the reconfigurable nature of field programmable devices to provide a mechanism to enforce such a policy.

In the context of this chapter, a memory access policy is a description of what accesses to memory are legal and which are not. Our method rests on the ability to formally describe the access policy using a specialized language. The formalism results in two significant capabilities: the ability to reason about policy soundness and the ability to automatically derive refinements to the policy. We present a set
of tools through which the policy description can be automatically transformed and directly synthesized to a circuit. This circuit, represented as a bit-stream, can then be loaded into a reconfigurable hardware module and used as an execution monitor to analyze memory accesses of individual cores on the FPGA and enforce the memory access policy.

The techniques presented in this chapter are steps towards a cohesive methodology for those seeking to build reconfigurable systems that can securely control data at different sensitivity labels and modules acting at different security clearance levels on a single chip (i.e., systems that can provide multi-level security). In order for such a methodology to be accepted by the embedded design community it is critical that the resulting hardware provides both high performance and efficient use of the FPGA fabric. Within the security community, the methods must be formally grounded. Finally, the integration of these requirements must be understandable to those in both communities. Throughout this dissertation we strive to strike a balance between engineering and formal evaluation; between performance, security, and clarity. Specifically, in this chapter, we make the following contributions:

- We specify a memory access policy language, based on formal regular languages, for expressing the set of legal accesses and allowed policy transitions for stateful policies.
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

- We demonstrate how our language can express classical security scenarios, such as isolation, controlled sharing, and Chinese wall.

- We present a policy compiler that translates an access policy described in this language into a synthesizable hardware module.

- We evaluate the effectiveness and efficiency of this novel enforcement mechanism by synthesizing several policies down to a modern FPGA and analyzing the area and performance.

- We provide a motivating example of a reconfigurable system from the field of computer vision.

The remainder of the chapter is organized as follows: Section 3.2 describes our approach to providing memory protection. In Section 3.3, we explain the algorithms behind our reference monitor design flow. In Section 3.4, we describe our access policy language including several example policies. We present our reference monitor synthesis results in Section 3.5. Finally, we conclude in Section 3.6 and discuss where there is room for future work.
3.2 Protecting Memory on an FPGA

A Multilevel-secure run-time management system must protect different logical modules from interfering, intercepting, or corrupting any use of a shared resource. On an embedded system, the primary resource of concern is memory. Whether it is on-chip block RAM, off-chip DRAM, or backing-store such as Flash, a serious issue in the design of any high performance secure system is the allocation and reallocation of memory in a way that is efficient, flexible, and protected. On a SP processor, security domains may be enforced through the use of a page table and associated TLB. Superpages, which are very large memory pages, can also be used to provide memory protection, and their large size makes it possible for the TLB to have a lower miss rate [70]. Segmented Memory [78] and Mondrian Memory Protection [100], a finer-grained scheme, address the inefficiency of providing memory protection at the granularity of a page (or a superpage) by allowing different protection domains to have different permissions on the same memory region.

While a TLB may be used to speed up page table accesses, this requires additional associative memory (not available on FPGAs) and greatly decreases the performance of the system in the worst case. Therefore, few embedded processors and even fewer reconfigurable devices support even this most basic method of
protection. Instead, reconfigurable architectures on the market today support a simple linear addressing of the physical memory. **Hence, on a modern FPGA the memory is essentially flat and unprotected.**

Preventing unauthorized accesses to memory is fundamental to both effective debugging, error prevention, and computer security. Even if the system is not under attack, many of the most insidious bugs are a result of errant memory accesses which affect multiple sub-systems. Ensuring protection and separation of memory when multiple concurrent logic modules are active requires a new mechanism to ensure that the security properties of the system are enforced.

To provide separation in memory between multiple different interacting modules, we adapt some of the key concepts from separation kernels. Rushby originally proposed that a separation kernel \[40\] \[56\] \[75\] creates within a single shared machine an environment which supports the various components of the system, and it provides the communication channels between them in such a way that individual components of the system cannot distinguish this shared environment from a physically distributed one. A separation kernel partitions all resources under its control into blocks such that the actions of a subject in one block are isolated from (viz., cannot be detected by or communicated to) a subject in another block, unless an explicit means for that communication has been established. For a mul-
tilelevel secure system, each block typically represents a different classification level, and the allowed communications conform to the MLS-label lattice [21].

We propose to treat the separate cores of the FPGA and related memory regions as blocks of a separation kernel. By building a specialized circuit that recognizes a language of legal accesses between blocks, and then by realizing that circuit directly onto the reconfigurable device as a specialized state machine through which all off-chip memory accesses are routed, every memory access can be checked with only a small additional latency. Although implementing the enforcement module into a separate off-chip hardware module would lessen the impact of covert channel attacks between modules on the chip, this would introduce additional latency. We describe techniques to isolate the enforcement module in [37].

3.3 Policy Description and Synthesis

While reconfigurable systems typically do not have traditional memory protection enforcement mechanisms, the programmable nature of the devices means that we can build whatever mechanisms we need as long as they can be implemented efficiently. In fact, we exploit the fine grain re-programmability of FPGAs to provide word-level stateful memory protection by implementing a compiler that can
translate a memory access policy directly into a circuit. The enforcement mechanism generated by our compiler will help prevent a corrupted module or processor from compromising other modules on the FPGA with which it shares memory.

We have developed a security primitive for providing isolation of cores at the gate level by surrounding each core with a “moat” that blocks wiring connectivity from the outside [37].

We begin with an explanation of our memory access policies, and we describe how a policy can be expressed and then compiled down to a synthesizable module. In this section we explain both the high level policy description and the automated sequence of steps, or design flow, for converting a memory access policy into a hardware enforcement module. Assurance that the conversion is accurate and complete is discussed as future work.

### 3.3.1 Memory Access Policy

Once a high level policy is developed based on the requirements of the system and the organizational security policy [89], it must be expressed in a precise form to allow engineers to build concrete enforcement mechanisms. In the context of this chapter we concentrate on policies as they relate to memory accesses. In particular, the enforcement mechanisms we consider in this chapter belong to the Execution Monitoring (EM) class [83], which monitor the execution of a
target, which in our case is one or more modules on the FPGA. The enforcement mechanism is also a Reference Validation Mechanism (RVM) [3], which must be tamper-proof, always invoked, and small enough to be subject to analysis and test, the completeness of which can be assured. We describe techniques for isolating the reference monitor in [37].

Although Erlingsson et al. have proposed the idea of merging the reference monitor in-line with the target system [22], in a system with multiple interacting cores, this approach has the drawback that the reference monitors are distributed, which is problematic for stateful policies. It may also prohibit the use of third-party bit-streams or require access to source code and the re-compilation of third-party bit-streams. Although there exist security policies that execution monitors are incapable of enforcing, such as information flow policies [76], we argue that in the future our execution monitors could be combined with static analysis techniques to enforce a more broad range of policies if required. We therefore begin by describing a well defined method for describing memory access policies.

The goal of our memory access policy description is to precisely describe the set of legal memory access patterns, specifically those that can be recognized by an execution monitor capable of tracking address ranges of arbitrary size within an enforcement framework that prohibits all other access. Furthermore, it should be possible to describe complex behaviors such as sharing, exclusivity, and atomicity,
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

in an understandable fashion. An engineer can then write a policy description in our input form (as a series of “re-writing” productions) and have it transformed automatically to an extended type of regular expression. By extending regular languages to fit our needs we can have a human-readable input format, and we can build off of theoretical contributions which have created a refinement path to state machines and hardware [1].

There are three pieces of information that we will incorporate into our execution monitor. The Accessing Modules ($M$) are the unique identifiers for a specific principal on the chip, such as a specific intellectual property core or one of the on-chip processors. Throughout this chapter we simply refer to these distinct units of activity on the FPGA as “Modules.” The Access Methods ($A$) are typically Read and Write, but may include special memory operators such as execution, zeroing or incrementing if required. Elements of $A$ are used to describe “permissions.” The set $P$ is a partitioning of physical memory into “ranges.” The Memory Range Specifier ($R$) describes a set of contiguous physical addresses to which a specific permission can be assigned. Our language describes an access policy through a sequence of productions, which specify the relationship between principals ( $M$: modules ), access rights ( $A$: read, write, etc.), and objects ( $R$: memory ranges1).

---

1 An interval of the address space including high ($R_{\text{high}}$) and low ($R_{\text{low}}$) bounds
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

The terminals of the language are memory accesses descriptors which ascribe a specific right for a specific module to access a specific object until the descriptor is negated or deleted \(^2\). Formally, the terminals of the productions are tuples of the form \((M, A, R)\), and the universe of tuples forms a power set \(\Sigma = M \times A \times R\). Given two sets of tuples, \(a\) and \(b\), “\(ab\)” indicates the union of \(a\) and \(b\). A memory access policy is precisely defined as a formal language \(L \subseteq \Sigma\) which can be either generalized as being infinite or focused on a fixed number of modules, ranges, and accesses. \(L\) needs to satisfy the property that \(\forall x, t : \text{tuple set} \mid t \subseteq \Sigma, xt \subseteq L \rightarrow x \subseteq L\), so that any legal access sequence will be incrementally recognized as legal along the way.

One thing to note is that memory accesses refer to a specific memory address, while memory access descriptors are defined over the set of all memory ranges \(R\) (i.e., the power set of addresses). A memory access \((M, A, k)\), where \(k\) is a particular address, is contained in a memory access descriptor \((M', A', R)\) iff \(M = M', A = A',\) and \(R_{low} \leq k \leq R_{high}\). A sequence of memory accesses \(a = a_0, a_1, ..., a_n\) is said to be legal iff \(\forall 0 \leq i \leq n \exists s_i \in L \mid a_i \in s_i\). In order to enforce this policy during the execution of an FPGA, we need three things.

1. A notation with the details for a specific policy can be precisely defined under \(L\). We describe \(L\) later in this section.

\(^2\)Details of revocation will be discussed in Section 3.4
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

2. A method for automatically creating a circuit which recognizes memory access sequences that are legal under $L$. We describe this method in Section 3.3.2.

3. A method for preventing all accesses that are not legal under $L$. We describe our enforcement architecture in Section 3.5.1.

We begin with a description of (1) through the use of a simple example. Consider a straightforward isolation policy that simply enforces the separation in memory of two different modules. $Module_1$ is only allowed to access memory in the range of $[0x8e7b008,0x8e7b00f]$, and $Module_2$ is only allowed to access memory in the range of $[0x8e7b018,0x8e7b01b]$. In our memory access policy definition format, this is coded as the following set of productions:

$$
\begin{align*}
    rw & \rightarrow r \mid w; \\
    Range_1 & \rightarrow [0x8e7b008,0x8e7b00f]; \\
    Range_2 & \rightarrow [0x8e7b018,0x8e7b01b]; \\
    Access_1 & \rightarrow \{Module_1, rw, Range_1\}; \\
    Access_2 & \rightarrow \{Module_2, rw, Range_2\}; \\
    Policy & \rightarrow (Access_1 | Access_2)^*;
\end{align*}
$$

Each of these productions is a re-writing rule as in a standard grammar. The non-terminal $Policy$ is the start symbol of the grammar that defines the overall access policy ($L$ as described above). Through the use of a grammar we allow the hierarchical composition of more complex policies. In this case $Access_1$ and $Access_2$ are simple access descriptors, but we want to allow more complex sets
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

of memory accesses, such that all sequences of accesses that can be derived from Policy by application of the grammar’s productions are legal.

Since we eventually want to transform the access policy to hardware logic in a limited space, we limit our language to sequences that can be described with grammatical constructs no more complex than a regular expression [59], with the added ability to express ranges. Although a regular language is limited to a type-3 regular grammar in the Chomsky hierarchy, it is inconvenient for security administrators to express policies in right-linear or left-linear form, which would not allow “range” expressions. Since a language can be recognized by many grammars, any grammar that can be automatically transformed into type-3 form is acceptable, so we present the end user with an extended regular grammar that is later transformed by extracting first terminals from non-terminals.

Note that the atomic unit of enforcement is an address range, and that the ranges are of arbitrary granularity. The smallest granularity that we currently allow in the policy definition format is at the word boundary, and we can support any sized range from a single word to the entire address space. Also, ranges may be of the same or different size, unlike traditional memory pages. We will later show how this ability can be used to set up special control words that help in securely coordinating between modules.
Although we are restricted to policies that are equivalent to a finite automata with range checking, we have constructed many example policies including isolation and Chinese wall in order to demonstrate the versatility and efficiency of our approach. In Section 3.4.4 we describe a “redaction policy,” in which modules with multiple security clearance levels are interacting within a single embedded system. However, now that we have introduced our memory access policy definition format, we describe how it can be transformed automatically to an efficient circuit for implementation on an FPGA.

3.3.2 Hardware Synthesis

We have developed a policy compiler that converts an access policy, as described above, into a circuit that can be loaded onto an FPGA to serve as the policy enforcement module. At a high level the technique partitions the module into two parts, range discovery and language recognition. Specifically the steps of our design flow are:

1. User creates the access policy (described above) and inputs it to the compiler, which:

   2. Builds a syntax tree from the policy.

   3. Transforms the syntax tree to an expanded intermediate form.
4. Expands Policy to a regular expression defined over the alphabet Σ.

5. Converts the regular expression to a non-deterministic finite automaton (NFA).

6. Constructs an equivalent minimized state machine from the NFA.

7. Factors the ranges into sizes that are a power of two.

8. Organizes the set of ranges as a trie\(^3\), and creates a logic tree that recognizes them.

9. Exports the state machine and range detection logic as Synthesizable Verilog.

10. Inputs hardware description expressed in Verilog to Quartus software, which synthesizes, places, and routes circuit.

11. Bit-stream loader loads the synthesized bit-stream onto the FPGA.

### 3.3.3 Design Flow Details

**Access Policy** — To describe the process of transforming a policy to a circuit, we again consider a simple isolation policy with two modules, which can only access their own single range:

\(^3\)an ordered tree data structure for storing lookup tables
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

\[ \text{Access} \rightarrow \{ \text{Module}_1, \text{rw}, \text{Range}_1 \} \mid \{ \text{Module}_2, \text{rw}, \text{Range}_2 \}; \]
\[ \text{Policy} \rightarrow (\text{Access})^*; \]

Building and Transforming a Parse Tree – Next, we use Lex [55] and Yacc [43] to build a parse tree from our security policy. Internal nodes represent operators such as concatenation, alternation, and repetition. Figure 3.1 shows the parse tree for our example policy.

![Parse tree of the simple access policy](image)

Figure 3.1: Parse tree of the simple access policy

We must then transform the parse tree into a large single production with no non-terminals on the right hand side, from which we can generate a regular expression. This process of macro expansion requires an iterative replacement of all the non-terminals in the policy. We apply the productions to the parse tree by substituting the left hand side of each production with its right hand side. Figure 3.2 shows the transformed parse tree for our policy.
Building the Regular Expression — Next, we find the subtree corresponding to *Policy* and traverse this subtree to obtain the regular expression. By this stage we have completely eliminated all of the RHS non-terminals, and we are left with a single regular expression which can then be converted to an NFA. The regular expression for our access policy is:

\[
((\{\text{Module}_1, \text{rw}, \text{Range}_1\}) \mid (\{\text{Module}_2, \text{rw}, \text{Range}_2\}))^*
\]

Constructing the NFA — Once the regular expression has been formed, an NFA can be constructed from this regular expression using Thompson’s Algorithm [1] as implemented by Gerzic [25]. Figure 3.3 shows the NFA for our policy. Notice that the policy transitions can occur in parallel. We will use the FPGA to exploit this for faster processing.
Figure 3.3: NFA derived from the regular expression

Converting the NFA to a DFA — From this NFA we can construct a DFA through subset construction [1] as implemented by Gerzic [25]. Following the creation of the DFA, we apply Hopcroft’s Partitioning Algorithm [1] as implemented by Grail [74] to minimize the DFA. Figure 3.4 shows the minimized DFA for our policy.

Figure 3.4: NFA converted to a minimized DFA
Processing the Ranges – Before we can convert the DFA into Verilog, we must perform some processing on the ranges so that the circuit can efficiently determine which range contains a given address. Our system converts the ranges to an internal format using “don’t care” bits. For example, 10XX can be 1000, 1001, 1010, or 1011, which is the range [8,11]. Hardware can be easily synthesized to check if an address is within a particular range by performing a bit-wise XOR on just the significant bits. Using this optimization, any aligned power of two range (i.e., the cardinality of the range is a power of two) can be efficiently described, and any non-power of two range can be converted into a covering set of $O(\log_2 |\text{range}|)$ power of two ranges. For example the range [7,12] (0111, 1000, 1001, 1010, 1011, 1100) is not an aligned power of two range but can be converted to a set of aligned power of two ranges: $\{[7,7],[8,11],[12,12]\}$ (or equivalently $\{0111|10XX|1100\}$).

Converting the DFA to Verilog – Because state machines are a very common hardware primitive, there are well-established methods of translating a description of state transitions into a hardware description language such as Verilog. Figure 3.5 shows the hardware decision module we wish to build.

As previously described, an access descriptor specifies the allowed accesses between a module and a range. Each DFA transition represents an access descriptor,
Figure 3.5: The inputs to the enforcement module are the module ID, op, and address. The range ID is determined by performing a parallel search over all ranges, similar to a content addressable memory (CAM). The module ID, op, and range ID together form an access descriptor, which is the input to the state machine logic. The output is a single bit: either grant or deny the access.

A memory access request comprises three inputs: the module ID, the op \{read, write, etc.\}, and the address. The output is a single bit: 1 for grant and 0 for deny. First, the hardware converts the memory access address to a bit vector. To do this, it checks all the ranges in parallel and sets the bit corresponding to the range ID that contains the input address (if any).
Then the memory access request is processed through the DFA. If an access descriptor matches the access request, the DFA transitions to the accept state and outputs a 1. If there is no transition for an access request, the machine always transitions to the rejecting state, which is a “dummy” sink state. This is important for security because an attacker might try to access an address not covered by the policy or try to insert illegal characters into the input, and results in a “fail secure” machine.

**State Machine Synthesis** – The final step in the design flow is the actual conversion of Verilog code to a bit-stream that can be loaded onto an FPGA. Using the Quartus tools from Altera, which does synthesis, optimization, and place-and-route, we turn each machine into an actual implementation. After testing the circuit to verify that it accepts a sample of valid accesses and rejects invalid accesses, we are ready to measure the area and cycle time of our design.

### 3.4 Example Applications

To further demonstrate the utility of our language, we use it to express several different policies. We have already demonstrated an isolation policy, which can be easily extended to include overlapping ranges, shared regions, and most any static policy. The true power of our system comes from the description of *stateful*
policies that involve revocation or conditional access or other forms of dynamic policy. Let us first discuss a traditional example: access control lists.

### 3.4.1 Access Control List

A secure system that employs access control lists will associate every object in the system with a list of principals along with the rights of each principal to access the object. For example, suppose our system has two objects, $Range_1$ and $Range_2$. $Class_1$ is a class of principals ($Module_1$ and $Module_2$), and $Class_2$ is another class of principals ($Module_3$ and $Module_4$). Either $Class_1$ or $Class_2$ may access $Range_1$, but only $Class_2$ may access $Range_2$. We express such an access control list policy below:

\[
\begin{align*}
Class_1 & \rightarrow Module_1 | Module_2; \\
Class_2 & \rightarrow Module_3 | Module_4; \\
List_1 & \rightarrow Class_1 | Class_2; \\
List_2 & \rightarrow Class_2; \\
Access_1 & \rightarrow \{List_1, rw, Range_1\}; \\
Access_2 & \rightarrow \{List_2, rw, Range_2\}; \\
Policy & \rightarrow (Access_1 | Access_2)^*;
\end{align*}
\]

In general, since access control list policies are stateless, the resulting DFA will have one state, and the number of transitions will be the sum of the number of principals that may access each object. In this example, $Module_1$, $Module_2$, $Module_3$, and $Module_4$ may access $Range_1$, and $Module_3$ and $Module_4$ may access $Range_2$. The total number of transitions in this example is $4+2=6$. 
3.4.2 Controlled Sharing

Secure system design requires the prevention of unintended flows of information between principals such as cores, but there are times when cores need to communicate with each other. Our language makes possible the secure transfer of data from one core to another. Rather than requiring large communication buffers or multiple copies of the data, we can simply transfer the control of a specified range of data from one module to the next. For example, suppose Module\textsubscript{1} wants to securely transfer some data to Module\textsubscript{2}. Rather than establishing a direct channel between Module\textsubscript{1} and Module\textsubscript{2}, an access policy can be created that synchronizes the transition of permissions during the exchange. Using formal languages to express security policies makes such an exchange possible. Consider the example below:

\begin{align*}
\text{Module}_{1|2} & \rightarrow \text{Module}_{1} | \text{Module}_{2}; \\
\text{Access}_{1} & \rightarrow \{\text{Module}_{1}, \text{rw}, \text{Range}_{1}\} \mid \{\text{Module}_{1|2}, \text{rw}, \text{Range}_{2}\}; \\
\text{Access}_{2} & \rightarrow \{\text{Module}_{2}, \text{rw}, (\text{Range}_{1} \mid \text{Range}_{2})\}; \\
\text{Trigger} & \rightarrow \{\text{Module}_{1}, \text{rw}, \text{Range}_{2}\}; \\
\text{Policy} & \rightarrow (\text{Access}_{1})^* (\epsilon \mid \text{Trigger} \ (\text{Access}_{2})^*);
\end{align*}

Initially, Module\textsubscript{1} can access Range\textsubscript{1} and Range\textsubscript{2}, and Module\textsubscript{2} can only access Range\textsubscript{2}. However, the first time Module\textsubscript{1} accesses Range\textsubscript{2} (signaling to Module\textsubscript{2} that Module\textsubscript{1} is ready to exchange), Access\textsubscript{1} is deactivated by this
trigger event, revoking the permissions for Module_1 from both Ranges. As a result of the trigger, Module_2 has exclusive access to Range_1 and Range_2.

### 3.4.3 Chinese Wall

Another security scenario that can be efficiently expressed using our policy language is the Chinese wall [13]. Consider an example of this scenario, in which a lawyer who looks at the set of documents of Company_1 should not view the set of files of Company_2 if Company_1 and Company_2 are in the same conflict-of-interest (COI) class. This lawyer may also view the files of Company_3 provided that Company_3 belongs to a different COI class than Company_1. Figure 3.6 shows a Venn Diagram for this situation.

![Venn Diagram for Chinese Wall](image)

**Figure 3.6:** A Chinese wall policy. This Venn Diagram shows two conflict-of-interest classes, $Class_A$ and $Class_B$.

We express a Chinese wall policy below, where Module_1 corresponds to the lawyer and each range corresponds to a company:
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

\[
Access_1 \rightarrow \{\text{Module}_1, \text{rw}, (\text{Range}_1 \mid \text{Range}_3)\}*;
\]

\[
Access_2 \rightarrow \{\text{Module}_1, \text{rw}, (\text{Range}_1 \mid \text{Range}_4)\}*;
\]

\[
Access_3 \rightarrow \{\text{Module}_1, \text{rw}, (\text{Range}_2 \mid \text{Range}_3)\}*;
\]

\[
Access_4 \rightarrow \{\text{Module}_1, \text{rw}, (\text{Range}_2 \mid \text{Range}_4)\}*;
\]

\[
Policy \rightarrow Access_1 \mid Access_2 \mid Access_3 \mid Access_4;
\]

In our Chinese wall policy, there are two COI classes. One contains \text{Range}_1 and \text{Range}_2, and the other contains \text{Range}_3 and \text{Range}_4. For simplicity, we have restricted this policy to one module since with multiple modules, the restrictions to a module are independent of the actions of other modules so each module requires its own state machine. Figure 3.7 shows the DFA that recognizes legal accesses for this policy.

Figure 3.7: This DFA recognizes legal accesses for this Chinese Wall policy. A principal that accesses \text{Range}_4 (black) is subsequently prohibited from accessing \text{Range}_3 (dark gray), but it may access either \text{Range}_1 (white) or \text{Range}_2 (light gray), because they are in a different class. An access to \text{Range}_4 results in a transition to state 2 (black), from which an access to \text{Range}_1 results in a transition to state 1 (black or white).
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

In general, for Chinese wall security policies, the number of states scales exponentially to the number of COI classes. Because the number of possible legal accesses is the serial product of the number of ranges (companies) in each separate COI class. The number of transitions also scales exponentially to the number of COI classes for the same reason. Fortunately, the number of states and the number of transitions both scale linearly to the number of ranges. In addition, the number of transitions scales linearly in the number of ranges.

3.4.4 Redaction

Our security language can also be used to enforce forms of redaction [81], even at very high throughput (such as for video). Military hardware such as avionics [98] may contain processing components that are “cleared” for different levels of data, and a TS component must not leak sensitive information to a U component [87]. However, the TS component may be required to send a document to the U component; a third component does this by redacting TS data from the document. Figure 3.8 shows the architecture of a redaction scenario that is based on separation.

A multilevel database contains both top secret (TS) and unclassified (U) data. Module_1 has a TS label, and Module_2 has a U label. Module_1 and Module_2 are initially isolated, since they have different labels. Therefore, Range_1 belongs to
Figure 3.8: A redaction architecture. A database contains both Top Secret and Unclassified data. Module\(_1\) has a Top Secret (TS) clearance, and Module\(_2\) has an Unclassified (U) clearance. Any database query requested by Module\(_2\) must have all TS data redacted by the Trusted Server Module\(_3\). Furthermore, Module\(_2\) must be prevented from accessing the result of a database query performed by Module\(_1\) because such a query result may contain TS data. This is accomplished by revoking Module\(_2\)'s permission to access the temporary storage (Range\(_3\)) where query results are written by the Trusted Server. IP stands for Intellectual Property.

\(\text{Module}_1\), and Range\(_2\) belongs to \(\text{Module}_2\). Module\(_3\) acts as a trusted server of information contained in the database, and this server must have a security label range from U to TS. Range\(_3\) is temporary storage used for holding information that has just been retrieved from the database by the trusted server. Range\(_4\) (the control word) is used for performing database queries: a module writes to Range\(_4\) to request that Module\(_3\) retrieve some information from the database and then write the query result to the temporary storage. Any database query requested by Module\(_2\) must have all TS data redacted by the trusted server. If a request is made by Module\(_1\) for top secret information, it is necessary to revoke Module\(_2\)'s
read access to the temporary storage, and this access must not be reinstated until the trusted server zeroes out the sensitive information contained in the temporary storage. One way of implementing the zeroing out functionality is to use a special access right \( z \) in conjunction with logic that erases the contents of the temporary storage. We express our redaction policy below:

\[
\begin{align*}
\text{rw} & \rightarrow r \mid w; \\
\text{Access}_2 & \rightarrow \{\text{Module}_1, \text{rw}, \text{Range}_1\} \mid \{\text{Module}_1, r, \text{Range}_3\} \\
& \quad \mid \{\text{Module}_2, \text{rw}, \text{Range}_2\} \mid \{\text{Module}_2, w, \text{Range}_4\} \mid \{\text{Module}_3, w, \text{Range}_3\}; \\
\text{Access}_1 & \rightarrow \{\text{Module}_2, r, \text{Range}_3\} \mid \text{Access}_2; \\
\text{Trigger} & \rightarrow \{\text{Module}_1, w, \text{Range}_4\}; \\
\text{Clear} & \rightarrow \{\text{Module}_3, z, \text{Range}_3\}; \\
\text{SteadyState} & \rightarrow (\text{Access}_2 \mid \text{Clear Access}_1^* \text{Trigger})^*; \\
\text{Policy} & \rightarrow \varepsilon \mid \text{Access}_1^* \mid \text{Access}_1^* \text{Trigger SteadyState} \\
& \quad \mid \text{Access}_1^* \text{Trigger SteadyState Clear Access}_1^*;
\end{align*}
\]

\text{Access}_1 is the less restrictive access mode, and \text{Access}_2 is the more restrictive access mode. The Trigger event changes the access mode from \text{Access}_1 to \text{Access}_2, and the Clear event causes the machine to transition from \text{Access}_2 back to \text{Access}_1. In general, the DFA for a redaction policy will have one state for each access mode. Applying our redaction policy to a real-world video privacy system would likely require some additional complexity.

### 3.4.5 Bell and LaPadula Confidentiality Model

The Bell and LaPadula (B&L) Model is a formal model of multilevel security in which a subject may not read an object with a higher security label (no read-
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

up), and a subject may not write to an object with a lower security label (no write-down) [6]. This model is designed to protect the confidentiality of classified information. All B&L policies are stateless in that the rules don’t change and the labels of individual subjects and objects upon which the rules are based, don’t change. We express a B&L policy below:

\[
\text{Access}_{\text{B&L}} \rightarrow \{\text{Module}_1, r, \text{Range}_1\} \mid \{\text{Module}_1, r, \text{Range}_2\} \mid \{\text{Module}_2, r, \text{Range}_2\} \mid \{\text{Module}_2, w, \text{Range}_1\} \mid \{\text{Module}_2, w, \text{Range}_2\};
\]

\[
\text{Policy} \rightarrow (\text{Access}_{\text{B&L}})^*;
\]

In our simple example, Module\(_1\) has a TS label, Module\(_2\) has a U label, Range\(_1\) has a S label, and Range\(_2\) has a U label. We leave to future work the covert channel analysis of these mechanisms.

3.4.6 High Water Mark

High water mark is similar to B&L in that no read-up is permitted, but object labels change over time, and write-down is allowed. Following a write-down, the security label of the object written to must change to the label of the subject that performed the write; thus, high water mark policies are stateful. We express our high water mark policy below:

\[
\text{Access}_1 \rightarrow \{\text{Module}_1, r, \text{Range}_1\} \mid \{\text{Module}_1, r, \text{Range}_2\} \mid \{\text{Module}_1, w, \text{Range}_2\} \mid \{\text{Module}_2, w, \text{Range}_1\};
\]

\[
\text{Access}_2 \rightarrow \text{Access}_{\text{B&L}} \mid \{\text{Module}_1, w, \text{Range}_1\};
\]

\[
\text{Access}_3 \rightarrow \text{Access}_1 \mid \{\text{Module}_2, w, \text{Range}_2\};
\]

\[
\text{Access}_4 \rightarrow \text{Access}_1 \mid \{\text{Module}_1, w, \text{Range}_1\} \mid \{\text{Module}_2, w, \text{Range}_2\};
\]
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

\( Trigger_1 \rightarrow \{ \text{Module}_1, w, \text{Range}_1 \} \);
\( Trigger_2 \rightarrow \{ \text{Module}_1, w, \text{Range}_2 \} \);
\( Path_1 \rightarrow (\epsilon | Trigger_1 Access_2^* (\epsilon | Trigger_2 Access_4^*)) \);
\( Path_2 \rightarrow (\epsilon | Trigger_2 Access_3^* (\epsilon | Trigger_1 Access_4^*)) \);
\( Policy \rightarrow Access_{B&L}^* | (\epsilon | Path_1 | Path_2) \);

We use trigger events to express the write-downs. The number of triggers \( T \) in the high water mark policy is equal to the number of write-downs that would be illegal in the B&L policy. The number of states \( S \) in the DFA that enforces the high water mark policy is \( O(2^T) \), and the number of transitions in the DFA that are triggers is \( O(T!T) \). If \( N \) is the number of transitions in the corresponding stateless B&L policy, then the number of transitions in the high water mark DFA that are not triggers is \( O((N)(S)) \). Therefore, the total number of transitions is \( O(T!T + (N)(S)) \).

3.4.7 Biba Integrity Model

The Biba model is the dual of the Bell-LaPadula model [8], but the label spaces of the policies are distinct. Both read-down and write-up with respect to the ordering of integrity labels are prohibited. Like B&L, all Biba policies are stateless. We express our B&L policy below:

\( Access_{\text{Biba}} \rightarrow \{ \text{Module}_1, w, \text{Range}_1 \} | \{ \text{Module}_1, w, \text{Range}_2 \} | \{ \text{Module}_2, r, \text{Range}_1 \} \)
\( | \{ \text{Module}_2, r, \text{Range}_2 \} | \{ \text{Module}_2, w, \text{Range}_2 \} ; \)
\( Policy \rightarrow (Access_{\text{Biba}})^* ; \)
Low water mark is to Biba as high water mark is to B&L. Since low water mark is similar to high water mark, we do not discuss it further.

3.4.8 Dynamic Policies

The ability to change the policies in response to external events is useful. For example, if the system comes under attack, it may be necessary to change to a more restrictive policy. We express a dynamic policy below:

\[
\text{Policy} \rightarrow \text{Policy}_1 (\epsilon | \text{Trigger}_1 (\text{Policy}_2) (\epsilon | \text{Trigger}_2 (\text{Policy}_3)));
\]

\text{Policy}_1, \text{Policy}_2, \text{and Policy}_3 can be any three policies. If the policies come from different sources, pre-processing can be used to prevent naming conflicts (e.g., if two policies define \text{Access}_1 differently). Trigger events specify the circumstances under which a policy change can occur. \text{Trigger}_1 causes the policy to change from \text{Policy}_1 to \text{Policy}_2, and \text{Trigger}_2 causes the policy to change from \text{Policy}_2 to \text{Policy}_3. Every state in \text{Policy}_1 has an additional transition (\text{Trigger}_1) to the first state of \text{Policy}_2, and every state in \text{Policy}_2 has an additional transition (\text{Trigger}_2) to the first state of \text{Policy}_3. The number of states in the combined policy is \(O((S_1) + (S_2) + (S_3))\), where \(S_N\) is the number of states in \text{Policy}_N. The number of transitions in the combined policy is \(O((T_1) + (T_2) + (T_3))\), where \(T_N\) is the number of transitions in \text{Policy}_N.
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

In the above scenario, the system must start in Policy_1. The system may or may not transition to Policy_2. If the system transitions to Policy_2, the system may or may not transition to Policy_3. Supporting the ability to go in any order requires more complex expressions and more complex DFAs. In addition, the ability to return to an earlier policy has several security implications, especially when stateful policies are involved. Understanding the organizational requirements for dynamic security policies is the topic of related research [12] [24].

Although switching back and forth between an arbitrary number of stateful policies would require modifying our compiler, it is possible to use our language to switch back and forth between two stateless policies Policy_1 and Policy_2 using the following expression:

\[
\text{SteadyState} \rightarrow (\text{Policy}_2 | \text{Trigger}_2 \text{Policy}_1 \text{Trigger}_1)^*; \\
\text{Policy} \rightarrow \text{Policy}_1 | \text{Policy}_1 \text{Trigger}_1 \text{SteadyState} \\
| \quad \text{Policy}_1 \text{Trigger}_1 \text{SteadyState Trigger}_2 \text{Policy}_1 | \epsilon;
\]

\text{Trigger}_1 \text{ changes the policy from Policy}_1 \text{ to Policy}_2, \text{ and Trigger}_2 \text{ changes the policy back to Policy}_1.
3.5 Integration and Evaluation

Now that we have described several different memory access policies that could be enforced using a stateful monitor, we need to demonstrate that such systems could be efficiently realized on reconfigurable hardware.

3.5.1 Enforcement Architecture

The placement of the enforcement mechanism can have a significant impact on the performance of the memory system. Figure 3.9 shows two architectures for the enforcement mechanism which assumes that modules on the FPGA can only access shared memory via the bus.

In the figure on the left, the enforcement mechanism sits between the memory and the bus, which means that every access must pass through the enforcement mechanism before going to memory. In the case of a read, the request cannot proceed to memory until the enforcement mechanism approves the access. This results in a large delay which is the sum of the time to determine the legality of the access and the memory latency. We can mitigate this problem by having the enforcement mechanism snoop on the bus or through the use of various caching mechanisms for keeping track of accesses that have already been approved. This scenario is shown in the figure on the right. In the case of a read, the request is sent
Figure 3.9: Two alternative architectures for the enforcement mechanism. In the figure on the left, a memory access must pass through the enforcement mechanism (E) before going to memory. In the figure on the right, the enforcement mechanism (E) snoops on the bus, and a buffer (B) prevents access to the data until the access is approved. Arbiters prevent the bus from being accessed by more than one module at a time.

to memory, and the memory access occurs in parallel with the task of determining the legality of the read. A buffer holds the data until the enforcement mechanism grants approval, at which time E sends the data across the bus. In the case of a write, the data to be written is stored in the buffer until the enforcement mechanism grants approval, at which time E sends the data from the bus to memory. Thus, both architectures provide the isolation and omnipotence required of a reference or execution monitor.

Since a module may be sending sensitive data over the bus, it is necessary to prevent other modules from accessing the bus at the same time. We address this
problem by placing an arbiter between each module and the bus. In a system with two modules, for example, the arbiters could allow one module to access the bus on even clock cycles and the other module to access the bus on odd clock cycles. We discuss a secure communication architecture for FPGAs as well as a method of ensuring the isolation of the reference monitor at the gate level in [37].

3.5.2 Evaluation

Of the different policies we discussed in Section 3.4, we focus primarily on characterizing the isolation policy in order to separate the effect of range detection on system efficiency. Rather than tying our results to the particular reconfigurable system prototype we are developing, we quantify the results of our design flow on a randomly generated set of ranges over which we enforce isolation. The range matching constitutes the majority of the hardware complexity (assuming there are a large number of ranges), and there has already been a great deal of work in the CAD community on efficient state machine synthesis [65].

To obtain data detailing the timing and resource usage of our range matching state machines, we ran the memory access policy description through our front-end and synthesized\(^5\) the results with Quartus II 4.2 [2]. Compilations are optimized
\(^5\)the back-end handles netlist creation, placement, routing, and optimization for both timing and area
for the target FPGA device (Altera Stratix EPS1S10F484C5), which has 10,570 available logic cells, and Quartus will utilize as many of these cells as possible.

3.5.3 Synthesis Results

In general, a DFA for an isolation policy always has exactly one state, and there is one transition for each \( \{\text{ModuleID, op, RangeID}\} \) tuple. We have determined that for our isolation policy, there is a linear relationship between the number of transitions and the number of ranges. Figure 3.11 shows that the area of the resulting circuit scales nearly linearly with the number of ranges for the compartmentalization policy. The slope is approximately four logic cells for every range.

\[ \text{Figure 3.10: Setup time and cycle time. Setup time is the time required to determine the range to which an address belongs. Cycle time is the time required to perform one state machine transition (one clock cycle). Although setup time is often more than one cycle, pipelining can provide better throughput.} \]
Figure 3.11: Circuit area versus number of ranges. There is a nearly linear relationship between the circuit area and the number of ranges.

Figure 3.10 explains the components that make up total time: setup time and cycle time. Figure 3.12 shows the cycle time ($T_{\text{clock}}$) for machines of various sizes. $T_{\text{clock}}$ is primarily the time for one DFA transition, and it is very close to the maximum frequency of this particular Altera Stratix device (one clock cycle).

Figure 3.12: Cycle time versus number of ranges. There is a nearly constant relationship between the cycle time and the number of ranges.

Figure 3.13 shows the setup time ($T_{\text{su}}$), which is primarily the time to determine the range to which the input address belongs.
Figure 3.13: Setup time versus number of ranges. Above 170 ranges, there is a nearly linear relationship between the setup time and the number of ranges. This time can be reduced with pipelining.

Although $T_{clock}$ remains nearly constant with the number of ranges, $T_{su}$ increases nearly linearly above 170 ranges. Fortunately, $T_{su}$ can be reduced by pipelining the circuitry that determines what range contains the input address.

In the series of isolation circuits from the first experiment above, we varied the number of ranges from 69 to 652, but they all had the same policy. In our second experiment, we only used a handful of ranges, but we varied the policy. Figure 3.14 shows the area of the circuits resulting from the example policies presented in this chapter. Since we only used a handful of ranges, the circuits from the second experiment are much smaller in area than the circuits from the first experiment.

The complexity of the circuit is a combination of the number of ranges and the number of DFA states and transitions. In our dynamic policy, $Policy_1$ is our isolation policy, $Policy_2$ is our Biba policy, and $Policy_3$ is our controlled sharing policy. Returning to an earlier policy is not allowed since $Policy_3$ is stateful.
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

![Number of Logic Cells vs. Policy](image1)

**Figure 3.14:** Circuit area versus access policy. The area is related to the number of states, transitions, and ranges. The circuit area is greatest for the dynamic policy.

As expected, the circuit for the dynamic policy has the greatest area because it consists of three policies. The next biggest circuit belongs to Chinese wall, followed by redaction, high water mark, low water mark, and controlled sharing.

Figure 3.15 shows that the cycle time is relatively stable across different policies, remaining between 6.2 and 7.1 ns (one clock cycle).

![Cycle Time vs. Policy](image2)

**Figure 3.15:** Cycle time for each access policy. Cycle time is relatively stable across different policies, remaining between 6.2 and 7.1 ns (one clock cycle).
Figure 3.16 shows that the setup time remains very stable at slightly above one clock cycle. This differs significantly from the first experiment, which had much larger setup times due to the large number of ranges (on the order of hundreds of ranges in the first experiment compared with just a handful of ranges in the second experiment).

**Figure 3.16:** Setup time for each access policy. Setup time is greatest for Biba, followed by B&L and dynamic.

### 3.5.4 Impact of the Reference Monitor on System Performance

Since FPGAs do not operate at an extremely high frequency, they achieve their performance from spatial parallelism. FPGA applications such as DSPs, signal processing, and intrusion detection systems are throughput-driven and therefore are latency-insensitive. These applications are designed using careful scheduling
Chapter 3. Policy-Driven Memory Protection for Reconfigurable Systems

and pipelining techniques. For these reasons, we argue that our technique will not impact the performance significantly. For example, since an FPGA operating at 200MHz will have a cycle time of 5ns (similar to the device used in our evaluation), hence our reference monitor only adds at most a two cycle delay in this case.

3.6 Summary

Reconfigurable systems are blurring the line between hardware and software, and they represent a large and growing market. Due to the increased use of reconfigurable logic in mission-critical applications, a new set of security primitives is needed to prevent improper memory sharing and to contain memory bugs in these physically addressed embedded systems. We have demonstrated a method and language for specifying access policies that can be used as both a description of legal access patterns and as an input specification for direct synthesis to a reconfigurable logic module. Our architecture ensures that the policy module is invoked for every memory access.

Our formal access policy language provides a convenient and precise way to describe the fine-grained memory separation of modules on an FPGA. We have used our policy compiler to translate a variety of security policies to hardware enforcement modules, and we have analyzed the area and performance of these
circuits. Our synthesis data show that our methods are both efficient and scalable in the number of ranges that must be recognized. In addition to the reconfigurable domain, our methods can be applied to systems-on-a-chip as part of a more general scheme. In Chapter 4, we will show how our architecture ensures that the reference monitor is always invoked, tamperproof, and non-bypassable.

Since expressing some policies in our language requires complex expressions, we do not expect a human engineer to work in our language. Because usability is fundamental to system security [41] [30], in Chapter 6 we present a higher-level language along with a set of tools to assist the engineer in constructing mathematically precise policies. This work builds on the policy engineering work of Fong et al. [23]. A higher-level language allows the engineer to express policies in terms of security concepts (e.g., isolation, controlled sharing, etc.) rather than in terms of modules and ranges.

We believe that sound reconfigurable system security requires both on-line checks by an execution monitor as well as static techniques. Static techniques and runtime checks complement each other, with each approach lending its advantages. Static analysis suffers from the problem of false positives, and some policies depend on runtime information [95]. Furthermore, the partial reconfiguration feature that allows some of the latest FPGAs to dynamically swap cores in and out makes static analysis more challenging.
On the other hand, execution monitors consume area and involve runtime overhead. An attacker can target the reference monitor or try to bypass it. Enforcing multiple parallel memory accesses requires replicating the reference monitor, which is more difficult for stateful policies. The reference monitor is only as good as the policy it enforces, and an improperly formed policy could allow an attacker to use the grant/deny decision of the reference monitor as a covert channel. We address this problem in Chapter 5 by presenting a method of analyzing stateful policies to detect possible covert channels. If one is highly concerned about covert channels, any core that succeeds in violating the policy through the use of a covert channel must be terminated, but the termination of cores has serious consequences, such as shutting down critical services.
Chapter 4

Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

If you are looking for perfect safety, you will do well to sit on a fence and watch the birds; but if you really wish to learn, you must mount a machine and become acquainted with its tricks by actual trial.

Wilbur Wright (1867-1912)

4.1 Introduction

In this chapter, we present a static technique that exploits the spatial nature of computation on FPGAs to provide physical isolation of cores. Our architecture uses this technique to ensure that the reference monitor is always invoked, tamperproof, and non-bypassable. Consider a system with two soft-processor cores and an AES encryption engine sharing a single FPGA. Each of these three cores
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

requires access to off-chip memory to store and retrieve data. How can we ensure that the encryption key for one of the processors cannot be obtained by the other processor by either reading the key from external memory or directly from the encryption core itself? There is no virtual memory on these systems, and after being run through an optimizing CAD tool the resulting circuit is a single entangled mess of gates and wires. To prevent the key from being read directly from the encryption core itself, we must find some way to isolate the encryption engine from the other cores at the gate level. To protect the key in external memory, we need to implement a memory protection module, we need to ensure that each and every memory access goes through this monitor, and we need to ensure that all cores are communicating only through their specified interfaces. To ensure these properties hold at even the lowest levels of implementation (after all the design tools have finished their transformations), we argue that slight modifications in the design methods and tools can enable the rapid static verification of finished FPGA bitstreams\footnote{Bitstreams are the term for the detailed configuration files that encode the exact implementation of a circuit on reconfigurable hardware – in many ways they are analogous to a statically linked executable on a traditional microprocessor.}. The techniques presented in this chapter are steps towards a cohesive reconfigurable system design methodology that explicitly supports cores with varying levels of trust and criticality – all sharing a single physical device.
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

Specifically, we present the idea of Moats and Drawbridges, a statically verifiable method to provide isolation and physical interface compliance for multiple cores on a single reconfigurable chip. The key idea of the Moat is to provide logical and physical isolation by separating cores into different areas of the chip with “dead” channels between them that can be easily verified. Note that this does not require a specialized physical device; rather, this work only assumes the use of commercially available commodity parts. Given that we need to interconnect our cores at the proper interfaces (Drawbridges), we introduce interconnect tracing as a method for verifying that interfaces carrying sensitive data have not been tapped or routed improperly to other cores or I/O pads. Furthermore, we evaluate a technique, configuration scrubbing, for ensuring that remnants of a prior core do not linger following a partial reconfiguration of the system to enable object reuse. Once we have a set of drawbridges, we need to enable legal inter-core communication. We describe two secure reconfigurable communication architectures that can be easily mapped into the unused moat areas (and statically checked for isolation), and we quantify the implementation trade-offs between them in terms of complexity of analysis and performance. Finally, to demonstrate the efficacy of our techniques, we apply them to a memory protection scheme that enforces the legal sharing of off-chip memory between multiple cores.
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

4.2 Reconfigurable Systems

As mentioned in Section 4.1, a reconfigurable system is typically constructed piecemeal from a set of existing modules (called cores) in order to save both time and money; rarely does one design a full system from scratch. One prime example of a module that is used in a variety of contexts is a soft-processor. A soft-processor is simply a configuration of logical gates that implements the functionality of a processor using the reconfigurable logic of an FPGA. A soft-processor, and other intellectual property (IP) cores\(^2\) such as AES implementations and Ethernet controllers, can be assembled together to implement the desired functionality. Cores may come from design reuse, but more often than not they are purchased from third party vendors, generated automatically as the output of some design tool, or even gathered from open source repositories. While individual cores such as encryption engines may be formally verified [57], a malicious piece of logic or compromised design tool may be able to exploit low level implementation details to quietly eavesdrop on, or interfere with, trusted logic. As a modern design may implement millions of logical gates with tens of millions of interconnections, the goal of this chapter is to explore design techniques that will allow the inclusion of both trusted and untrusted cores on a single chip, without the requirement that

\(^2\)Since designing reconfigurable modules is costly, companies have developed several schemes to protect this valuable intellectual property, which we discuss in Section 4.6.
expensive static verification be employed over the entire finished design. Such verification of a large and complex design requires reverse engineering, which is highly impractical because many companies keep details about their bit-streams proprietary.

### 4.2.1 Reconfigurable Hardware

FPGAs lie along a continuum between general-purpose processors and application-specific integrated circuits (ASICs). While general purpose processors can execute any program, this generality comes at the cost of serialized execution. On the other hand, ASICs can achieve impressive parallelism, but their function is literally hard wired into the device. The power of reconfigurable systems lies in their ability to flexibly customize an implementation down at the level of individual bits and logic gates without requiring a custom piece of silicon. This can often result in performance improvements on the order of 100x as compared to, per unit silicon, a similar microprocessor [14, 18, 96].

The growing popularity of reconfigurable logic has forced practitioners to begin to consider security implications, but as of yet there is no set of best design practices to guide their efforts. Furthermore, the resource constrained nature of embedded systems is perceived to be a challenge to providing a high level of security [49]. In this chapter we describe a set of low level methods that a) allow
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

effective reasoning about high level system properties, b) can be supported with minimal changes to existing tool flows, c) can be statically verified with little effort, d) incur relatively small area and performance overheads, and e) can be used with commercial off-the-shelf parts. The advantage of developing security primitives for FPGAs is that we can immediately incorporate our primitives into the reconfigurable design flow today, and we are not dependent on the often reluctant industry to modify the design of their silicon.

In the remainder of this chapter, we present our two concepts, moats and drawbridges, along with the details of how each maps to a modern reconfigurable device. In particular, for each approach we specify the threats that it addresses, the details of the technique and its implementation, and the overheads involved in its use. Finally, in Section 4.5, we show how these low-level protection mechanisms can be used in the implementation of a higher-level memory protection primitive.

4.3 Physical Isolation with Moats

As discussed in Section 4.2, a strong notion of isolation is lacking in current reconfigurable hardware design flows, yet one is needed to be certain that cores are not snooping on or interfering with each other. Before we can precisely describe
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

Figure 4.1: A simplified representation of an FPGA fabric is on the left. Configurable Logic Blocks (CLBs) perform logic level computation using Lookup Tables (LUTs) for bit manipulations and flip-flops for storage. The switch boxes and routing channels provide connections between the CLBs. SRAM configuration bits are used throughout the FPGA (e.g., to program the logical function of the LUTs and connect a segment in one routing channel to a segment in an adjacent routing channel). The FPGA floor plan on the right shows the layout of three cores – notice how they are intertwined.

the problem that moats attempt to solve, we need to begin with a brief description of how routing works (and the function it serves) in a modern FPGA.

On a modern FPGA, the vast majority of the actual silicon area is taken up by interconnect (approximately 90%). The purpose of this interconnect is to make it easy to connect logical elements together so that any circuit can be realized. For example, the output of one NAND gate may be routed to the input of another, or the address wires from a soft-processor may be routed to an I/O pad connected to external memory. The routing is completely static: a virtual wire is created
from input to output, but that signal may be routed to many different places simultaneously (e.g., one output to many inputs or vice versa).

The rest of the FPGA is a collection of programmable gates (implemented as small lookup-tables called LUTs), flip-flops for timing and registers, and I/O blocks (IOB) for transferring data into and out of the device. A circuit can be mapped to an FPGA by loading the LUTs and switch-boxes with a configuration, a method that is analogous to the way a traditional circuit might be mapped to a set of logical gates. An FPGA is programmed using a bitstream. This binary data is loaded into the FPGA to execute a particular task. The bitstream contains all the information needed to provide a functional device, such as the configuration interface and the internal clock cycle supported by the device.

Without an isolation primitive, it is very difficult to prevent a connection between two cores from being established. Place-and-route software uses performance as an objective function in its optimization strategy, which can result in the logical elements and the interconnections of two cores to be intertwined. Figure 4.2 makes the scope of the problem more clear. The left hand of Figure 4.2 shows the floor plan of an FPGA with two small cores (soft processors) mapped onto it. The two processors overlap significantly in several areas of the chip. Ensuring that the two never communicate requires that we trace every single wire to ensure that only the proper connections are made. Such verification of a large and
Figure 4.2: A simple two-core system mapped onto a small FPGA. The zoom-in to the right shows the wiring complexity at each and every switch-box on the chip. To statically analyze a large FPGA with 10s of cores and millions of logical gates, we need to restrict the degrees of freedom. Static verification of a large, complex design involving intertwined cores requires reverse engineering, which is highly impractical because many companies keep the necessary details about their bit-streams secret. With moats, fewer proprietary details about the bitstream are needed to accomplish this verification. The difficulty of this problem is made more clear by the zoom-in on the right of Figure 4.2. The zoom-in shows a single switch box, the associated LUTs (to the right of the switch box), and all the wires that cross through that one small portion of the chip. A modern FPGA contains on the order of 20,000 or more such boxes.
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

Figure 4.3: We use moats to physically isolate cores for security. In this example, segments can either span one or two switch boxes, which requires the moat size to have a length of two. Since the delay of a connection on an FPGA depends on the number of switch boxes it must pass through, restricting the length of segments reduces performance, but the moats can be smaller. Allowing longer segments improves performance, but the moats must waste more area.

Isolation is required in order to protect the confidentiality and integrity of a core’s data, and helps to prevent interference with a core’s functionality. Our technique allows a very simple static check to verify that, at least at the routing layer, the cores are sufficiently isolated.

4.3.1 Building Moats

Moats are a novel method of enhancing the security of FPGA systems via the physical isolation of cores. Our approach involves surrounding each core with a “moat” that blocks wiring connectivity from the outside. The core can only communicate with the outside world via a “drawbridge”, which is a precisely defined path to the outside world.
Chapter 4.  Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

One straightforward way to accomplish this is to align the routing tracks used by each of these modules and simply disable the switches near the moat boundaries. The problem with this simple approach is that, for the purposes of improving area and timing efficiency, modern FPGA architectures often support staggered, multiple track segments. For example, the Virtex platform supports track segments with lengths 1, 2 and 6, where the length is determined by measuring the number of Configuration Logic Blocks (CLBs) the segment crosses. For example, a length 6 segment will span 6 CLBs, providing a more direct connection by skipping unnecessary switch boxes along the routing path. Moreover, many platforms such as Virtex support “longline” segments, which span the complete row or column of the CLB array.

Figure 4.3 illustrates our moat architecture. If we allow the design tool to make use of segment lengths of one and two, the moat size must be at least two segments wide in order to successfully isolate two cores (otherwise signals could hop the moats because they would not require a switch box in the moat). To statically check that a moat is sound, the following properties are sufficient.

1. The target core is completely surrounded by moat of width at least $w$

2. The target core does not make any use of routing segments longer than length $w$
In fact, both of these properties are easy to inspect on an FPGA. We can tell if a switch box is part of a moat by simply checking that it is completely dead (i.e., all the routing transistors are configured to be disconnected). We can check the second property by examining all of the long line switch boxes to ensure that they are unused. These are easy to find because they are tied to the \emph{physical} FPGA design and are not a function of the specific core on the FPGA.

\subsection*{4.3.2 A Performance/Area Trade-off}

On an FPGA, the delay of a connection depends on the number of switch boxes it must pass through rather than the total length. Although large moats consume a great deal of chip area (because they reserve switch boxes without making use of them to perform an operation), they allow the design tools to make use of longer segments, which helps with the area and performance of each individual core. On the other hand, small moats require less chip area (for the moat itself), but having to use small segments negatively affects the area and performance of the cores.

A set of experiments is needed to understand the trade-offs between the size of the moats, the number of cores that can be protected using moats, and the performance and area implications for moat protection.
4.3.3 The Effect of Constrained Routing

We begin by quantifying the effect of constraining the tools to generate only configurations that do not use any routing segments longer than length $w$. The width of the moat could be any size, but the optimal sizes are dictated by the length of the routing segments. As mentioned before, FPGAs utilize routing segments of different sizes, most commonly 1, 2, 6 and long lines. If we could eliminate the long lines, then we would require a size 6 moat for protecting a core. By eliminating long lines and hex lines, we only need a moat of size 2, and so on.

In order to study the impact of eliminating certain long length segments on routing quality, we compare the routing quality of the MCNC benchmarks [60] on different segment configurations. We use the Versatile Placement and Routing (VPR) toolkit developed by the University of Toronto for such experiments. VPR provides mechanisms for examining trade-offs between different FPGA architectures and is popular within the research community [7]. Its capabilities to define detailed FPGA routing resources include support for multiple segment routing tracks and the ability for the user to define the distribution of the different segment lengths. It also includes a realistic cost model which provides a basis for the measurement of the quality of the routing result.

The effect of the routing constraints on performance and area can vary across different cores. Therefore, we route the 20 biggest applications from the MCNC
benchmark set [60] (the de facto standard for such experiments) using four different configurations. The baseline configuration supports segments with length 1, 2, 6 and longlines. The distribution of these segments on the routing tracks are 8%, 20%, 60% and 12% respectively, which is similar to the Xilinx Virtex II platform. The other three configurations are derived from the baseline configurations by eliminating the segments with longer lengths. In other words, configuration 1-2-6 will have no longlines, configuration 1-2 will support segments of length 1 and 2, and configuration 1 will only support segments of length 1.

After performing placement and routing, we measure the quality of the routing results by collecting the area and the timing performance based on the critical path of the mapped application. To be fair, all the routing tracks are configured using the same tri-state buffered switches with Wilton connection patterns [99] within the switch box. A Wilton switch box provides a good trade-off between routability and area, and is commonly used in FPGA routing architectures.

Figures 4.4 and 4.5 show the experimental results, where we provide the average hardware area cost and critical path performance for all the benchmarks over four configurations. The existence of longlines has little impact on the final quality of the mapped circuits. However, significant degradation occurs when we eliminate segments of length 2 and 6. This is caused by the increased demand for switch boxes, resulting in a larger hardware cost for these additional switch
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

![Average Area vs. Configuration](image)

**Figure 4.4:** Comparison of area for different configurations of routing segments. The baseline system has segments with length 1, 2, 6 and longline. The distribution is close to that of Virtex II: 8% (1), 20% (2), 60% (6) and 12% (longline). Other configurations are created by eliminating one or more classes of segments. For example, configuration 1-2-6 removes the longlines and distributes them proportionally to other types of segments.

resources. Moreover, the signal from one pin to another pin is more likely to pass more switches, resulting in an increase in the critical path timing. If we eliminate hex and long lines, there is a 14.9% area increase and an 18.9% increase in critical path delay, on average. If the design performance is limited directly by the cycle time, the delay in critical path translates directly into slowdown.

### 4.3.4 Overall Area Impact

While the results from Figures 4.4 and 4.5 show that there is some area impact from constraining the routing, there is also a direct area impact in the form of resources required to implement the actual moats themselves. Assuming that we
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

![Average Timing vs. Configuration](image)

**Figure 4.5:** Comparison of critical path timing for different configurations of routing segments. Unlike Figure 4.6, the graphs in Figures 4.4 and 4.5 do not include the overhead of the moat itself. The error bars show one standard deviation.

![Effective Utilization vs. Number of Cores](image)

**Figure 4.6:** The trade-off between the number of cores, the size of the moat, and the utilization of the FPGA. An increasing number of cores results in larger total moat area, which reduces the overall utilization of the FPGA. Larger moat sizes also will use more area resulting in lower utilization.
have a fixed amount of FPGA real estate, we really care about how much of that area is used up by a combination of the moats and the core inflation due to restricted routing. We can call this number the effective utilization. Specifically, the effective utilization is:

\[ U_{eff} = \frac{A_{AllRoutes}}{A_{RestrictedRoutes} + A_{Moats}} \]

Figure 4.6 presents the trade-offs between the moat size, the number of isolated cores on the FPGA, and the utilization of the FPGA. The FPGA used for these calculations was a Xilinx Virtex-4 Device which has 192 CLB rows and 116 CLB columns. The figure examines three different moat sizes: 1, 2 and 6 for a variable number of cores on the chip (conservatively assuming that a moat is required around all cores). As the number of cores increases, the utilization of the FPGA decreases since the area of the moats, which is unusable space, increases. However, when a small number of cores is used, a larger moat size is better because it allows us to make more efficient use of the non-moat parts of the chip. If you just need to isolate a single core (from the I/O pads) then a moat of width 6 is the best (consuming 12% of the chip resources). However, as the curve labeled “Moat Size = 2” in Figure 4.6 shows, a moat width of two has the optimal effective utilization for designs that have between two and 120 cores. As a point of reference, it should be noted that a modern FPGA can hold on the order of 100 stripped down microprocessor cores. The number of cores is heavily dependent on the
application, and the trade-off presented here is somewhat specific to our particular
platform, but our analysis method is still applicable to other designs. In fact, as
FPGAs continue to grow according to Moore’s Law, the percent overhead for
moats should continue to drop. Because the moats are perimeters, as the size of
a core grows by a factor of $n$, the cost of the moat only grows by $O(\sqrt{n})$.

4.3.5 Effective Scrubbing and Reuse of Reconfigurable Hardware

Moats allow us to reason about isolation without any knowledge of the inner
workings of cores, which are far too complex to feasibly determine whether a
particular element of a core is connected to another core. Furthermore, moats
also allow us to isolate cores designed with a less trustworthy tool chain from
cores that are the result of a more trustworthy tool chain. While these are both
useful properties, we need to make sure we can actually implement them. In fact,
a few of the latest FPGAs available have the ability to change a selective part of
their configuration, one column at a time [62]. A specialized core on the FPGA
can read one frame of the configuration, change part of this frame, and write the
modified frame back. This core must therefore be part of the trusted computing
base of the system.
Partial reconfiguration improves the flexibility of a system by making it possible to swap cores. If the number of possible configurations is small, then static verification is sufficient, but if the space of possible cores is infinite, then dynamic verification is necessary. For example, Baker et al. have developed an intrusion detection system based on reconfigurable hardware that dynamically swaps the detection cores [5] [4]. Since the space of intrusion detection rule sets is infinite, the space of detection cores is also infinite. We have developed a memory protection scheme for reconfigurable hardware in which a reconfigurable reference monitor enforces a policy that specifies the legal sharing of memory [38]. Partial reconfiguration could allow the system to change the policy being enforced by swapping in a different reference monitor. Since the space of possible policies is infinite, the space of possible reference monitors is also infinite. Lysaght and Levi have devised a dynamically reconfigurable crossbar switch [61]. By using dynamic reconfiguration, their 928x928 crossbar uses 4,836 CLBs compared to the 53,824 CLBs required without reconfiguration.

To extend our model of moats to this more dynamic case, we not only need to make sure that our static analysis must be simple enough to be performed on-line by a simple embedded core (which we argue it is), but we also need to make sure that nothing remains of the prior core’s logic when it is replaced with a
different core. In this section, we describe how we can enable object reuse through configuration scrubbing.

By rewriting a selective portion of the configuration bits for a certain core, we can erase any information it has stored in memory or registers. The ICAP (Internal Configuration Access Port) on Xilinx devices allows us to read, modify, and write back the configuration bitstream on Virtex II devices. The ICAP can be controlled by a Microblaze soft core processor or an embedded PowerPC processor if the chip has one. The ICAP has an 8-bit data port and typically runs at a clock speed of 50 MHz. Configuration data is read and written one frame at a time. A frame spans the entire height of the device, and frame size varies based on the device.

Table 4.1 gives some information on the size and number of frames across several Xilinx Virtex II devices. The smallest device has 404 frames, and each frame requires 5.04 us to reconfigure, or equivalently, erase. Therefore, reconfiguring (erasing) the entire devices takes around 2 ms.

<table>
<thead>
<tr>
<th>Device</th>
<th># Frames</th>
<th>Frame Length</th>
<th>R/W time for 1 frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC2V40</td>
<td>404</td>
<td>26</td>
<td>5.04 us</td>
</tr>
<tr>
<td>XC2V500</td>
<td>928</td>
<td>86</td>
<td>14.64 us</td>
</tr>
<tr>
<td>XC2C2000</td>
<td>1456</td>
<td>146</td>
<td>24.24 us</td>
</tr>
<tr>
<td>XC2V8000</td>
<td>2860</td>
<td>286</td>
<td>46.64 us</td>
</tr>
</tbody>
</table>
To sanitize a core we must perform 3 steps. First we must read in a configuration frame. The second step is to modify the configuration frame so that the flip-flops and memory are erased. The last step is to write back the modified configuration frame. The number of frames and how much of the frame we must modify depend on the size of the core that is being sanitized. This process must be repeated since each core will span the width of many frames. In general, the size of the core is linearly related to the time that is needed to sanitize it.

Our object reuse technique can also disable a core if extreme circumstances should require it, such as tampering. Embedded devices such as cell phones are very difficult to sanitize [69]. Smart phones contain valuable personal data, and the theft or loss of a phone can result in serious consequences such as identity theft. Embedded devices used by the military may contain vital secrets that must never fall into enemy hands. Furthermore, valuable IP information of the cores is stored in the form of the bitstream on the FPGA. A method of disabling all or part of the device is needed to protect important information stored on the FPGA in the extreme case of physical tampering.

The IBM 4758 is an example of a cryptographic coprocessor that has been designed to detect tampering and to disable itself whenever tampering occurs [97]. The device is surrounded by specialized packaging containing wire mesh.
Any tampering of the device disturbs this mesh, and the device can respond by disabling itself.

4.4 Drawbridges: Interconnect Interface Conformance with Tracing

In the previous section, we described an effective method for isolating cores using moats. Our moat methodology eliminates the possibility for external cores to tap into the information contained in a core surrounded by the moat. However, cores do not work in isolation and must communicate with other cores to receive and send data. Therefore, we must allow controlled entry into our core. The entry or communication is only allowed with prespecified transactions through a “drawbridge”. We must know in advance which cores we need to communicate with and the location of those cores on the FPGA. Often times, it is most efficient to communicate with multiple cores through a shared interconnection (i.e., a bus). Again, we must ensure that bus communications are received by only the intended recipient(s). Therefore, we require methods to ensure that 1) communication is established only with the specified cores and 2) communication over a shared medium does not result in a covert channel. In this section, we
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

present two techniques, interconnect tracing and a bus arbiter, to handle these two requirements.

We have developed an interconnect tracing technique for preventing unintended flows of information on an FPGA. Our method allows a designer to specify the connections on a chip, and a static analysis tool checks that each connection only connects the specified components and does not connect with anything else. This interconnect tracing tool takes a bitstream file and a text file that defines the modules and interconnects in a simple language which we have developed. The big advantage of our tool is that it allows us to perform the tracing on the bitstream file. We do not require a higher level description of the design of the core. Performing this analysis during the last stage of design allows us to catch illegal connections that could have originated from any stage in the design process including the design tools themselves.

In order for the tracing to work we must know the locations of the modules on the chip and the valid connections to/from the modules. To accomplish this we place moats around the cores during the design phase. We now know the location of the cores and the moats, and we use this information to specify a text file that defines: all the cores along with their location on the chip, all I/O pins used in the design, and a list of valid connections. Then our tool uses the JBits API [29] to analyze the bitstream and check to make sure there are no invalid connections.
in the design. The process of interconnect tracing is performed by analyzing the bitstream to determine the status of the switchboxes. We can use this technique to trace the path that a connection is routed along and ensure that it goes where it is supposed to. This tracing technique allows us to ensure that the different cores can only communicate through the channels we have specified and that no physical trap doors have been added anywhere in the design.

Ensuring that interconnects between modules are secure is a necessity to developing a secure architecture. This problem is made more complicated by the abundance of routing resources on an FPGA and the ease with which they can be reconfigured. Our proposed interconnect tracing technique allows us to ensure the integrity of connections on a reconfigurable device. This tool gives us the ability to perform checking in the final design stage: right before the bitstream is loaded onto the device.

4.4.1 Efficient Communication under the Drawbridge Model

In modern reconfigurable systems, cores communicate with each other via a shared bus. Unfortunately, the shared nature of a traditional bus architecture raises several security issues. Malicious cores can obtain secrets by snooping on the bus. In addition, the bus can be used as a covert channel to leak secret data
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

Figure 4.7: Architecture alternative 1. There is a single arbiter and each module has a dedicated connection to the arbiter.

Figure 4.8: Architecture alternative 2. Each module has its own arbiter that prevents bus snooping and a central time multiplexer that connects to all the arbiters.
from one core to another. The ease of reconfigurability on FPGAs allows us to address these problems at the hardware level.

To address this problem of covert channels and bus snooping, we have developed a shared memory bus with a time division access. The bus divides the time equally among the modules, and each module can read/write one word to/from the shared memory during its assigned time slice. Our approach of arbitrating by time division eliminates covert channels. With traditional bus arbitration, there is a possibility of a bus-contention covert channel to exist in any shared bus system where multiple cores or modules access a shared memory. Via this covert channel, a malicious core can modulate its bus references, altering the latency of bus references for other modules. This enables the transfer of information between any two modules that can access the bus [36]. This covert channel could be used to send information from a module with a high security clearance to a module with lower security clearance (write-down), which would violate a Bell-LaPadula multilevel policy and cannot be prevented through the use of the reference monitor. To eliminate this covert channel, we give each module an equal share of time to use the bus, eliminating the transfer of information by modulating bus contention. Since each module can only use the bus during its allotted time slice, it has no way of changing the bus contention. One module cannot even tell if any of the other modules are using the bus. While this does limit performance
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

of the bus, it removes the covert channel. The only other feasible way that we see to remove this covert channel is to give each module a dedicated connection to all other modules. Requiring a dedicated direct connection between each set of modules that need to communicate would be inefficient and costly. Dedicated channels would require a worst case of $O(n^2)$ connections, where $n$ is the number of modules in the design. Our architecture requires only $O(n)$ connections.

Bus snooping is another major concern associated with a shared bus. Even if we eliminate the covert channels there is nothing to prevent bus snooping. For example, let us consider a system where we want to send data from a classified module to another and where there are unclassified modules on the same bus. We need a way to ensure that these less trusted modules cannot obtain this information by snooping the bus. To solve this problem, we place an arbiter between the module and the memory. The arbiter only allows each module to read during its time share of the bus. In addition a memory monitor is required, but for this work we assume that such a configuration can be implemented on the FPGA using the results of our work on memory protection [38].

4.4.2 Architecture Alternatives

We devised two similar architectures to prevent snooping and to eliminate covert channels on the bus. In our first architecture, each module has its own
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

separate connection to a single arbiter, which sits between the shared memory and the modules. This arbiter schedules access to the memory equally according to a time division scheduling (Figure 4.7). A module is only allowed to read or write during its allotted time, and when a module reads, the data is only sent to the module that issued the read request. The second architecture is more like a traditional bus. In this design, there is an individual arbiter that sits between each module and the bus. These arbiters are all connected to a central timing module which handles the scheduling (Figure 4.8). The individual arbiters work in the same way as the single arbiter in the first architecture to prevent snooping and to remove covert channels. To make interfacing easy, both of these architectures have a simple interface so that a module can easily read/write to the shared memory without having to worry about the timing of the bus arbiter.

During the design process, we found that the first architecture seemed easier to implement, but we anticipated that the second architecture would be more efficient. In our first architecture (Figure 4.7, everything is centralized, making the design of a centralized memory monitor and arbiter much easier to design and verify. In addition, a single moat could be used to isolate this functionality. Our second architecture (Figure 4.8) intuitively should be more scalable and efficient since it uses a bus instead of individual connections for each module, but the
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

Table 4.2: Comparison of Communication Architectures

<table>
<thead>
<tr>
<th></th>
<th>Architecture 1</th>
<th>Architecture 2</th>
<th>Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>146</td>
<td>169</td>
<td>15.75</td>
</tr>
<tr>
<td>Flip Flops</td>
<td>177</td>
<td>206</td>
<td>16.38</td>
</tr>
<tr>
<td>4 Input LUTs</td>
<td>253</td>
<td>305</td>
<td>20.55</td>
</tr>
<tr>
<td>Max Clock Frequency</td>
<td>270.93</td>
<td>271.297</td>
<td>0.14</td>
</tr>
</tbody>
</table>

arbiters have to coordinate, the memory monitor has to be split (if that is even possible), and each arbiter need to be protected by its own moat.

To test our hypotheses, we developed prototypes of both of the architectures. The prototypes were developed in VHDL and synthesized for a Xilinx Virtex-II device in order to determine the area and performance of the designs on a typical FPGA. We did not account for the extra moat or monitor overhead, but with this assumption results of the analysis of the two architectures, which can be seen in Table 4.2, were not what we first expected. During synthesis of the second architecture, the synthesis tool converted the tri-state buffers\(^3\) in the bus to digital logic. As a result, the second architecture used more area than the first and only had a negligible performance advantage. Contrary to what we expected, the first architecture used roughly 15% less area on the FPGA and is simpler to implement and verify. Since the performance difference between the two was almost negligible, the first architecture is the better design choice.

\(^3\)tri-state buffers are gates that can output either a 0, 1, or Z – a high impedance state in which the gate acts as if it was disconnected from the wire.
This bus architecture allows modules to communicate securely with a shared memory and prevents bus snooping and certain covert channels. When combined with the reference monitor this secure bus architecture provides a secure and efficient way for modules to communicate.

4.5 Application: Memory Policy Enforcement

Now that we have described isolation and its related primitives, we provide an example of the application of isolation to memory protection, an even higher-level primitive. Saltzer and Schroeder identify three key elements that are necessary for protection: “Conceptually, then, it is necessary to build an impenetrable wall around each distinct object that warrants separate protection, construct a door in the wall through which access can be obtained, and post a guard at the door to control its use.” [79]. In addition, the guard must be able to identify the authorized users. In the case of protecting cores, our moat primitive is analogous to the wall, and our drawbridge primitive is analogous to the door. Our interconnect tracing and secure bus primitives act as the guard.

One way of protecting memory in an FPGA system is to use a reference monitor that is loaded onto the FPGA along with the other cores [38]. Here, the reference monitor is analogous to the guard because it decides the legality of
Chapter 4. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

every memory access according to a policy. This requires that every access go through the reference monitor. Without our isolation primitive, it is easy for a core to bypass the reference monitor and access memory directly. Since moats completely surround a core except for a small amount of logic (the drawbridge) for communicating with the rest of the chip, it is much easier to prevent a core from bypassing the reference monitor.

Saltzer and Schroeder describe how protection mechanisms can protect their own implementations in addition to protecting users from each other [79]. Protecting the reference monitor from attack is critical to the security of the system, but the fact that the reference monitor itself is reconfigurable makes it vulnerable to attack by the other cores on the chip. However, moats can mitigate this problem by providing physical isolation of the reference monitor.

Our isolation primitive also makes it harder for an unauthorized information flow from one core to another to occur. Establishing a direct connection between the two cores would clearly thwart the reference monitor. If moats surround each core, it is much harder to connect two cores directly without crossing the moat.

As we described above, a reference monitor approach to memory protection requires that every memory access go through the reference monitor. However, cores are connected to each other and to main memory by means of a shared bus. As we explained in Section 4.4.1, the data on a shared bus is visible to all cores.
Our secure bus primitive protects the data flowing on the bus by controlling the sharing of the bus with a fixed time division approach.

A memory protection system that allows dynamic policy changes requires an object reuse primitive. It is often useful for a system to be able to respond to external events. For example, during a fire, all doors in a building should be unlocked without exception (a more permissive policy than normal), and all elevators should be disabled (a less permissive policy than normal). In the case of an embedded device, a system under attack may wish to change the policy enforced by its reference monitor. There are several ways to change policies. One way is to overwrite the reference monitor with a completely different one. Our scrubbing primitive can ensure that no remnants of the earlier reference monitor remain. Since cores may retain some information in their local memory following a policy change, our scrubbing primitive can also be used to scrub the cores.

4.6 Related Work

There has always been an important relationship between the hardware a system runs on and the security of that system. Reconfigurable systems are no different, although to the best of our knowledge we are the first to address the problem of isolation and physical interface conformance on them. However, in
addition to the related work we have already mentioned, we do build on the results of prior related efforts. In particular, we build on the ideas of reconfigurable security, IP protection, secure update, covert channels, direct channels, and trap doors. While a full description of all prior work in these areas is not possible, we highlight some of the most related.

4.6.1 Reconfigurable Hardware Security

To provide memory protection on an FPGA, we propose the use of a reconfigurable reference monitor that enforces the legal sharing of memory among cores [38]. A memory access policy is expressed in a specialized language, and a compiler translates this policy directly to a circuit that enforces the policy. The circuit is then loaded onto the FPGA along with the cores. While their work addresses the specifics of how to construct a memory access monitor efficiently in reconfigurable hardware, they do not address the problem of how to protect that monitor from routing interference, nor do they describe how to enforce that all memory accesses go through this monitor. This chapter directly supports their work by providing the fundamental primitives that are needed to implement memory protection on a reconfigurable device.

There appears to be little other work on the specifics of managing FPGA resources in a secure manner. Chien and Byun have perhaps the closest work,
where they addressed the safety and protection concerns of enhancing a CMOS processor with reconfigurable logic [15]. Their design achieves process isolation by providing a reconfigurable virtual machine to each process, and their architecture uses hardwired TLBs to check all memory accesses. Our work could be used in conjunction with theirs, using soft-processor cores on top of commercial off-the-shelf FPGAs rather than a custom silicon platform. In fact, we believe one of the strong points of our work is that it may provide a viable implementation path to those that require a custom secure architecture, for example execute-only memory [58] or virtual secure co-processing [54].

Gogniat et al. propose a method of embedded system design that implements security primitives such as AES encryption on an FPGA, which is one component of a secure embedded system containing memory, I/O, CPU, and other ASIC components [26]. Their Security Primitive Controller (SPC), which is separate from the FPGA, can dynamically modify these primitives at runtime in response to the detection of abnormal activity (attacks). In this work, the reconfigurable nature of the FPGA is used to adapt a crypto core to situational concerns, although the concentration is on how to use an FPGA to help efficiently thwart system level attacks rather than chip-level concerns. Indeed, FPGAs are a natural platform for performing many cryptographic functions because of the large number of bit-level operations that are required in modern block ciphers. However, while there is a
great deal of work centered around exploiting FPGAs to speed cryptographic or intrusion detection primitives, systems researchers are just now starting to realize the security ramifications of building systems around hardware which is reconfigurable.

Most of the work relating to FPGA security has been targeted at the problem of preventing the theft of intellectual property and securely uploading bitstreams in the field. Because such attacks directly impact their bottom line, industry has already developed several techniques to combat the theft of FPGA IP, such as encryption [12] [45] [46], fingerprinting [51], and watermarking [52]. However, establishing a root of trust on a fielded device is challenging because it requires a decryption key to be incorporated into the finished product. Some FPGAs can be remotely updated in the field, and industry has devised secure hardware update channels that use authentication mechanisms to prevent a subverted bitstream from being uploaded [34] [33]. These techniques were developed to prevent an attacker from uploading a malicious design that causes unintended functionality. Even worse, the malicious design could physically destroy the FPGA by causing the device to short-circuit [31]. However, these authentication techniques merely ensure that a bitstream is authentic. An “authentic” bitstream could contain a subverted core that was designed by a third party.
4.6.2 Covert Channels, Direct Channels, and Trap Doors

The work in Section 4.4.1 directly draws upon the existing work on covert channels. Exploitation of a covert channel results in the unintended flow of information between cores. Covert channels work via an internal shared resource, such as power consumption, processor activity, disk usage, or error conditions [88] [73]. Classical covert channel analysis involves the articulation of all shared resources on chip, identifying the share points, determining if the shared resource is exploitable, determining the bandwidth of the covert channel, and determining whether remedial action can be taken [47]. Storage channels can be mitigated by partitioning the resources, while timing channels can be mitigated with sequential access, a fact we exploit in the construction of our bus architecture. Examples of remedial action include decreasing the bandwidth (e.g., the introduction of artificial spikes (noise) in resource usage [80]) or closing the channel. Unfortunately, an adversary can extract a signal from the noise, given sufficient resources [67].

Of course our technique is primarily about restricting the opportunity for direct channels and trap doors [91]. Our memory protection scheme is an example of that. Without any memory protection, a core can leak secret data by writing the data directly to memory. Another example of a direct channel is a tap that connects two cores. An unintentional tap is a direct channel that can be estab-
lished through luck. For example, the place-and-route tool’s optimization strategy may interleave the wires of two cores.

4.7 Summary

The design of reconfigurable systems is a complex process, with multiple software tool chains that may have different trust levels. Since it is not cost-effective to develop an optimized tool chain from scratch to meet assurance needs, only the most sensitive cores should be designed using a trusted tool chain. To meet performance needs, most cores could be designed with commodity tools that are highly optimized but untrusted, which results in multiple cores on a chip with different trust levels. Our methodology will not lead to those less trusted portions becoming more dependable or correct, but it will isolate trusted portions from the effects of their subversion or failure. To address this situation, developers will need to build monitored or fail-safe systems on top of FPGAs to prevent the theft of critical secrets.

We have presented two low-level protection mechanisms to address these challenges, moats and drawbridges, and we have analyzed the trade-offs of each. Although larger moats consume more area than smaller moats, they have better performance because longer segments can be used. We are working on an im-
proved version of moats in which no area must be dedicated to dead areas for the moats. Although cores still must be kept separate, they can touch each other, and static checking is performed near the borders where cores touch. Our interconnect tracing primitive works together with our moat primitive in a complementary way by allowing smaller moats to be used without sacrificing performance. We have also described how these basic primitives are useful in the implementation of a higher-level memory protection primitive, which can prevent unintended sharing of information in embedded systems.
Chapter 5

Detecting Covert Channels in Stateful Policy Enforcement Systems

If we knew what it was we were doing, it would not be called research, would it?

Albert Einstein (1879-1955)

You have enemies? Good. That means you’ve stood up for something, sometime in your life.

Winston Churchill (1874-1965)

5.1 Introduction

In this chapter, we present a technique for analyzing stateful policies to detect possible covert channels during the design phase so that the internal state of the reference monitor cannot be used as a covert channel. Security is a fundamental design parameter for modern computer systems. Many systems employ low-level
mechanisms to enforce security policies (i.e., formal top-level specifications). For example, network intrusion detection systems use sensors that are strategically placed throughout the network to scan traffic and identify attacks so that corrective action can be taken [50]. Firewalls can be configured to filter out traffic that offends a stateful policy [28]. Stateful policy enforcement mechanisms are incorporated into a wide variety of systems, including wireless routers, embedded devices, application servers, operating systems, peer-to-peer computing, web services, and user applications. These mechanisms can enforce both stateful and stateless policies. Some FPGA-based embedded systems use reconfigurable reference monitors to enforce stateful policies that specify the legal sharing of memory [38].

Schneider describes the class of security policies that can be enforced with an execution monitor [83]. Such an enforcement mechanism is a reference monitor, which decides the legality of a particular request to access a system resource. If a subject attempts to violate the policy, corrective action must be taken, such as terminating the subject or substituting an acceptable execution step for an unacceptable one. Although the importance of not allowing a subject to violate the policy should be obvious, there is another more subtle reason for the need to respond to policy violations: covert channels. A reference monitor makes a binary decision to either grant or deny a particular access. If the reference monitor is
Chapter 5. Detecting Covert Channels in Stateful Policy Enforcement Systems

enforcing a stateful policy, subjects can observe the internal state of the policy by observing this decision. Subjects can also change the internal state by making access requests. This ability to observe and modify a shared resource makes it possible for two subjects to establish a covert channel. We believe that terminating a subject that violates the policy is naive because this may disable critical services. A better approach is to detect covert channels in stateful policies so that they can be eliminated during the design phase.

We consider policies with bounded states, specifically those for which the language of legal accesses is regular and can be described by a finite state machine. We do not consider policies with unbounded states or those policies that use the notion of time to determine legality. The class of policies with bounded states is important because it encompasses many classic security scenarios, including isolation, Chinese wall, and high water mark.

Some stateful policies only allow a few bits to be leaked, while others allow an unbounded amount of data to be leaked. A stateful policy expressed in a regular language is equivalent to a directed graph, and some graphs contain cycles which allow the internal state of the policy to alternate between two or more states an infinite number of times. If certain properties of the cycle are met, then a possible unbounded storage channel exists. We have developed an automated way of detecting these unbounded channels. Once one of these storage channels
Chapter 5. Detecting Covert Channels in Stateful Policy Enforcement Systems

has been identified, the best course of action is to revise the policy to eliminate the cycle, but this is not always possible. Therefore, we propose a technique for coping with such a storage channel by counting the number of times that a cycle goes around at runtime. If the counter exceeds a threshold, the system dynamically changes to a policy in which the covert channel has been eliminated. This limits the amount of data that can be leaked to a specific value. Specifically, the contributions of this chapter are:

- A precise definition of storage channels in policy enforcement systems
- An automated method of detecting unbounded storage channels
- A technique for measuring the bandwidth of a storage channel at runtime
- A description of different options for taking corrective action

5.2 Storage Channels in Stateful Policy Enforcement Systems

Kemmerer [47] [48] has devised a shared resource matrix method of identifying storage and timing channels in computer systems. In both storage and timing channels, the sender has a higher security label than the receiver. In a storage channel, the sender alters a data item, and the receiver interprets the value. In a
timing channel, the sender modulates the time needed for the receiver to perform a task, and the receiver interprets the delay or lack of delay. This chapter will focus on storage channels rather than timing channels because our research has concentrated on developing hardware reference monitors that make a grant or deny decision in constant time: precisely one cycle. We leave to future work the application of our methods to timing channels. In this section, we will show how our reference monitor can be used as a storage channel if the policy has certain properties.

Throughout this chapter, we will consider as a motivating example the case of a reference monitor that is part of an embedded system that contains multiple modules (IP cores) that each perform a particular function. Since IP cores are frequently obtained from untrusted third parties or are created using untrusted tool chains, the cores on a device may operate at different trust levels. Therefore, a core with a higher security label must be prevented from using a shared resource to leak data to a core with a lower security label. Cores that reside on the same device need to share resources such as memory, but a core must be prevented from reading or modifying another core’s data. Therefore, the reference monitor enforces a policy that specifies the legal sharing of memory among cores. The principals in the system are the modules, and the objects in the system are specific
ranges of memory. Each module has a unique ID (e.g., Module$_1$), and each range has a unique ID (e.g., Range$_1$).

Kemmerer identifies four criteria for storage channels: the sender and receiver must have access to the same shared resource attribute, the sender must be able to change the shared attribute, the receiver must be able to detect the change, and the sender and receiver must be able to initiate communication and sequence events [47] [48]. In the case of a reference monitor, the sender and the receiver are cores, both of which must have access to the reference monitor. Since a policy is enforced by a DFA, we can think of policies as directed graphs. Each node of the graph is a state of the policy, and each edge is a transition of the policy. In order for a policy to have a storage channel, there must be a non-trivial cycle in the graph. A trivial cycle is a transition from a node to itself, and a non-trivial cycle is any cycle involving two or more nodes. Of course, stateless policies do not have non-trivial cycles. The presence of a cycle in the graph allows the internal state of the reference monitor to alternate between two or more states an infinite number of times. This property allows an unbounded amount of data to be leaked via the covert channel. The sender must be able to change the state of the policy within the cycle by causing at least one DFA transition within the cycle. Finally, the receiver must be able to detect a change in the state. Therefore, at least two
nodes within the cycle must have different access matrices with respect to the receiver.

We assume the most conservative assignment of security labels to principals that results in the largest information flow. For example, if a possible covert channel from Module$_1$ to Module$_2$ is identified, we assume that Module$_1$ has a higher security label than Module$_2$. This worst-case analysis should assume that the channel that receives the most information should be assumed to have a low security label.

Figure 5.1 shows a DFA that enforces a memory access policy. Each node in the graph is a state in the policy, and we show the access matrix at each node. The columns of the access matrix are the principals (modules), and the rows are the objects (ranges). This DFA contains a non-trivial cycle that satisfies the criteria for a storage channel. The symbols of our language are triples consisting of module ID, access right, and range ID. For example, if principal Module$_1$ reads object Range$_1$, we express this as \( \{ \text{Module}_1, r, \text{Range}_1 \} \), or \( \{ M_1, r, R_1 \} \) for short. In this example, Module$_1$ has a higher security label than Module$_2$, and we claim that a necessary condition for a covert channel from Module$_1$ to Module$_2$ exists because Module$_1$ controls at least one of the transitions within the cycle and at least two nodes within the cycle have access matrices that differ with respect to Module$_2$. Initially, Module$_2$ can read Range$_1$. Module$_1$ can then change the
state by reading $Range_1$. Now, $Module_2$ can no longer read $Range_1$. $Module_1$ can then change the state again by reading $Range_1$.

We now describe how this cycle is used to establish an illegal information flow from $Module_1$ to $Module_2$. $Module_2$ continually tries to read $Range_1$. If access is granted, $Module_2$ knows that the current state is the first state, but if access is denied, $Module_2$ knows that the current state is the second state. $Module_1$ alternates between the two states by reading $Range_1$. $Module_2$ receives a bit of information when the current state remains stable for a fixed number of cycles $T$. Another way of transmitting a bit is to treat one complete cycle as a bit, similar to a Morse code pulse. There are many ways for the sender to encode the data to be leaked. The magnitude of the information flow can be calculated in terms on the number of possible encodings of the data and the probability of each symbol [67] [68].

As mentioned above, one of the criteria for a storage channel is that the sender must be able to change the shared attribute. The sender does not have to be able to cause every DFA transition within the cycle. One is sufficient because the remaining transitions can be caused by the receiver. If neither the sender nor the receiver is able to cause a particular transition within the cycle, the sender can wait a large number of cycles so that such a transition is very likely to occur. This allows the cycle to come around again. This case is shown in Figure 5.2. $Module_1$ would
like to send some data to Module$_2$, but Module$_3$ controls one of the transitions. Module$_1$ simply waits a sufficient length of time during which Module$_3$’s transition is likely to occur. Clearly, we are being conservative in assuming that a possible unbounded covert channel exists from Module$_1$ to Module$_2$ because Module$_3$’s transition could occur infrequently, resulting in a low bandwidth. Since we are using a static technique to detect possible covert channels, not all possible channels identified can be exploited at runtime.

Another criteria for a storage channel mentioned above is that the receiver must be able to detect the change. Not every node in the cycle must differ with respect to the receiver. In fact, the access matrices of just two states within the cycle must differ with respect to the receiver. Suppose we have a large cycle with many nodes, most of which have access matrices that are identical with respect to the receiver. If just two of them differ, the receiver will be able to detect that the cycle has repeated. Figure 5.3 shows this case. In this cycle that contains three nodes, only one of the nodes differs from the other two nodes with respect to Module$_2$. Still, Module$_2$, is able to detect the change, allowing data to leak from Module$_1$, which controls all of the transitions within the cycle, to Module$_2$. 

112
Figure 5.1: A non-trivial cycle. This figure shows the DFA that enforces a security policy. Each node of the graph is a state in the policy, and we show the access matrix at each node. Module\textsubscript{1} has a higher security label than Module\textsubscript{2}. Initially, Module\textsubscript{2} can read Range\textsubscript{1}. Module\textsubscript{1} can then change the state by reading Range\textsubscript{1}. Now, Module\textsubscript{2} can no longer read Range\textsubscript{1}. Module\textsubscript{1} can then change the state again by reading Range\textsubscript{1}. According to our criteria, there is a possible storage channel from Module\textsubscript{1} to Module\textsubscript{2}.
Figure 5.2: Suppose that Module\textsubscript{1} would like to leak some data to Module\textsubscript{2}. In this example, Module\textsubscript{1} causes one of the transitions, and Module\textsubscript{3} causes the other transition. The two access matrices differ with respect to Module\textsubscript{2}. Since Module\textsubscript{3} is not a party in the exchange, Module\textsubscript{1} must wait a sufficient length of time for Module\textsubscript{3}’s transition to occur, allowing the cycle to come around again.
Figure 5.3: A cycle consisting of three nodes. Two of the nodes are identical with respect to Module2, but one is different from the other two. Since at least one node differs, Module2 can detect the change, allowing data to leak from Module1 to Module2.
5.3 Automatically Detecting Storage Channels

A program that automatically detects a possible covert channel in a policy must first determine if its DFA has any cycles. Topological sort is a well-known algorithm for detecting cycles in a directed graph [17]. The first step is to select a vertex in the graph with no incoming edges and remove it, repeating this process until there are no more vertices left. If this process cannot finish, then the graph contains a cycle. Identifying the set(s) of vertices that make up the cycle(s) involves tracing the graph recursively. The following pseudo-code demonstrates this process:

Procedure DetectChannels (Graph G)
{
    Array of Lists Senders
    Array of Lists Receivers
    If (Topological_Sort(G) == False)
    
        Output ‘‘Graph G Contains No Cycles.’’
        Return
    
    Recursively_Trace_Graph_to_Find_Cycles(G)
    For (All Cycles C Found)
        For (All Edges E in C)
            M = Module that causes transition E
            Add M to Senders[C]
        
        For (All Vertices V in C)
            Matrix M1 = Access Matrix of V
            For (All Vertices V' in C)
                Matrix M2 = Access Matrix of V'
                CompareMatrices(M1, M2, Receivers[C])
                Output ‘‘Possible Covert Channels:’’
        
        For (All Cycles C Found)
            Output Cross_Product(Senders[C], Receivers[C])
    
    Return
}
Procedure CompareMatrices(Matrix M1, Matrix M2, List R)
{
    For (All Rows R)
        For (All Columns C)
            If (M1[R][C] != M2[R][C])
                Add C to R

We applied our covert channel detector to several example policies. Figure 5.4 shows a redaction policy, which alternates between a more restrictive and less restrictive access matrix. Our detector identified four possible covert channels: from Module1 to Module2, from Module1 to Module3, from Module3 to Module1, and from Module3 to Module2. Figure 5.6 demonstrates a transitive property of covert channels: a possible channel from Module1 to Module2 and a possible channel from Module2 to Module3 implies a possible channel from Module1 to Module3.

Figure 5.5 shows a policy that dynamically switches between a B&L policy and a Biba policy, and there is a possible covert channel from Module1 to Module2. The Trigger transition (\{Module1,w,Range_8\}) causes the policy change from B&L to Biba, and the Clear transition (\{Module1,w,Range_9\}) causes the policy to change from Biba to B&L. Dynamic policies that switch back and forth between two or more policies frequently have possible covert channels [102] [104], and our detector is ideal for finding them. One way of dealing with the problem of covert
Chapter 5. Detecting Covert Channels in Stateful Policy Enforcement Systems

channels in dynamic policies is to have a module with a low security label perform the policy transitions. If this is not possible and a module with a higher security label must be used to perform the policy transitions, then it is essential that this module be trusted. Even if a covert channel exists, as long as policy switching is infrequent, the bandwidth is low, and we describe the use of counters to measure this bandwidth in Section 5.4.

Figure 5.7 shows a Chinese wall policy with two conflict-of-interest classes: \(\{\text{Range}_1,\text{Range}_2\}\) and \(\{\text{Range}_3,\text{Range}_4\}\). Although it does not have any cycles, this policy is not completely free of covert channels. Module_1 could leak one bit of information to Module_2 and one bit to Module_3, or Module_1 could leak one bit to Module_2 and Module_4, or Module_1 could leak one bit to Module_3 and Module_4. While two bits does not seem like a lot of information, there are some highly sensitive applications for which even leaking two bits is unacceptable. In a graph that does not have any cycles, the maximum amount of information that can be leaked depends on the longest path length from the initial state to any final state.
Chapter 5. Detecting Covert Channels in Stateful Policy Enforcement Systems

Figure 5.4: This redaction policy has four possible covert channels: from Module$_1$ to Module$_2$, from Module$_1$ to Module$_3$, from Module$_3$ to Module$_1$, and from Module$_3$ to Module$_2$.

Figure 5.5: A dynamic policy that switches between a B&L policy and a Biba policy. There is a possible covert channel from Module$_1$ to Module$_2$. 
Figure 5.6: This example policy shows a transitive property of covert channels. A possible channel from Module\textsubscript{1} to Module\textsubscript{2} and a possible channel from Module\textsubscript{2} to Module\textsubscript{3} implies a possible channel from Module\textsubscript{1} to Module\textsubscript{3}.
5.4 Measuring the Bandwidth of Storage Channels

Once a possible covert channel has been identified, the system designer can modify the policy in order to eliminate the problematic cycle. If this is not an option, one way of coping is to use counters to measure the bandwidth of the covert channel. A counter keeps track of the number of times the cycle occurs, and the system ensures that the counters stay below a threshold value. In Section 5.5 we describe an option for corrective action should a counter exceed its threshold. A cycle can be expressed as a regular expression, and a piece of hardware to
recognize this expression can be easily built. For example, a cycle from State\textsubscript{1} to State\textsubscript{2} to State\textsubscript{3} and back to State\textsubscript{1} can be expressed as $S_1(S_2S_3S_1)^+$. Regular expressions can even identify large cycles that contain smaller cycles within. This “monitor monitor” can be incorporated into the reference monitor. The price of this measurement mechanism should be balanced against the cost of ensuring that the module with a high security label will not leak secret information. Typically, the cost of such mechanisms is much lower than the price of ensuring that a module is trusted.

5.5 Options for Corrective Action

The best way of dealing with a possible covert channel is to modify the policy in order to eliminate it. This can be done by removing the cycle, but if the cycle cannot be eliminated, the next best thing is to change the transitions within the cycle or the access matrices of the nodes of the cycle in order to eliminate the possible covert channel. If this is not possible, then the security labels of the modules should be changed so that information cannot flow from a high module to a low module. If this is not possible then we need to be able to cope with the covert channel by using our counter-based technique. In Section 5.4, we described a technique of using counters to measure the bandwidth of a covert channel in a
policy. In this section, we present an option for corrective action should a counter exceed its threshold value. As we explained in Section 5.2, terminating a core is highly problematical because critical services may be disabled. Rather than terminating the receiver, we propose changing the policy in response to a counter exceeding its threshold.

Figure 5.8 shows an example of this concept. A stateful policy with two states has a cycle resulting in a possible covert channel from Module$_1$ to Module$_2$. A counter monitors the number of times the cycle completes. If the counter exceeds a threshold value, the policy changes so that the nodes in the cycle are identical with respect to Module$_2$. This is accomplished by revoking Module$_2$’s privilege to write to Range$_1$ in the second state of the stateful policy.

In a stateful policy with $M$ modules that are receivers in a possible covert channel and $S$ states, the total number of states in this combined policy will be $O(S(2^M))$. If the stateful policy has $T$ transitions, the total number of transitions in the combined policy will be $O(T(M!M + 2^M))$. The cost of this privilege revocation mechanism should be balanced against the cost of ensuring that a module with a high security label is trusted. The cost of ensuring that a module is trusted is usually much higher than the price of building a mechanism. To ensure the greatest likelihood that critical services will be maintained, only those privileges that pertain to the covert channel should be revoked when a core causes
Chapter 5. Detecting Covert Channels in Stateful Policy Enforcement Systems

a counter to exceed its threshold. In other words, the revocation is performed on a per-channel basis.

To ensure that the combined policy does not introduce any new covert channels, the combined graph should be run through the detector. This extra step is just to be on the safe side, since building the combined policy should not introduce any new covert channels if done correctly. Adding the counters does not introduce any new cycles – it merely limits the number of times the existing cycles are allowed to go around. In addition, constructing the combined policy does not introduce any new cycles because the system never returns to an earlier policy following a policy transition. The detector should only identify in the combined policy those covert channels that we already know about – the exact same cycles that we are coping with by adding counters to them. The detector can be modified to check this list of known covert channels against the covert channels found in the combined policy so that it can report that no new covert channels were introduced in the construction of the combined policy.

5.6 Related Work

Policy engineering is an extremely important problem because an enforcement mechanism is only as good as the policy it enforces. Correctly designing a system
Figure 5.8: Coping with a covert channel in a stateful policy with two states. A counter measures the bandwidth of a covert channel from Module$_1$ to Module$_2$ by counting the number of times that a cycle occurs in the original policy on the left. If this counter exceeds a predetermined threshold, it is necessary to switch to the policy on the right, in which the covert channel has been eliminated by making the access matrices in both nodes of this policy identical with respect to Module$_2$. The total number of states in this combined policy is $S(2^M) = 4$, and the total number of transitions is $T(M!M + 2^M) = 6$, since $S$ is 2, $M$ is 1, and $T$ is 2.
that relies on a set of complex security policies calls for a new set of techniques to make it tractable for a human to correctly formulate policy specifications. Fong has developed a new approach to policy design by constraining the reference monitor to only track a “shallow execution history” of permitted resource access events [23]. Although this restriction limits the number of enforceable policies, many classic security policies can still be enforced. Breaking down the class of policies that can be enforced by an execution monitor into subclasses makes the problem of policy design more tractable because specialized policy languages and verification techniques can be tailored to these classes, and they are more easily decomposed into reusable components.

Since Lampson first introduced the concept of covert channels [53], several techniques for detecting covert channels in policy specifications have been proposed, including shared resource matrix methodology [47], information flow [66], and noninterference [32] [27], although this chapter focuses on the shared resource matrix method. Tsai et al. developed a static method of identification of covert storage channels in source code by using information flow analysis to identify kernel variables that are visible or can be altered [94]. They observe that not all potential covert channels can be exploited because the conditions that make the channel possible may not exist at runtime. They distinguish between potential covert channels and real covert channels.
Chapter 5. Detecting Covert Channels in Stateful Policy Enforcement Systems

There is much prior work in estimating the bandwidth of covert channels. Millen has devised a formula for calculating the bandwidth of a covert channel that is expressed in terms on the number of possible encodings of the data and the probability of each symbol [67] [68]. Shieh proposes a method of measuring the bandwidth of covert channels in multilevel operating systems [85]. He observes that resource-exhaustion channels can be modeled as finite-state graphs, but event-count channels cannot. Tsai and Gligor developed a Markov model to compute the maximum bandwidth of a covert storage channel under different system loads [93].

5.7 Summary

We have presented a novel method of dealing with the problem of covert channels in stateful policy enforcement systems that employ a reference monitor. We have considered the class of policy specifications that can be expressed in a regular language and therefore have a bounded number of states. A reference monitor is a runtime security primitive, and we have developed an automatic method of identifying security policies that could allow the reference monitor to be used as a covert channel. We have identified a range of corrective actions that can be considered once a possible covert channel is detected. The ideal alternative is
Chapter 5. Detecting Covert Channels in Stateful Policy Enforcement Systems

to eliminate the covert channel by changing the policy, but in case this option is not available, we have presented a method of coping with the covert channel by using a counter-based hardware primitive that measures the bandwidth of the covert channel. If the counter exceeds a predetermined threshold, further action is needed. A naive response is to terminate the core that is the receiver in the covert channel, but this option is highly problematical. We have presented an alternative to terminating cores that involves dynamically changing the policy when a counter exceeds its threshold by revoking the access rights of the receiver on a per-channel basis. Our technique revokes only those access rights that are needed to deal with the covert channel.

To achieve the greatest impact, new security techniques must be easily used by system designers, who usually are not computer security experts. This chapter attempts to make the job of designing policies easier for engineers by detecting possible covert channels so that they can be eliminated during the design phase. To make policy engineering easier and more precise, in Chapter 6 we present a higher-level language for expressing security policies in a more intuitive manner.
Chapter 6

Expressing Security Policies Precisely

In theory there is no difference between theory and practice. In practice there is.

Yogi Berra (1925-)

6.1 Introduction

Our approach to reconfigurable system security isolates cores with both static and runtime techniques that work together. A key element of our isolation strategy is a reference monitor that provides memory protection by enforcing a resource access policy expressed in a specialized language. Since a reference monitor is only as good as the policy it enforces, it is critical that the policy is constructed properly. The language that we developed in our earlier work [38] is fairly low-level, with many complicated regular expressions that are specified in terms of modules and ranges. We would like to make the embedded system designer’s
job easier by providing a higher-level language in which resource access policies are expressed in terms of higher-level security concepts. A compiler automatically translates the higher-level policy specification to the lower-level specification. This will reduce the possibility of human error in the building of policies, resulting in more accurate policies.

Research has shown that usability is critical to system security [30]. Security policies can be expressed at different layers of abstraction [89]. At the highest level is the organizational security policy [86], which is a document written in English that describes the security requirements of the organization. On the other hand, computer systems have mechanisms that enforce policies expressed at a much lower level of abstraction. Currently the faithful translation from a high-level organizational security policy to lower-level implementations is an open area of research.

We have developed a higher-level language and a compiler for translating this language into our earlier lower-level language. Policies expressed in our higher-level language are significantly less complicated than those expressed in our lower-level language, especially for stateful policies such as Chinese wall, water mark, and low water mark, which have exponential growth rates in the number of states. Expressing a high water mark or low water mark policy in our higher-level language only requires specifying the security labels of each module and range, and
expressing a Chinese wall policy only requires specifying the elements of each conflict-of-interest class. Our language currently supports ten different kinds of policies:

- Isolation
- Controlled sharing
- Access list
- Chinese wall
- Redaction
- Bell and LaPadula
- High water mark
- Biba
- Low water mark
- Dynamic policies

In addition to a higher-level language, we provide specialized tools to assist in the construction of mathematically precise security policies. In a correctly formed policy, the language of legal accesses and the language of illegal access should not intersect. An engineer constructing a policy can use a tool to check that a specific instance of illegal behavior does not intersect the policy of legal behavior under construction. If there is an overlap, the engineer is notified of the specific problem that must be fixed.
Chapter 6. Expressing Security Policies Precisely

6.2 A Higher-Level Language

6.2.1 Isolation

Isolation is a fixed (stateless) model in which each core is restricted to a fixed range (or set of ranges) of memory. Each range can only be assigned to one core. In our higher-level language, one specifies a set of compartments, and each compartment contains one or more modules and one or more ranges. A compartment with multiple modules is an equivalence class since the elements of the equivalence class are treated the same with respect to the policy. A module may only access the range(s) in its compartment:

\[
\begin{align*}
\text{Isolation;} \\
\text{Compartment}_1 & \rightarrow \text{Module}_1; \\
\text{Compartment}_1 & \rightarrow \text{Range}_1; \\
\text{Compartment}_2 & \rightarrow \text{Module}_2; \\
\text{Compartment}_2 & \rightarrow \text{Range}_2; \\
\text{Compartment}_2 & \rightarrow \text{Range}_3; \\
\text{Compartment}_3 & \rightarrow \text{Module}_3; \\
\text{Compartment}_3 & \rightarrow \text{Module}_4; \\
\text{Compartment}_3 & \rightarrow \text{Range}_4; \\
\text{Compartment}_3 & \rightarrow \text{Range}_5;
\end{align*}
\]

Our compiler translates the higher-level specification above to a lower-level specification:

\[
\begin{align*}
\text{Access}_0 & \rightarrow \{\text{Module}_1,r,\text{Range}_1\}; \\
\text{Access}_1 & \rightarrow \{\text{Module}_2,r,\text{Range}_2\} | \{\text{Module}_2,r,\text{Range}_3\}; \\
\text{Access}_2 & \rightarrow \{\text{Module}_3,r,\text{Range}_4\} | \{\text{Module}_3,r,\text{Range}_5\} | \{\text{Module}_4,r,\text{Range}_4\}
\end{align*}
\]
Chapter 6. Expressing Security Policies Precisely

Figure 6.1: An isolation policy.

\[
\text{Policy} \rightarrow (Access_0 | Access_1 | Access_2)^*;
\]

Figure 6.1 shows the resulting DFA for our isolation policy. Since the access matrix is drawn at each node, trivial transitions (from a state to itself) are not shown.

6.2.2 Controlled Sharing

Sometimes cores that are isolated need to communicate with each other. A controlled sharing policy allows the secure transfer of data from one core to another by synchronizing the transition of permissions during the exchange. In our higher-level language, one specifies the From module, the To module, and a Buffer range for making the exchange. We have implemented controlled sharing within the context of isolation, one also specifies the modules and ranges that belong to each compartment:
Chapter 6. Expressing Security Policies Precisely

Figure 6.2: A controlled sharing policy.

\[ init \]

\[ M1 M2 \]
R1: r_ __
R2: __ r_
R3: _w __
\[ \{M1,w,R3\} \]

\[ M1 M2 \]
R1: r_ __
R2: __ r_
R3: __ r_

Figure 6.2: A controlled sharing policy.

\[ CS; \]
From → Module1;
To → Module2;
Buffer → Range3;
Compartment1 → Module1;
Compartment1 → Range1;
Compartment2 → Module2;
Compartment2 → Range2;

Our compiler translates the higher-level specification above to a lower-level specification:

Access0 → \{Module1,r,Range1\};
Access1 → \{Module2,r,Range2\};
Access2 → (Access0 | Access1)*;
Access3 → (Access0 | Access1 | \{Module2,r,Range3\})*;
Trigger → \{Module1,w,Range3\};
Policy → (Access2*) (\epsilon | Trigger (Access3))*;

Figure 6.2 shows the DFA for our controlled sharing policy.
6.2.3 Access List

Sometimes a long list of subjects need to have access to the same object. Our access list policy is an isolation policy in which one or more modules belong to a list. The subjects in the policy are expressed in terms of lists rather than individual modules. Our access list policy is a mandatory rather than a discretionary access control policy. In our higher-level language, one first specifies the modules belonging to each list, and then expresses the isolation policy in terms of these lists:

\[
AL; \\
List_1 \rightarrow Module_1; \\
List_1 \rightarrow Module_2; \\
List_1 \rightarrow Module_3; \\
List_1 \rightarrow Module_4; \\
List_2 \rightarrow Module_3; \\
List_2 \rightarrow Module_4; \\
Compartment_1 \rightarrow List_1; \\
Compartment_1 \rightarrow Range_1; \\
Compartment_2 \rightarrow List_2; \\
Compartment_2 \rightarrow Range_2;
\]

Our compiler translates the higher-level specification above to a lower-level specification:

\[
Access_1 \rightarrow \{Module_1, r, Range_1\} | \{Module_2, r, Range_1\} | \{Module_3, r, Range_1\} \\
| \{Module_4, r, Range_1\}; \\
Access_2 \rightarrow \{Module_3, r, Range_2\} | \{Module_4, r, Range_2\}; \\
Policy \rightarrow (Access_1 | Access_2)^*;
\]

Figure 6.3 shows the DFA for our access list policy.
6.2.4 Chinese Wall

A Chinese wall policy [13] is expressed in our higher-level language by specifying the ranges that belong to each Conflict-of-Interest class as well as the module that is the subject in the policy:

\[
\begin{align*}
\text{Chinese;} \\
\text{Class}_1 & \rightarrow \text{Range}_1; \\
\text{Class}_2 & \rightarrow \text{Range}_2; \\
\text{Class}_2 & \rightarrow \text{Range}_3; \\
\text{Class}_2 & \rightarrow \text{Range}_4; \\
\text{Subject} & \rightarrow \text{Module}_1;
\end{align*}
\]

Our compiler translates the higher-level specification above to a lower-level specification:

\[
\begin{align*}
\text{Access}_0 & \rightarrow (\{\text{Module}_1, r, \text{Range}_1\} \mid \{\text{Module}_1, r, \text{Range}_3\})^*; \\
\text{Access}_1 & \rightarrow (\{\text{Module}_1, r, \text{Range}_1\} \mid \{\text{Module}_1, r, \text{Range}_4\})^*; \\
\text{Access}_2 & \rightarrow (\{\text{Module}_1, r, \text{Range}_1\} \mid \{\text{Module}_2, r, \text{Range}_3\})^*; \\
\text{Access}_3 & \rightarrow (\{\text{Module}_1, r, \text{Range}_1\} \mid \{\text{Module}_2, r, \text{Range}_4\})^*; \\
\text{Policy} & \rightarrow \text{Access}_0 \mid \text{Access}_1 \mid \text{Access}_2 \mid \text{Access}_3;
\end{align*}
\]

Figure 6.4 shows the DFA for our Chinese wall policy.
6.2.5 Redaction

We describe a redaction policy in our earlier work [38]. In our higher-level language, one specifies the liberal and restrictive policies, the trigger event, and the clear event:

Redaction:
Restrictive → \{\text{Module}_1, \text{rw}, \text{Range}_1\} | \{\text{Module}_1, r, \text{Range}_3\} | \{\text{Module}_2, \text{rw}, \text{Range}_2\} | \{\text{Module}_2, w, \text{Range}_4\} | \{\text{Module}_3, \text{rw}, \text{Range}_3\};
Liberal → Restrictive | \{\text{Module}_2, r, \text{Range}_3\};
Trigger → \{\text{Module}_1, w, \text{Range}_4\};
Clear → \{\text{Module}_3, z, \text{Range}_3\};

Our compiler translates the higher-level specification above to a lower-level specification:
Figure 6.5: A redaction policy.

\[
\begin{align*}
Access_1 & \rightarrow \{\text{Module}_1, rw, Range_1\} \mid \{\text{Module}_1, r, Range_3\} \mid \{\text{Module}_2, rw, Range_2\} \\
Access_2 & \rightarrow Access_1 \mid \{\text{Module}_2, r, Range_3\}; \\
\text{Trigger} & \rightarrow \{\text{Module}_1, w, Range_4\}; \\
\text{Clear} & \rightarrow \{\text{Module}_3, z, Range_3\}; \\
\text{SteadyState} & \rightarrow (Access_1 \mid \text{Clear Access}_2^* \text{Trigger})^*; \\
\text{Policy} & \rightarrow \epsilon \mid Access_2^* \mid Access_2^* \text{Trigger SteadyState} \\
& \quad \mid Access_2^* \text{Trigger SteadyState Clear Access}_2^*;
\end{align*}
\]

Figure 6.5 shows the DFA for our redaction policy.

6.2.6 Bell and LaPadula Confidentiality Model

The Bell and LaPadula (B&L) Model is a formal model of multilevel security in which a subject may not read an object with a higher security label (no read-up), and a subject may not write to an object with a lower security label (no
write-down) [6]. This model is designed to protect the confidentiality of classified information. One express a B&L policy by specifying the security labels of each module and range:

\[
\begin{align*}
B & \& L; \quad Module_1 \rightarrow TS; \\
& \quad Module_2 \rightarrow U; \\
& \quad Range_1 \rightarrow S; \\
& \quad Range_2 \rightarrow U;
\end{align*}
\]

Our compiler translates the higher-level specification above to a lower-level specification:

\[
Policy \rightarrow (\{Module_1, r, Range_1\} \mid \{Module_1, r, Range_2\} \mid \{Module_2, w, Range_1\} \\
\mid \{Module_2, r, w, Range_2\})^*;
\]

Figure 6.6 shows the DFA for our B&L policy.
6.2.7 High Water Mark

The high water mark model is an extension to B&L. High water mark is identical to B&L in that no read-up is permitted, but write-down is allowed in high water mark. Following a write-down, the security label of the object written to must change to the label of the subject that performed the write. Unlike B&L, high water mark policies are stateful. One expresses a high water mark policy by specifying the security labels of each module and range:

\[ \text{High}; \]
\[ \text{Module}_1 \rightarrow TS; \]
\[ \text{Module}_2 \rightarrow U; \]
\[ \text{Range}_1 \rightarrow S; \]
\[ \text{Range}_2 \rightarrow U; \]

Our compiler translates the higher-level specification above to a lower-level specification:

\[ \text{Trigger}_1 \rightarrow \{ \text{Module}_1, w, \text{Range}_1 \}; \]
\[ \text{Trigger}_2 \rightarrow \{ \text{Module}_1, w, \text{Range}_2 \}; \]
\[ \text{Access}_0 \rightarrow (\{ \text{Module}_1, r, \text{Range}_1 \} | \{ \text{Module}_1, r, \text{Range}_2 \} | \{ \text{Module}_2, w, \text{Range}_1 \}
  | \{ \text{Module}_2, w, \text{Range}_2 \}\}^*; \]
\[ \text{Access}_1 \rightarrow (\{ \text{Module}_1, rw, \text{Range}_1 \} | \{ \text{Module}_1, r, \text{Range}_2 \} | \{ \text{Module}_2, w, \text{Range}_1 \}
  | \{ \text{Module}_2, w, \text{Range}_2 \}\}^*; \]
\[ \text{Access}_{12} \rightarrow (\{ \text{Module}_1, rw, \text{Range}_1 \} | \{ \text{Module}_1, rw, \text{Range}_2 \} | \{ \text{Module}_2, w, \text{Range}_1 \}
  | \{ \text{Module}_2, w, \text{Range}_2 \}\}^*; \]
\[ \text{Access}_2 \rightarrow (\{ \text{Module}_1, r, \text{Range}_1 \} | \{ \text{Module}_1, rw, \text{Range}_2 \} | \{ \text{Module}_2, w, \text{Range}_1 \}
  | \{ \text{Module}_2, w, \text{Range}_2 \}\}^*; \]
\[ \text{Access}_{21} \rightarrow (\{ \text{Module}_1, rw, \text{Range}_1 \} | \{ \text{Module}_1, rw, \text{Range}_2 \} | \{ \text{Module}_1, w, \text{Range}_2 \}
  | \{ \text{Module}_2, rw, \text{Range}_2 \}\}^*; \]
\[ \text{Path}_1 \rightarrow (\epsilon | \text{Trigger}_1 \text{Access}_1^* (\epsilon | \text{Trigger}_2 \text{Access}_{12}^*)); \]
Chapter 6. Expressing Security Policies Precisely

![Diagram of a high water mark policy]

Figure 6.7: A high water mark policy.

Path\(_2\) → (\(\epsilon \mid Trigger_2\) Access\(_2^*\) (\(\epsilon \mid Trigger_1\) Access\(_{21}^*\)));
Policy → Access\(_0^*\) (\(\epsilon \mid Path_1 \mid Path_2\));

Figure 6.7 shows the DFA for our high water mark policy.

6.2.8 Biba Integrity Model

The Biba model is the dual of the Bell-LaPadula model [8]. Since Biba is designed to protect the integrity of classified data, both read-down and write-up are not permitted in Biba. One expresses a Biba policy in our higher-level language by specifying the security labels of each module and range:

Biba;
Module\(_1\) → TS;
Module\(_2\) → U;
Range\(_1\) → S;
Range\(_2\) → U;
Chapter 6. Expressing Security Policies Precisely

![Figure 6.8: A biba policy.](image)

Our compiler translates the higher-level specification above to a lower-level specification:

\[
Policy \rightarrow (\{\text{Module}_1, w, \text{Range}_1\} \mid \{\text{Module}_1, w, \text{Range}_2\} \mid \{\text{Module}_2, r, \text{Range}_1\} \mid \{\text{Module}_2, rw, \text{Range}_2\})^*;
\]

Figure 6.8 shows the DFA for our Biba policy.

6.2.9 Low Water Mark

Just as the high water mark model is an extension to B&L, the low water mark is an extension to Biba. Low water mark is identical to Biba in that write-up is prohibited, but read-down is allowed in low water mark. Following a read-down, the security label of the subject that performed the read-down must change to the label of the object that was read. One expresses a low water mark policy in our higher-level language by specifying the security labels of each module and range:
Chapter 6. Expressing Security Policies Precisely

Low;
Module_1 \rightarrow TS;
Module_2 \rightarrow U;
Range_1 \rightarrow S;
Range_2 \rightarrow U;

Our compiler translates the higher-level specification above to a lower-level specification:

\begin{align*}
Trigger_1 & \rightarrow \{\text{Module}_1, r, \text{Range}_1\}; \\
Trigger_2 & \rightarrow \{\text{Module}_2, r, \text{Range}_2\}; \\
Access_0 & \rightarrow (\{\text{Module}_1, w, \text{Range}_1\} \mid \{\text{Module}_1, w, \text{Range}_2\} \mid \{\text{Module}_2, r, \text{Range}_1\} \\
& \mid \{\text{Module}_2, rw, \text{Range}_2\}\}; \\
Access_1 & \rightarrow (\{\text{Module}_1, rw, \text{Range}_1\} \mid \{\text{Module}_2, r, \text{Range}_1\} \mid \{\text{Module}_1, w, \text{Range}_2\} \\
& \mid \{\text{Module}_2, rw, \text{Range}_2\}\}; \\
Access_{12} & \rightarrow (\{\text{Module}_1, r, \text{Range}_1\} \mid \{\text{Module}_2, r, \text{Range}_1\} \mid \{\text{Module}_1, rw, \text{Range}_2\} \\
& \mid \{\text{Module}_2, rw, \text{Range}_2\}\}; \\
Access_2 & \rightarrow (\{\text{Module}_1, r, \text{Range}_1\} \mid \{\text{Module}_2, r, \text{Range}_1\} \mid \{\text{Module}_1, rw, \text{Range}_2\} \\
& \mid \{\text{Module}_2, rw, \text{Range}_2\}\}; \\
Access_{21} & \rightarrow (\{\text{Module}_1, r, \text{Range}_1\} \mid \{\text{Module}_2, r, \text{Range}_1\} \mid \{\text{Module}_1, rw, \text{Range}_2\} \\
& \mid \{\text{Module}_2, rw, \text{Range}_2\}\}; \\
Path_1 & \rightarrow (\epsilon \mid Trigger_1 Access_1* (\epsilon \mid Trigger_2 Access_{12}*) ); \\
Path_2 & \rightarrow (\epsilon \mid Trigger_2 Access_2* (\epsilon \mid Trigger_1 Access_{21}*) ); \\
Policy & \rightarrow Access_0* (\epsilon \mid Path_1 | Path_2); \\
\end{align*}

Figure 6.9 shows the DFA for our low water mark policy.

6.2.10 Dynamic Policies

The ability to change the policies in response to external events is useful. For example, if the system comes under attack, it may be necessary to change to a more restrictive policy. One expresses a dynamic policy in our higher-level
Chapter 6. Expressing Security Policies Precisely

![Diagram showing a low water mark policy.]

**Figure 6.9:** A low water mark policy.

...language by specifying the policies to switch between as well as the trigger events that cause a policy change:

\[
\begin{align*}
    Dynamic; \\
    Trigger_1 & \rightarrow \{Module_1,w,Range_8\}; \\
    Trigger_2 & \rightarrow \{Module_1,w,Range_9\}; \\
    Policy_1 & \rightarrow isolation; \\
    Policy_2 & \rightarrow biba; \\
    Policy_3 & \rightarrow cs;
\end{align*}
\]

*Policy*$_1$, *Policy*$_2$, and *Policy*$_3$ can be any three policies. If the policies come from different sources, pre-processing can be used to prevent naming conflicts (e.g., if two policies define *Access*$_1$ differently). Trigger events specify the circumstances under which a policy change can occur. *Trigger*$_1$ causes the policy to change from *Policy*$_1$ to *Policy*$_2$, and *Trigger*$_2$ causes the policy to change from *Policy*$_2$ to *Policy*$_3$. In this example, it is not possible to return to an earlier policy.
Figure 6.10: A dynamic policy in which returning to an earlier policy is not allowed.

We arbitrarily chose isolation to be Policy₁, Biba to be Policy₂, and controlled sharing to be Policy₃. Understanding the organizational requirements for dynamic security policies is the topic of related research [12] [24]. Our compiler translates the higher-level specification above to a lower-level specification:

\[ \text{Policy} \rightarrow (\text{Policy}_1) (\epsilon | \text{Trigger}_1 (\text{Policy}_2) (\epsilon | \text{Trigger}_2 (\text{Policy}_3))) \]

Figure 6.10 shows the DFA for our dynamic policy in which returning to an earlier policy is not permitted.
The ability to return to an earlier policy is problematic when stateful policies are involved because it is necessary to remember the state of the earlier policy. In the following example of a dynamic policy, returning to an earlier policy is permitted, but one is restricted to switching between two stateless policies. In our higher-level language, one expresses such a dynamic policy by specifying the two stateless policies as well as the trigger events that cause a policy change:

```
Dynamic2;
Trigger1 → \{Module1,w,Range8\};
Trigger2 → \{Module1,w,Range9\};
Policy1 → b&l;
Policy2 → biba;
```

`Trigger1` causes the policy to change from `Policy1` to `Policy2`, and `Trigger2` causes the policy to change from `Policy2` back to `Policy1`. Our compiler translates the higher-level specification above to a lower-level specification:

```
SteadyState → (Policy2 | Trigger2 Policy1 Trigger1)*;
Policy → ε | Policy1 | Policy1 Trigger1 SteadyState
             | Policy1 Trigger1 SteadyState Trigger2 Policy1;
```

Figure 6.11 shows the DFA for our dynamic policy in which returning to an earlier policy is allowed.
6.3 Installing and Using the Policy Compiler

6.3.1 Installation Instructions

Our policy compiler is for academic use only. You will need the Java SDK, gcc, and dot (GraphViz). After extracting the archive, build the parser:

```
% make
```

Next, compile the Java files:

```
% wget java.sun.com/developer/technicalArticles/Programming/sprintf/PrintfFormat.java
% javac *.java
```
Next, build RegEx, an implementation of Thompson’s Algorithm and subset construction by Gerzic [25], which we have modified to output state machines in a format that is compatible with Grail:

% g++ -o regex AG_RegEx.cpp

Finally, build Grail, a symbolic computation environment for finite-state machines, regular expressions, and finite language [74]:

% cd grail/longlong
% g++ -o ../../longlong.out grail.cpp
% cd ../..

6.3.2 Using the Policy Compiler

To illustrate the use of our compiler, we will try a simple example isolation policy. Create a file *toy.policy* with the following contents:

```
Isolation;
Compartment1->Module1;
Compartment1->Range1;
Compartment2->Module2;
Compartment2->Range2;
```

**Specifying the ranges** — Ranges are specified in the file *ranges*, and an example is provided in the archive. The first line of the file specifies $Range_1$, the second line of the file specifies $Range_2$, and so on. Each line of the file has the starting and ending address of the range separated by a space. Each range must
be an aligned power of two range. We are now ready to use the compiler on this policy:

```
% ./run.sh toy
not found
0 is a start state
There are 1 unique states
This graph does NOT contain a cycle.
```

If the policy compiler is installed correctly, the script `run.sh` performs the following steps:

- `run.sh` processes the ranges.
- `run.sh` creates a file `toy.p` (the lower-level policy specification) from `toy.policy` (the higher-level policy specification).
- `run.sh` runs `toy.p` through the parser.
- The resulting regular expression is fed as input to regex, which creates a file `grail_machine`, which is a DFA expressed in Grail format.
- `run.sh` runs `grail_machine` through Grail to produce a file `gm_toy`, which is the minimized DFA.
- `run.sh` creates from `gm_toy` a file `toy.v`, which is a Verilog HDL description of the reference monitor that enforces the policy.
- `run.sh` checks to see whether the DFA has any possible covert channels.
• *run.sh* creates a file *toy.dot*, which expresses the DFA in Graphviz format.

• *run.sh* runs *toy.dot* through Graphviz, resulting in a file *toy.ps*, which is a PostScript version of the DFA that can be printed or displayed on the screen. Figure 6.12 shows this graph.

Let’s take a look at the file *toy.p*, which is the lower-level policy specification generated from *toy.policy*, the higher-level policy specification:

```
Access0->Module1ReadsRange1;
Access1->Module2ReadsRange2;
Policy->(Access0|Access1)*;
```

Now, let’s examine the file *toy.v*, which is the Verilog HDL description of the reference monitor that enforces *toy.policy*:

```verilog
module State_Machine(clock, reset, module_id, op, address, is_legal);
    input clock, reset;
    input [4:0] module_id;
    input [1:0] op;
    input [31:0] address;
    output is_legal;
    reg is_legal;
    reg[0:0] state;
    parameter s0 = ’d0;
    parameter s1 = ’d1;
    wire r0;
    wire r1;
    assign r0=(address[31:5]==27’d142395)?1’b1:1’b0;
    assign r1=(address[31:7]==25’d35599)?1’b1:1’b0;
    always @(state)
        begin
            case (state)
                s0:
```
Chapter 6. Expressing Security Policies Precisely

```verilog
is_legal=1'b1;
s1:
    is_legal = 1'b0;
default:
    is_legal = 1'b0;
endcase
end
always @(posedge clock or posedge reset)
if (reset) state = s0;
else
    case (state)
s0:
        case({module_id,op,r0,r1})
            9'b000100101: //2 1 1
                state = s0;
            9'b000010110: //1 1 0
                state = s0;
            default:
                state = s1;
        endcase
        state = s1;
s1:
        state = s1;
default:
        state = s1;
endcase
endmodule
```

The reference monitor expressed in the above Verilog code has three inputs (module_id, op, and address) and one output (is_legal). It uses exactly one cycle to make a decision as to the legality of the requested memory access according to the policy. The assign statements check the ranges in parallel to determine the range of address. The expression `{module_id,op,r0,r1}` concatenates module_id, op, r0, and r1 to form a single transition symbol. The first case statement determines
the value of islegal according to whether the state is accepting or rejecting. The second case statement determines the next state according to the current state and the transition character.

6.4 Incremental Construction of Mathematically Precise Policies

In order for a policy to be precise, it must accept all behavior which is legal and reject all behavior which is illegal. Constructing policies can be challenging without an automatic way of verifying that the policy reflects the intent of the person creating that policy. Our methods make it possible to determine if there is any conflict between behavior that should be legal and behavior that should be illegal.
6.4.1 Theoretical Foundations

In order for a policy to be correct, there must be no behavior that is recognized as both legal and illegal. In other words, the intersection between the language of legal accesses and the language of illegal accesses must be the empty set. If the language of legal behavior and the language of illegal behavior intersect, the person constructing the policy must be notified of the offending overlapping behavior. We can easily determine the intersection of two languages by computing the cross product of their corresponding state machines, a process which requires quadratic time. Figure 6.13 illustrates this concept.

Figure 6.14 shows our incremental approach of constructing policies. A “rough draft” of a policy of legal accesses is tested for correctness by checking whether specific instances of known illegal behavior overlap with the legal policy. Since this process is automated, the system can test a very large set of known illegal behavior and notify the user of any behavior that is known to be illegal but is recognized as legal.

6.4.2 A Simple Example

Consider a language of legal behavior $L_{Legal} = (A|B|C)^*$ over the alphabet $A, B, C, D, E$. Figure 6.15 shows the DFA that accepts $L_{Legal}$. Suppose also that we have a language of illegal behavior $L_{Illegal} = (C|D|E)^*$. Figure 6.16 shows that
DFA that accepts $L_{\text{Illegal}}$. Figure 6.17 shows the DFA that accepts $L_{\text{Legal}} \times L_{\text{Illegal}}$, which is $C^*$. 

### 6.4.3 Example: Chinese Wall

We now apply our technique of ensuring policy correctness to a more complex, stateful security scenario. In our earlier work [38], we described a Chinese Wall policy, and we showed the language of legal accesses. A reference monitor that rejects illegal behavior is equivalent to one that accepts legal behavior.

The following is the language of illegal behavior for our Chinese wall policy:

\[
\begin{align*}
\text{Access}_1 & \rightarrow \{\text{Module}_1, r, \text{Range}_1\}; \\
\text{Access}_2 & \rightarrow \{\text{Module}_1, r, \text{Range}_2\}; \\
\text{Access}_3 & \rightarrow \{\text{Module}_1, r, \text{Range}_3\}; \\
\text{Access}_4 & \rightarrow \{\text{Module}_1, r, \text{Range}_4\}; \\
\text{Anything} & \rightarrow (\text{Access}_1 \mid \text{Access}_2 \mid \text{Access}_3 \mid \text{Access}_4 \mid \epsilon)^*; \\
\text{Access}_5 & \rightarrow \text{Anything } \text{Access}_1 \text{ Anything } \text{Access}_2 \text{ Anything}; \\
\text{Access}_6 & \rightarrow \text{Anything } \text{Access}_2 \text{ Anything } \text{Access}_1 \text{ Anything}; \\
\text{Access}_7 & \rightarrow \text{Anything } \text{Access}_3 \text{ Anything } \text{Access}_4 \text{ Anything}; \\
\text{Access}_8 & \rightarrow \text{Anything } \text{Access}_4 \text{ Anything } \text{Access}_3 \text{ Anything}; \\
\text{Illegal} & \rightarrow \text{Access}_5 \mid \text{Access}_6 \mid \text{Access}_7 \mid \text{Access}_8; \\
\text{Policy} & \rightarrow \text{Illegal}.
\end{align*}
\]

Figure 6.18 shows the DFA that recognizes legal accesses for the Chinese Wall policy, and Figure 6.19 shows the DFA that recognizes illegal accesses. Since both policies have been correctly constructed, there is no intersection between the language of legal access and the language of illegal accesses. We have verified this by taking the cross product of the two policies:
Figure 6.13: A Venn Diagram that illustrates the logic behind our scheme. In a correct policy, there should be no intersection between legal and illegal accesses.

Both `chinese.policy` and `chinese2.p` are distributed with our compiler. The file `chinese.policy` contains the higher-level Chinese wall policy specification (legal behavior). The file `chinese2.p` contains the lower-level policy of illegal behavior for the Chinese wall policy. There should be no output from the last command (`./fmcross...`), indicating that the intersection of legal behavior and illegal behavior is the empty set.
Figure 6.14: An automated approach to the incremental construction of policies. Several examples of known illegal behavior can be automatically checked against a “rough draft” policy of legal accesses to determine if there is any intersection.

6.4.4 Monotonic Policy Changes

The ability to determine the intersection of two policies is also useful for dynamic policies. In a system with the ability to switch policies dynamically, suppose that the system changes from a more restrictive policy to a less restrictive policy. In this situation, a core could retain sensitive information in its local memory after the new policy becomes effective. Access to this data was legal under the old policy, but becomes prohibited under the new policy. One way of dealing with this problem is to sanitize all of the cores in the system following a policy change, but this approach has several drawbacks including the possible disruption
Chapter 6. Expressing Security Policies Precisely

Figure 6.15: DFA that recognizes the language \((A|B|C)^*\). An input of either \(D\) or \(E\) causes this DFA to transition to the rejecting state (State 1).

Figure 6.16: DFA that recognizes the language \((C|D|E)^*\). An input of either \(A\) or \(B\) causes this DFA to transition to the rejecting state (State 1).

Figure 6.17: DFA that recognizes the language \(C^*\).

Figure 6.18: DFA that recognizes legal accesses for a Chinese Wall policy.
Chapter 6. Expressing Security Policies Precisely

Figure 6.19: DFA that recognizes illegal accesses for a Chinese wall policy.

of critical services. Another solution is to always change to a more restrictive policy. In a system that only allows changes to monotonically more restrictive policies, each policy is a subset of the previous policy. In other words, the intersection of \( \text{Policy}_i \) and \( \text{Policy}_{i+1} \) is identical to \( \text{Policy}_{i+1} \). Suppose that a set of policies \( \{ \text{Policy}_1, \text{Policy}_2, \text{Policy}_3, ..., \text{Policy}_N \} \) is available in a dynamic policy system. To determine which policy changes are legal, one takes the intersection of every \( (\text{Policy}_i, \text{Policy}_j) \) pair and checks if the result is identical to \( \text{Policy}_j \). If it is, then a change from \( \text{Policy}_i \) to \( \text{Policy}_j \) is monotonically more restrictive and therefore legal.
6.5 Summary

We rely on embedded devices to do more and more computing for us. These devices have become quite sophisticated, with multiple functions converging onto a single device. Reconfigurable hardware is at the heart of many high-performance embedded systems, and the embedded systems community requires novel security primitives which address the realities of modern reconfigurable hardware. We have developed a security primitive that uses a reconfigurable reference monitor to provide memory protection for FPGA systems. This reference monitor enforces a memory access policy expressed in a specialized language, and a compiler translates the policy specification directly to a circuit. Unfortunately, a reference monitor is only as good as the policy it enforces. In this chapter, we have presented a higher-level language for expressing security policy specifications in terms of higher-level security concepts. This reduces the chances that an embedded systems designer will make an error when constructing a policy. Many embedded designers are not experts in security, and tools are needed to assist them in developing secure hardware systems. Our language makes it much simpler to express policies than the language we originally developed in our earlier work. In addition to the higher-level language, we have developed a set of tools to assist in the construction of mathematically precise security policies. These tools
can determine the intersection of two policies. Since there should be no overlap between legal and illegal behavior in a properly constructed policy, this ability can be exploited to verify that a “rough draft” of a policy under construction does not overlap known instances of illegal behavior. In addition, the ability to take the intersection of two policies is useful in ensuring that policy changes are monotonic in dynamic policy systems.
Chapter 7

Conclusions and Future Work

Of every 100 soldiers, 10 do not belong there and should be sent home. 80 are just targets. Nine are the true warriors, and we are glad to have them, for they make the battle. But one, he is the leader, and he brings the rest home.

Heraclitus (540 B.C. - 480 B.C.)

Embedded devices perform a critical role in both the commercial and military sectors. More and more functionality is being packed onto a single device in order to realize the cost savings of increased integration. Since reconfigurable hardware is at the heart of many of these embedded devices, new efficient security primitives are needed. Since reconfigurable systems can have multiple cores on the same FPGA operating at different trust levels, we have presented a set of primitives, both static and runtime, that work together to separate cores so that they do not interfere with each other. At the root of our hierarchy of primitives is a reconfigurable reference monitor that enforces a memory access policy that specifies the legal sharing of memory among cores. A compiler translates this
policy specification to a hardware description of a circuit that can be directly loaded onto the FPGA. Additional primitives are needed to make this reference monitor strategy a success. The reference monitor must not be tampered with or bypassed, and our moat/drawbridge primitive logically isolates cores as well as the reference monitor. The reference monitor must not be used as a covert channel, and to address this problem we have presented a technique for analyzing stateful policies to detect possible covert channels. Since the reference monitor is only as good as the policy it enforces, it is important that embedded designers, who are not necessarily computer security experts, be able to express security policies precisely. Our solution to this problem is a higher-level language as well as a set of tools that use formal methods to ensure that the construction of policies is mathematically precise.

We see many opportunities for future work in this area. As the underlying fabric of computing changes from a general-purpose uni-processor model with virtual memory and disk to a model in which embedded devices such as cell phones perform more and more of the world’s computing tasks, a new approach to system development is needed. System design under this new model will require changes to the way in which software is developed in order to ensure performance, correctness, and security. Since embedded devices have constrained resources such as processor speed and power, application developers will need to exploit the
Chapter 7. Conclusions and Future Work

performance that comes from the parallelism of raw hardware. Future systems will either be chip multi-processor systems running multiple threads, Systems-on-a-Chip with multiple special-purpose cores on a single ASIC chip, or a compromise between these two extremes on a reconfigurable device.

Most of the security primitives we have developed are not restricted to the reconfigurable domain, and we would like to apply them to chip multi-processor systems and Systems-on-a-Chip in the short-term. For example, our reference monitor would be immediately useful for the security of CMP systems by providing policy-driven separation of processor cores. Also in the short-term, we would like to investigate the use of security policies to drive the floor planning of FPGA systems. For example, cores that handle highly sensitive data should be placed together, and they should be isolated from less sensitive cores. In addition, we would like to explore the use of a secure form of virtual memory [9] to provide resource arbitration.

In the short to intermediate term, we would like to study the problem of adaptive computing on reconfigurable devices. The partial reconfiguration capability of the latest FPGA devices makes a new form of computing possible. A system with multiple reconfigurable cores will dynamically profile itself to determine whether it would be helpful to swap one core for another. For example, if the system detects that a cryptographic operation is being performed, more hardware area
could be allocated to this task. Analysis of the security implications of such a computing model will be worthwhile, and new security primitives will be needed.

A reference monitor approach does not guarantee the end-to-end security of information in the system. In the intermediate to long term, we would like to study the possibility of incorporating information flow into a secure FPGA architecture, since the importance of FPGAs is growing. Such an architecture would require the ability to bind a tag, or security label, to individual units of data. The study of tagged architectures for FPGAs is an active area of research. Hardware mechanisms distributed throughout the chip would need to check these tags to enforce an information flow policy.

The current state of embedded systems security leaves much to be desired. More spending on computer security in general has not resulted in fewer attacks. Designing complex systems that are trustworthy is challenging, and a holistic approach to system design is needed. Developments from the field of computer security research need to be adopted by the mainstream embedded design community, and new techniques are needed to manage security in FPGA designs.
Bibliography


Bibliography


