Enforcing Memory Policy Specifications in Reconfigurable Hardware

Ted Huffmire*, Timothy Sherwood†, Ryan Kastner‡, and Timothy Levin*

* Naval Postgraduate School
Department of Computer Science
Monterey, CA 93943
{tdhuffmi,televin}@nps.edu

† University of California, Santa Barbara
Department of Computer Science
Santa Barbara, CA 93106
sherwood@cs.ucsb.edu

‡ University of California, San Diego
Department of Computer Science and Engineering
La Jolla, CA 92093
kastner@ucsd.edu

Abstract

While general-purpose processor based systems are built to enforce memory protection to prevent the unintended sharing of data between processes, current systems built around reconfigurable hardware typically offer no such protection. Several reconfigurable cores are often integrated onto a single chip where they share external resources such as memory. 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 one another. We investigate the design and synthesis of a FPGA memory protection mechanism capable of enforcing access control policies and a methodology for translating formal policy descriptions into FPGA enforcement mechanisms. The efficiency of our access language design flow is evaluated in terms of area and cycle time across a variety of security scenarios. We also describe a technique for ensuring that the internal state of the reference monitor cannot be used as a covert storage channel.

Keywords: Reconfigurable hardware, Protection mechanisms, Security and Privacy Protection, Access controls

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 fraction 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.

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, integrated 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 possibility 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 paper 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 paper, 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 automaticaly 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 paper 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 paper we strive to strike a balance between engineering and formal evaluation; between performance, security, and clarity. Specifically, this paper makes 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.
- 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.

In this article, we extend our preliminary work [27] to incorporate:

- A more thorough discussion of the architecture of reconfigurable systems
- A motivating example of a reconfigurable system from the field of computer vision
- Additional example policies and synthesis results, including B&L, Biba, high water mark, and dynamic policies
- A description of a technique to prevent the internal state of the reference monitor from being used as a covert storage channel.
- Substantial revisions and corrections throughout the paper

The remainder of the paper is organized as follows: Section 2 provides background on FPGAs and describes the threat model that we are addressing. In Section 3, we explain the algorithms behind our reference monitor design flow. In Section 4, we describe our access policy language including several example policies. We present our reference monitor synthesis results in Section 5. We describe our technique for preventing the reference monitor from being used as a covert storage channel in Section 6. Finally, we conclude in Section 7 and discuss where there is room for future work.

2 Reconfigurable Systems

Increasingly we are seeing reconfigurable devices emerge as the flexible and high-performance workhorses inside a variety of high performance embedded computing systems [7, 11, 14, 31, 43, 54]. 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 [14]. 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 [10, 13, 63]. 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 [36]. 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.

2.1 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* and *or* gates. Figure 1 shows a modern FPGA-based embedded system.

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 aimed at optimizing physical aspects of the FPGA architecture [5]. 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.

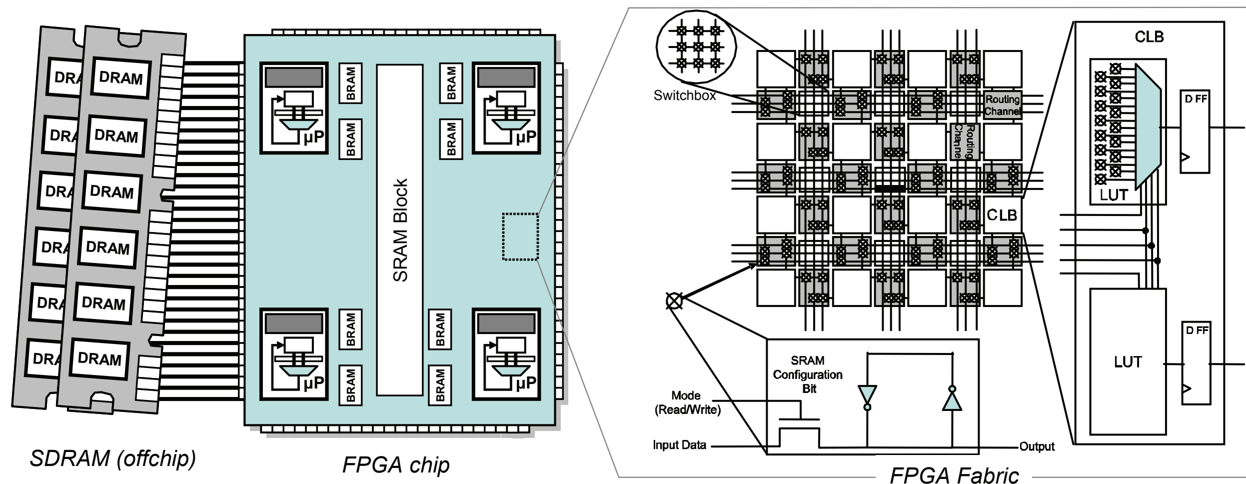

Figure 1: A Modern FPGA-based Embedded System: Reconfigurable logic, blocks of SRAM, and hard-wired microprocessors all share the same piece of silicon, and, more importantly, the same off-chip memory. The reconfigurable logic is a fabric of tiny lookup tables and statically scheduled routing hardware that can be configured to emulate almost any possible circuit.

2.1.1 Reconfigurable Devices and Security

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 [8, 33, 38] and to secure the FPGA's program logic update channels [25, 24]. However, few researchers have begun to consider the security ramifications of compromised hardware [22].

It is important to understand the different attacks against FPGAs that are possible in order to develop countermeasures [66]. 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 [59]. 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 [22]. Solutions to these problems include encryption [8] [32] [33], fingerprinting [37], and watermarking [38]. While there are a variety of attacks possible, our work is concerned with addressing the problem of memory protection on reconfigurable systems. In particular this paper is concerned with techniques to provide *separation* while allowing controlled interaction between multiple interacting cores and modules with respect to their use of off-chip memory[1]. 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.

[1]The same approach is applicable to on-chip memory, but we leave this to future work.

2.1.2 Protecting Memory on an FPGA

A secure run-time management system must protect different logical modules from interfering, intercepting, or corrupting any use of a shared resource without authorization. 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, the allocation and sharing of memory must be performed in a way that is efficient, flexible, and protected. On a general-purpose processor, interaction via shared memory can be controlled through the use of page table and associated TLB attributes. Use of Superpages, which are very large memory pages, makes it possible for the TLB to have a lower miss rate [48]. Segmented Memory [52] and Mondrian Memory Protection [65], a finer-grained scheme, address the inefficiency of providing per-process memory protection via global attributes by associating each process with distinct 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 by hardware mechanisms.**

Preventing unauthorized accesses to memory is fundamental to both effective debugging, error prevention, and computer security. However, memory management in software is complex and difficult: 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 approach to ensure that the security properties of the system are enforced.

To provide separation in memory between multiple interacting modules, we adapt some of the key concepts from separation kernels. Rushby originally proposed that a separation kernel [28] [41] [50] 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 separated one. A separation kernel partitions all resources under its control into blocks (subsets) 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 multilevel secure system, each block typically represents a different classification level, and the allowed communications conform to the MLS-label lattice [15].

We propose to treat the separate cores of the FPGA and related memory regions just as blocks of a separation kernel. The cores are isolated through a means we call "moats," and then we control interaction between cores in a highly assured manner. 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 [26].

2.2 Video Redaction: A Motivating Example

The purpose of redaction is to delete sensitive information from a document, audio recording, video feed, or other data stream. For example, a document containing sensitive information would need to have all top secret and secret data removed before a person with only a confidential clearance could read the document. In video redaction, the faces of people in video feeds from surveillance cameras are blurred if the person viewing the video does not have a high enough clearance level. IBM's PeopleVision project has developed such a video privacy system [56]. FPGAs are a natural choice for streaming applications because they can provide deep regular pipelines of computation, with no shortage of parallelism.

Consider how such a system might be developed. There would need to be at least three modules 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 enforcement module could prevent sensitive information from being shared between modules improperly (e.g. directly between the video interface and the network). In Section 4.4 we describe how such a situation might be handled using our methods.

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 mechanisms 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 [26].

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.

3.1 Memory Access Policy

Once a high level policy is developed based on the requirements of the system and the organizational security policy [60], it must be expressed in a precise form to allow engineers to build concrete enforcement mechanisms. In the context of this paper we concentrate on policies as they relate to memory accesses. In particular, the enforcement mechanisms we consider in this paper belong to the Execution Monitoring (EM) class [55], 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 [26].

Although Erlingsson et al. have proposed the idea of merging the reference monitor in-line with the target system [16], 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 [51], 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, 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 paper 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 ranges[2]).

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 [3]. 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 focussed on a fixed number of modules, ranges, and accesses. L needs to satisfy the property that $\forall x, t : tuple\ set \mid t \subseteq \Sigma, xt \subseteq L \rightarrow x \subseteq L$, 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
2. A method for automatically creating a circuit which recognizes memory access sequences that are legal under L
3. A method for preventing all accesses that are not legal under L

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:

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

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

[2] An interval of the address space including high (R_{high}) and low (R_{low}) bounds
[3] Details of revocation will be discussed in Section 4

$Access_2$ are simple access descriptors, but we want to allow more complex sets 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 [42], 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 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.

3.2 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:

- User creates the access policy (described above) and inputs it to the compiler, which:
- Builds a syntax tree from the policy.
- Transforms the syntax tree to an expanded intermediate form.
- Expands $Policy$ to a regular expression defined over the alphabet Σ.
- Converts the regular expression to a non-deterministic finite automaton (NFA).
- Constructs an equivalent minimized state machine from the NFA.
- Factors the ranges into sizes that are a power of two.
- Organizes the set of ranges as a trie[4], and creates a logic tree that recognizes them.
- Exports the state machine and range detection logic as Synthesizable Verilog.
- Inputs hardware description expressed in Verilog to Quartus software, which synthesizes, places, and routes circuit.
- Bit-stream loader loads the synthesized bit-stream onto the FPGA.

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:

$Access \rightarrow \{Module_1, rw, Range_1\} | \{Module_2, rw, Range_2\}$;
Policy$\rightarrow$($Access$)*;

[4]an ordered tree data structure for storing lookup tables

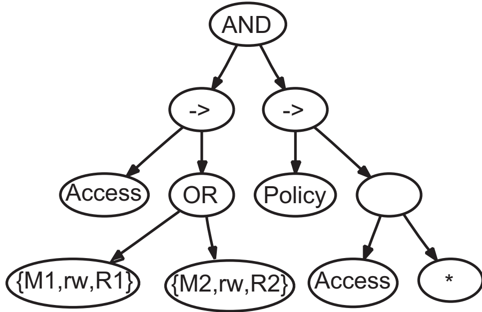

Figure 2: Parse tree of the simple access policy

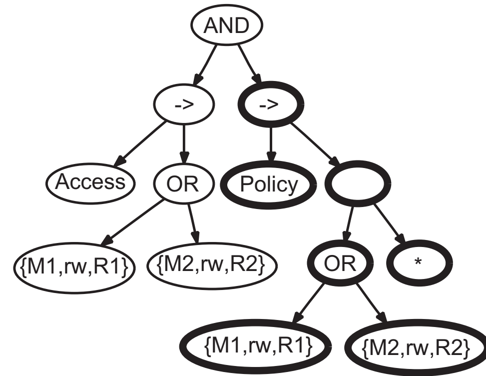

Figure 3: Expanded parse tree

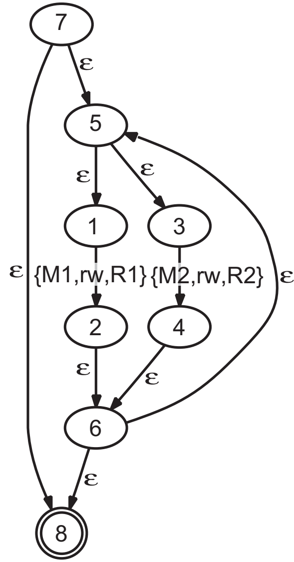

Figure 4: NFA derived from the regular expression

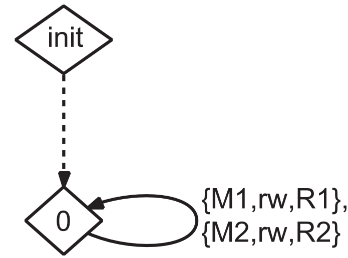

Figure 5: NFA converted to a minimized DFA

Building and Transforming a Parse Tree Next, we use Lex [40] and Yacc [30] to build a parse tree from our security policy. Internal nodes represent operators such as concatenation, alternation, and repetition. Figure 2 shows the parse tree for our example 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 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:

(($\{Module_1$,rw,$Range_1\}$) | ($\{Module_2$,rw,$Range_2\}$))*

Constructing the NFA Once the regular expression has been formed, we construct an NFA from this regular expression using Thompson's Algorithm [1] as implemented by Gerzic [19]. Figure 4 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.

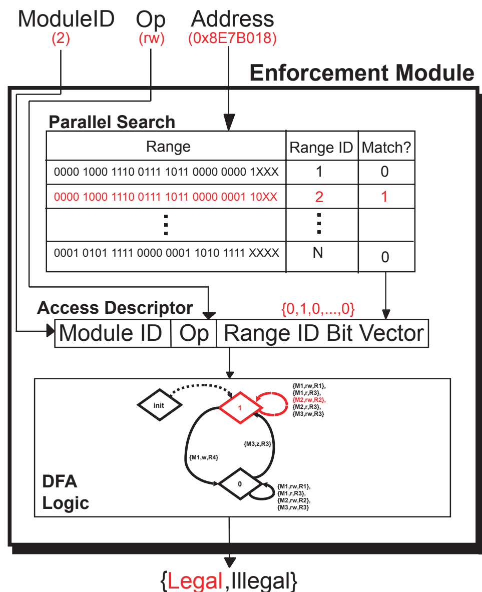

Figure 6: The inputs to the enforcement module are the module ID, op, and address. The range ID is determined by performing a parallel search over all ranges, similar to a content addressable memory (CAM). The module ID, op, and range ID together form an access descriptor, which is the input to the state machine logic. The output is a single bit: either grant or deny the access.

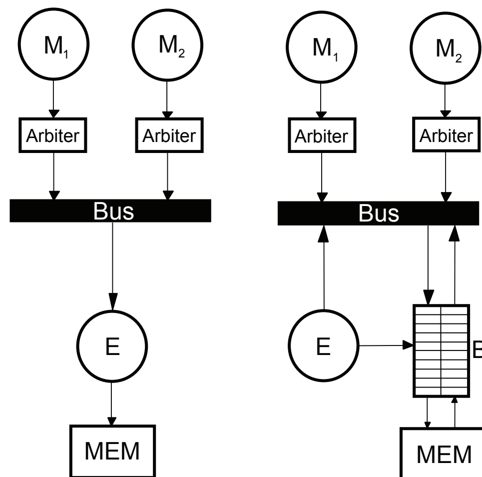

Figure 7: Two alternative architectures for the enforcement mechanism. In the figure on the left, a memory access must pass through the enforcement mechanism (E) before going to memory. In the figure on the right, the enforcement mechanism (E) snoops on the bus, and a buffer (B) prevents access to the data until the access is approved. Arbiters prevent the bus from being accessed by more than one module at a time.

Converting the NFA to a DFA From this NFA we can construct a DFA through subset construction [1] as implemented by Gerzic [19]. Following the creation of the DFA, we apply Hopcroft's Partitioning Algorithm [1] as implemented by Grail [49] to minimize the DFA. Figure 5 shows the minimized DFA for our policy.

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.[5] 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 $O(\log_2 |range|)$ 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 6 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, consisting of a module ID, an op, and a range ID bit vector. The range ID bit vector contains a bit for each possible range (currently a max of N ranges), and the descriptor's range is indicated by the (one) bit that is set.

[5] this is equivalent to performing a bit-wise XOR, masking the lower bits, and testing for non-zero except that in hardware the masking is unnecessary

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 outpus 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.

4 Example Applications

To further demonstrate the usefulness 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.

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:

$Class_1 \rightarrow Module_1 \mid Module_2$;
$Class_2 \rightarrow Module_3 \mid Module_4$;
$List_1 \rightarrow Class_1 \mid Class_2$;
$List_2 \rightarrow Class_2$;
$Access_1 \rightarrow \{List_1, rw, Range_1\}$;
$Access_2 \rightarrow \{List_2, rw, Range_2\}$;
$Policy \rightarrow (Access_1 \mid Access_2)^*$;

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.

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 $Module_1$ wants to securely transfer some data to $Module_2$. Rather than establishing a direct channel between $Module_1$ and $Module_2$, 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:

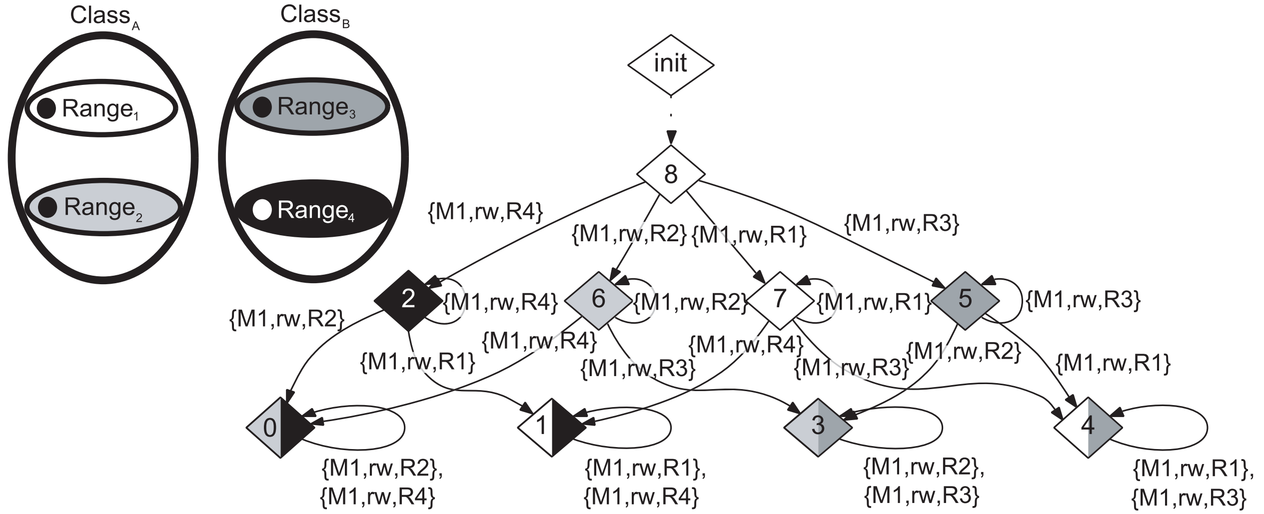

Figure 8: A Chinese wall policy. The Venn Diagram shows two conflict-of-interest classes, $Class_A$ and $Class_B$, and the DFA recognizes legal accesses for this Chinese Wall policy. A principal that accesses $Range_4$ (black) is subsequently prohibited from accessing $Range_3$ (dark gray), but it may access either *$Range_1$ (white) or $Range_2$ (light gray), because they are in a different class. An access to $Range_4$ results in a transition to state 2 (black), from which an access to $Range_1$ results in a transition to state 1 (black or white).*

$Module_{1|2} \rightarrow Module_1 \mid Module_2$;
$Access_1 \rightarrow \{Module_1, rw, Range_1\} \mid \{Module_{1|2}, rw, Range_2\}$;
$Access_2 \rightarrow \{Module_2, rw, (Range_1 \mid Range_2)\}$;
$Trigger \rightarrow \{Module_1, rw, Range_2\}$;
$Policy \rightarrow (Access_1)^* \, (\epsilon \mid Trigger \; (Access_2)^*)$;

Initially, $Module_1$ can access $Range_1$ and $Range_2$, and $Module_2$ can only access $Range_2$. However, the first time $Module_1$ accesses $Range_2$ (signaling to $Module_2$ that $Module_1$ is ready to exchange), $Access_1$ 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 $Range_2$.

4.3 Chinese Wall

Another security scenario that can be efficiently expressed using our policy language is the Chinese wall [9]. 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 8 shows a Venn Diagram for this situation. We express a Chinese wall policy below, where $Module_1$ corresponds to the lawyer and each range corresponds to a company:

$Access_1 \rightarrow \{Module_1, rw, (Range_1 \mid Range_3)\}^*$;
$Access_2 \rightarrow \{Module_1, rw, (Range_1 \mid Range_4)\}^*$;
$Access_3 \rightarrow \{Module_1, rw, (Range_2 \mid Range_3)\}^*$;
$Access_4 \rightarrow \{Module_1, rw, (Range_2 \mid Range_4)\}^*$;
$Policy \rightarrow Access_1 \mid Access_2 \mid Access_3 \mid Access_4$;

In our Chinese wall policy, there are two COI classes. One contains $Range_1$ and $Range_2$, and the other contains $Range_3$ and $Range_4$. 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

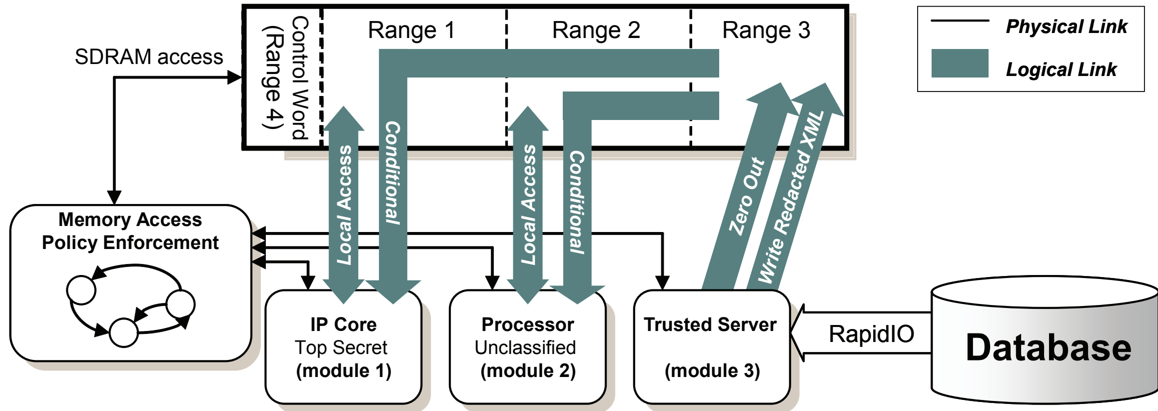

Figure 9: A redaction architecture. A database contains both Top Secret and Unclassified data. $Module_1$ has a Top Secret (TS) clearance, and $Module_2$ has an Unclassified (U) clearance. Any database query requested by $Module_2$ must have all TS data redacted by the Trusted Server $Module_3$. Furthermore, $Module_2$ 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.

module requires its own state machine. Figure 8 shows the DFA that recognizes legal accesses for this policy.

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.

4.4 Redaction

Our security language can also be used to enforce forms of redaction [53], even at very high throughput (such as for video). Military hardware such as avionics [64] 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 [58]. However, the TS component may be required to send a document to the U component; a third component does this by redacting TS dat afrom the document. Figure 9 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 $Module_1$, and $Range_2$ belongs to $Module_2$. $Module_3$ 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:

$rw \rightarrow r \mid w;$
$Access_2 \rightarrow \{Module_1,rw,Range_1\} \mid \{Module_1,r,Range_3\}$
$\quad \mid \{Module_2,rw,Range_2\} \mid \{Module_2,w,Range_4\} \mid \{Module_3,rw,Range_3\};$
$Access_1 \rightarrow \{Module_2,r,Range_3\} \mid Access_2;$
$Trigger \rightarrow \{Module_1,w,Range_4\};$
$Clear \rightarrow \{Module_3,z,Range_3\};$
$SteadyState \rightarrow (Access_2 \mid Clear\ Access_1{}^*\ Trigger)^*;$
$Policy \rightarrow \epsilon \mid Access_1{}^* \mid Access_1{}^*\ Trigger\ SteadyState$
$\quad \mid Access_1{}^*\ Trigger\ SteadyState\ Clear\ Access_1{}^*;$

$Access_1$ is the less restrictive access mode, and $Access_2$ is the more restrictive access mode. The Trigger event changes the access mode from $Access_1$ to $Access_2$, and the Clear event causes the machine to transition from $Access_2$ back to $Access_1$. 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.

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-up), and a subject may not write to an object with a lower security label (no write-down) [4]. 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:

$Access_{B\&L} \rightarrow \{Module_1,r,Range_1\} \mid \{Module_1,r,Range_2\} \mid \{Module_2,r,Range_2\}$
$\quad \mid \{Module_2,w,Range_1\} \mid \{Module_2,w,Range_2\};$
$Policy \rightarrow (Access_{B\&L})^*;$

In our simple example, $Module_1$ has a TS label, $Module_2$ has a U label, $Range1$ has a S label, and $Range_2$ has a U label. We leave to future work the covert channel analysis of these mechanisms.

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:

$Access_1 \rightarrow \{Module_1,r,Range_1\} \mid \{Module_1,r,Range_2\} \mid \{Module_1,w,Range_2\}$
$\quad \mid \{Module_2,w,Range_1\};$
$Access_2 \rightarrow Access_{B\&L} \mid \{Module_1,w,Range_1\};$
$Access_3 \rightarrow Access_1 \mid \{Module_2,w,Range_2\};$
$Access_4 \rightarrow Access_1 \mid \{Module_1,w,Range_1\} \mid \{Module_2,w,Range_2\};$
$Trigger_1 \rightarrow \{Module_1,w,Range_1\};$
$Trigger_2 \rightarrow \{Module_1,w,Range_2\};$
$Path_1 \rightarrow (\epsilon \mid Trigger_1\ Access_2{}^*\ (\epsilon \mid Trigger_2\ Access_4{}^*));$
$Path_2 \rightarrow (\epsilon \mid Trigger_2\ Access_3{}^*\ (\epsilon \mid Trigger_1\ Access_4{}^*));$
$Policy \rightarrow Access_{B\&L}{}^* \mid (\epsilon \mid Path_1 \mid Path_2);$

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 $O(T!T + (N)(S))$.

4.7 Biba Integrity Model

The Biba model is the dual of the Bell-LaPadula model [6], 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:

$Access_{Biba} \rightarrow \{Module_1,w,Range_1\} \mid \{Module_1,w,Range_2\} \mid \{Module_2,r,Range_1\}$
$\quad \mid \{Module_2,r,Range_2\} \mid \{Module_2,w,Range_2\};$
$Policy \rightarrow (Access_{Biba})*;$

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.

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:

$Policy \rightarrow Policy_1\ (\epsilon \mid Trigger_1\ (Policy2)\ (\epsilon \mid Trigger_2\ (Policy_3)));$

$Policy_1$, $Policy_2$, 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 $Policy_3$. Every state in $Policy_1$ has an additional transition ($Trigger_1$) to the first state of $Policy_2$, and every state in $Policy_2$ has an additional transition ($Trigger_2$) to the first state of $Policy_3$. 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 $Policy_N$. 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 $Policy_N$.

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 [8] [18].

Although switching back and forth between an arbitrary number of stateful policies would require modifiying 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:

$SteadyState \rightarrow (Policy_2 \mid Trigger_2\ Policy_1\ Trigger_1)*;$
$Policy \rightarrow Policy_1 \mid Policy_1\ Trigger_1\ SteadyState$
$\mid \quad Policy_1\ Trigger_1\ SteadyState\ Trigger_2\ Policy_1 \mid \epsilon;$

$Trigger_1$ changes the policy from $Policy_1$ to $Policy_2$, and $Trigger_2$ changes the policy back to $Policy_1$.

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.

5.1 Enforcement Architecture

The placement of the enforcement mechanism can have a significant impact on the performance of the memory system. Figure 7 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 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 [26].

5.2 Evaluation

Of the different policies we discussed in Section 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 [44].

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[6] the results with Quartus II 4.2 [2]. Compilations are optimized 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.

5.3 Synthesis Results

In general, a DFA for an isolation policy always has exactly one state, and there is one transition for each $\{ModuleID, op, RangeID\}$ 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 10 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.

Figure 11 shows the cycle time (T_{clock}) for machines of various sizes. T_{clock} is primarily the time for one DFA transition, and it is very close to the maximum frequency of this particular Altera Stratix device. Figure 12 shows the setup time (T_{su}), which is primarily the time to determine the range to which the input address belongs. 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.

[6]the back-end handles netlist creation, placement, routing, and optimization for both timing and area

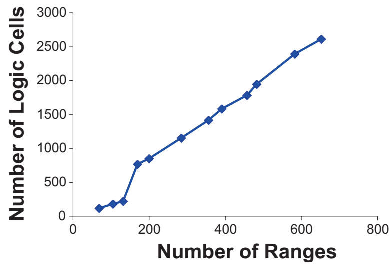

Figure 10: Circuit area versus number of ranges. There is a nearly linear relationship between the circuit area and the number of ranges.

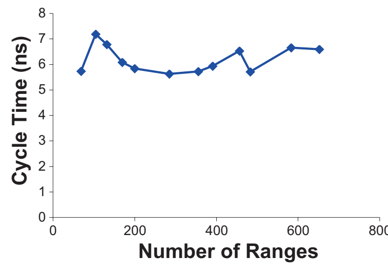

Figure 11: Cycle time versus number of ranges. There is a nearly constant relationship between the cycle time and the number of ranges.

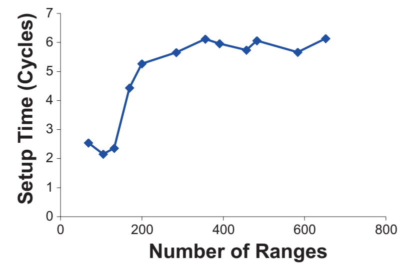

Figure 12: Setup time versus number of ranges. Above 170 ranges, there is a nearly linear relationship between the setup time and the number of ranges. This time can be reduced with pipelining.

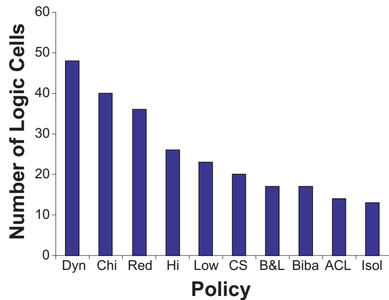

Figure 13: Circuit area versus access policy. The area is related to the number of states, transitions, and ranges. The circuit area is greatest for the dynamic policy.

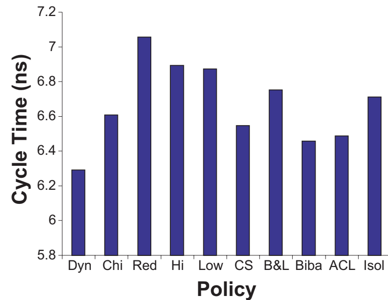

Figure 14: Cycle time for each access policy. Cycle time is greatest for redaction, followed by high water mark and low water mark.

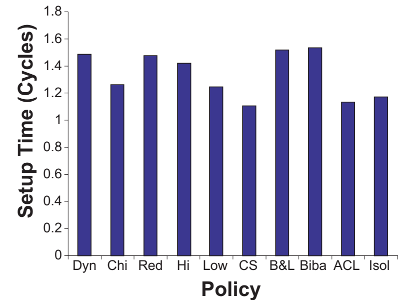

Figure 15: Setup time for each access policy. Setup time is greatest for Biba, followed by B&L and dynamic.

Figure 13 shows the area of the circuits resulting from the example policies presented in this paper. These circuits are much smaller in area than the series of isolation circuits above because the example policies have very few ranges, states, and transitions. 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. 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 14 shows that the cycle time is greatest for redaction, followed by high water mark, low water mark, B&L, and isolation. Figure 15 shows that the setup time is greatest for Biba, followed by B&L, dynamic, redaction, and high water mark.

5.4 Impact of the Reference Monitor on System Performance

Since FPGAs do not operate at high frequency, they achieve their performance from computational parallelism. Many FPGA applications such as DSPs, signal processing, and intrusion detection systems are throughput-driven, but do not have rigid latency requirements. therefore are latency-insensitive. For these applications, we argue that our technique does not impact performance significantly. For example, since an FPGA operating at 200MHz will have a cycle time of 5ns, our reference monitor only adds at most a two cycle delay in this case.

6 Covert Storage Channels

In a covert channel, internal state of the enforcement mechanism is used to transmit information in a manner that is contrary to the security policy. Some stateful policies define (and require) internal state changes that, if unchecked, can form the basis of a covert channel. In this section, we describe a method

of analyzing stateful policies to detect these "inherent" covert channels. We describe a method for measuring the potential bandwidth of the covert channel at runtime so that corrective action can be taken if the bandwidth exceeds a predetermined threshold value.

A reference monitor makes a binary decision to either grant or deny a particular access based on a policy. As a result, with certain stateful policies, subjects may be able to observe the internal state of the policy by observing the reference monitor's decisions. Subjects can change the state of the policy enforcement mechanism by making access requests that affect subsequent requests. The state of the policy is in effect a shared resource. This ability to obsere and modify the policy itself makes it possible for two subjects to illegally communicate.

Some stateful policies only allow a few bits to be leaked, while others allow a high bandwidth of data to be leaked. A stateful policy expressed in a regular language is equivalent to a directed graph, which contains cycles that allow the policy to alternate between two or more states continuously. If certain properties are met, then the cycle represents a possible covert channel. If it is not feasible to revise the policy to eliminate the cycle, 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 can take corrective action. One option for corrective action is to change to a policy without the covert channel. This limits the amount of data that can be leaked to a specific value.

6.1 Storage Channels in Stateful Policy Enforcement Systems

In both covert storage and timing channels, the sender has a higher security label than the receiver and interferes with the receiver's access to a shared system resource (e.g., the processor or disk) to signal the receiver. In a timing channel, the interference changes the time needed for the receiver to perform a task, and the receiver interprets the delay or lack of delay. On the other hand, in a storage channel, the interference changes the receiver's ability to access the resource. This section will focus on storage channels rather than timing channels. We close timing channels with respect to access decisions but assume that data access times, for example, are not otherwise modified by the actions of subjects. 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.

Kemmerer [34] [35] has devised a shared resource matrix method of identifying storage and timing channels in computer systems. 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 [34] [35]. In the case of our embedded system, the sender and the receiver are cores, and the shared resource is the state of the policy.

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. 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. This property allows a stream of information to be leaked if the sender can cause a DFA transition and the two have different access matrices with respect to the receiver such that the receiver can sense the policy change.

For covert channel analysis, 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, in the absence of any evidence to the contrary, we assume that $Module_1$ has a higher security label than $Module_2$.

Figure 16 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 our

criteria for a covert channel: (1) $Module_1$ has a higher security label than $Module_2$, (2) $Module_1$ controls at least one of the transitions within the cycle, and (3) at least two nodes within the cycle have access matrices that differ with respect to $Module_2$.

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 bandwidth of the information flow can be calculated in terms on the number of possible encodings of the data and the probability of each symbol [46] [47], over time.

As mentioned above, one of the criteria for a covert 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, but the sender and receiver would need to be able to sync their activities, e.g., by polling the state of the policy.. This case is shown in Figure 17. $Module_1$ 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. In this case, bandwidth would depend on the estimated rate of $Module_1$'s activity.

Since we are using a static technique to detect possible covert channels, not all possible channels identified can be exploited at runtime. Furthermore, analysis of the design may be required to estimate false positives. 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 18 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 $Module_2$.

6.2 Automatically Detecting Policy-Based Covert Storage Channels

To automatically detect a possible covert channel in a policy, we first determine if its DFA has any cycles. Topological sort is a well-known algorithm for detecting cycles in a directed graph [12]. 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 does not finish in an amount of time proportional to the size of the graph, then the graph contains a cycle. Identifying the set(s) of vertices that make up the cycle(s) involves tracing the graph recursively. As shown:

```
Procedure DetectChannels (Graph G)
{
   Array of Lists Senders
   Array of Lists Receivers
   If (Topological_Sort(G) == False)
      Output ''Graph G Contains No Cycles.''
      Return
   C = Recursively_Trace_Graph_to_Find_Cycles(G)
   For (All Cycles C)
      For (All Edges E in C)
```

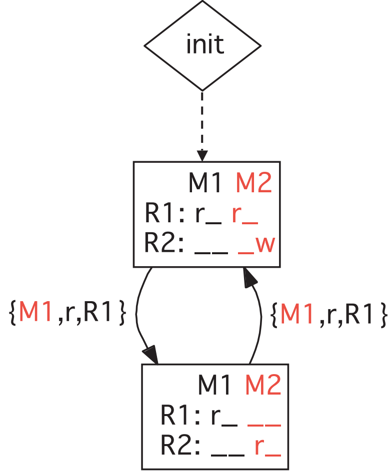

Figure 16: A non-trivial cycle. This figure shows the DFA that enforces a security policy. Each node of the graph is a state in the policy, and we show the access matrix at each node. $Module_1$ has a higher security label than $Module_2$. Initially, $Module_2$ can read $Range_1$. $Module_1$ can then 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 $Range_1$. According to our criteria, there is a possible storage channel from $Module_1$ to $Module_2$.

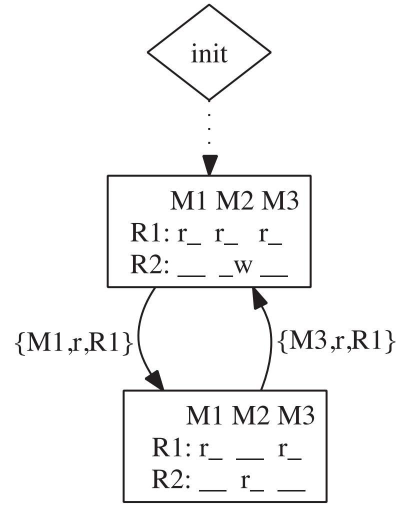

Figure 17: Suppose that $Module_1$ would like to leak some data to $Module_2$. In this example, $Module_1$ causes one of the transitions, and $Module_3$ causes the other transition. The two access matrices differ with respect to $Module_2$. Since $Module_3$ is not a party in the exchange, $Module_1$ must wait a sufficient length of time for $Module_3$'s transition to occur, allowing the cycle to come around again.

```
            M = Module that causes transition E
            Add M to Senders[C]
        For (All Vertices V in C)
                For (All Vertices V' in C) if V' != V then
                    For (All Rows row)
                        For (All Columns col)
                            If (Matrix(V)[row][col] != Matrix(V')[row][col])
                                Add col to Receivers[C]
    Output ''Possible Covert Channels:''
    For (All Cycles C Found)
        Output Cross_Product(Senders[C], Receivers[C])
    Return
}
```

We applied our covert channel detector to several example policies. Figure 19 shows the redaction policy described in Section 4.4, which alternates between a more restrictive and less restrictive access matrix. Our detector correctly identified 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 $Module_2$.

Dynamic policies that switch back and forth between two or more policies may have covert channels [67] [68]. One way of dealing with the problem of covert 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" to not drive the covert channel in violation of the policy. Even if a covert channel exists, if policy switching is infrequent, the bandwidth is low, and the channel might be acceptable.

In Section 4.3, we described a Chinese wall policy shown in Figure 8 with two conflict-of-interest classes: $\{Range_1, Range_2\}$ and $\{Range_3, Range_4\}$. Although it does not have any cycles, this policy is

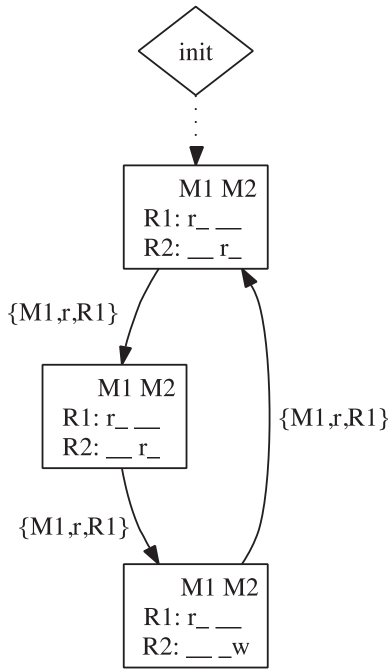

Figure 18: A cycle consisting of three nodes. Two of the nodes are identical with respect to $Module_2$, but one is different from the other two. Since at least one node differs, $Module_2$ can detect the change, allowing data to leak from $Module_1$ to $Module_2$.

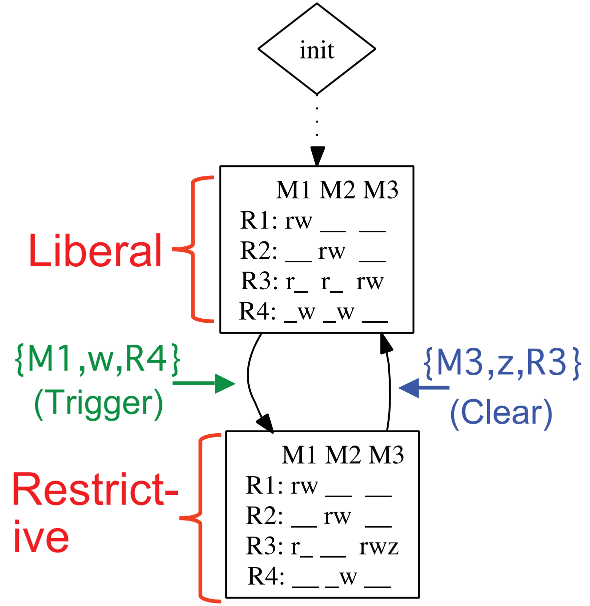

Figure 19: This redaction policy has four possible covert channels: from $Module_1$ to $Module_2$, from $Module_1$ to $Module_3$, from $Module_3$ to $Module_1$, and from $Module_3$ to $Module_2$.

not completely free of covert channels. $Module_1$ could leak one bit of information to $Module2$ and one bit to $Module3$, 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 may be 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 is bounded by the longest path length from the initial state to any final state.

6.3 Approaches to Covert Channel Management

Once a possible covert channel has been identified, the system designer, working with the enterprise security manager, can modify the policy in order to eliminate the problematic cycle. If this is not an option, we can close the covert channel or confine the bandwidth within certain limits, restricting the behavior of the system. This approach requires a method of tracking current bandwidth. One way of measuring the usage bandwidth of the covert channels is with counters. A counter keeps track of the number of times the cycle occurs within a sliding window of time relative to the current time, and the system ensures that the covert channel bandwidth stays below a threshold value.

6.3.1 Approaches for Measuring Covert Channels

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 $State_1$ to $State_2$ to $State_3$ and back to $State_1$ 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.

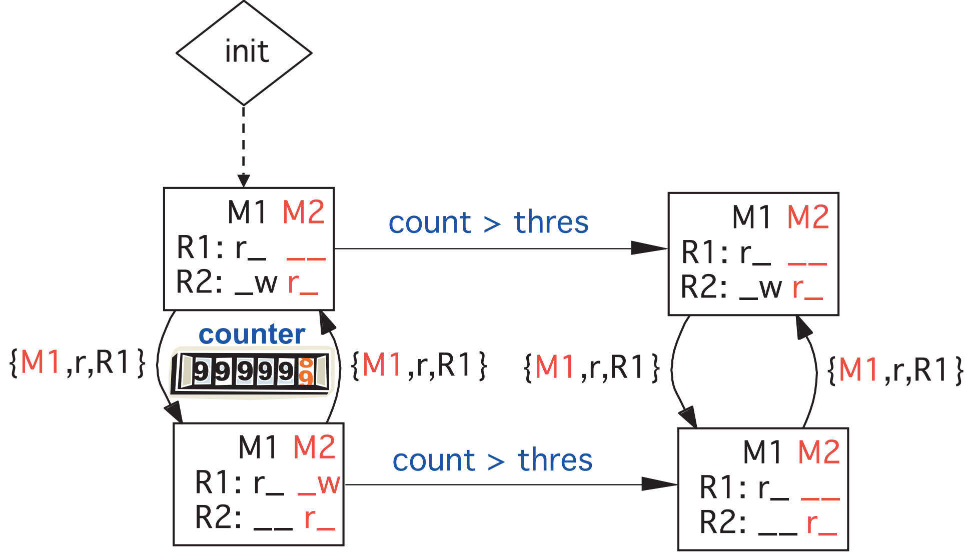

Figure 20: Coping with a covert channel in a stateful policy with two states. A counter measures the bandwidth of a covert channel from $Module_1$ to $Module_2$ by counting the number of times that a cycle occurs in the original policy on the left. If this counter exceeds a predetermined threshold, it 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 $Module_2$.

6.3.2 Options for Closing and Throttling Covert Channels

As we explained in Section 6.1, 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 20 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, during the measurement window, the policy changes so that the nodes in the cycle are identical with respect to $Module_2$, thus closing the covert channel. This is accomplished in the example by revoking $Module_2$'s privilege to write to $Range_1$ in the second state of the stateful policy. After a period of time, the policy can revert, if desired. This will be highly desirable in practice. Alternatively, we can add delays to transition between states to throttle the channel within acceptable limits.

Modification to the policy as above adds to the amount of logic that must reside on the FPGA. 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 a counter to exceed its threshold. In other words, the revocation would be 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, although the combined policy should not introduce any new covert channels if done correctly.

6.4 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 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 [17]. 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 [39], several techniques for detecting covert channels in policy specifications have been proposed, including shared resource matrix methodology [34], information flow [45], and noninterference [23] [20], although this section 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 [62]. 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 convert channels

There is much prior work in estimating the bandwidth of covert channels. Millen applied information theory to calculate the bandwidth of a covert channel based on the number of possible encodings of the data and the probability of each symbol [46] [47]. Shieh proposes a method of measuring the bandwidth of covert channels in multilevel operating systems [57]. 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 [61].

7 Conclusions

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 coarse or fine-grained computer security policies for 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 requirements 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.

We have also developed an automatic method of identifying security policies with inherent covert channels. We have identified a range of corrective actions that can be considered once a possible covert channel is detected. The ideal alternative is 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 closing it when it reaches a designated threshold, as determined by a hardware counter that measures the real-time bandwidth of potential covert channel exploitation.

Since expressing some policies in our language requires complex expressions, we do not expect the typical engineer to work in our language. Because usability is fundamental to system security [29] [21], we plan to develop a higher-level language along with a set of tools to assist the engineer in constructing mathematically precise policies. This will build on the policy engineering work of Fong et al. [17]. A higher-level language will allow the engineer to express policies in terms of security concepts (e.g., isolation, controlled sharing, etc.) rather than in terms of modules and ranges.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, MA, 1988.

[2] Altera Inc. Quartus II Manual, 2004.

[3] J.P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, ESD/AFSC, Hanscorn AFB, Bedford, MA, 1972.

[4] D.E. Bell and L.J. LaPadula. *Secure Computer Systems: Mathematical Foundations and Model*. The MITRE Corporation, Bedford, MA, USA, May 1973.

[5] Vaughn Betz, Jonathan Scott Rose, and Alexander Marqardt. *Architecture and CAD for deep-submicron FPGAs*. Kluwer Academic, Boston, MA, 1999.

[6] K.J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, 1977.

[7] K. Bondalapati and V.K. Prasanna. Reconfigurable computing systems. In *Proceedings of the IEEE*, volume 90(7), pages 1201–17, 2002.

[8] L. Bossuet, G. Gogniat, and W. Burleson. Dynamically configurable security for SRAM FPGA bitstreams. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, Santa Fe, NM, April 2004.

[9] D.F.C. Brewer and M.J. Nash. The chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, 1989.

[10] D.A. Buell and K.L. Pocek. Custom computing machines: an introduction. In *Journal of Supercomputing*, volume 9(3), pages 219–29, 1995.

[11] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. In *ACM Computing Surveys*, volume 34(2), pages 171–210, USA, 2002. ACM.

[12] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990.

[13] A. DeHon. Comparing computing machines. In *SPIE-Int. Soc. Opt. Eng. Proceedings of SPIE - the International Society for Optical Engineering*, volume 3526, pages 124–33, 1998.

[14] A. DeHon and J. Wawrzynek. Reconfigurable computing: what, why, and implications for design automation. In *Proceedings of the Design Automation Conference*, pages 610–15, West Point, NY, 1999.

[15] D.E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), May 1976.

[16] Ulfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: A retrospective. In *Proceedings of the 1999 Workshop on New Security Paradigms*, 1999.

[17] Philip W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, 2004.

[18] T. Fraser and L. Badger. Ensuring continuity during dynamic security policy reconfiguration in dte. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 15–26, 1998.

[19] Amer Gerzic. Codeguru: Write your own regular expression parser, November 2003.

[20] J.A. Goguen and J. Meseguer. Security policy and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[21] Peter Gutmann and Ian Grigg. Security usability. *IEEE Security and Privacy Magazine*, July/August 2005.

[22] I. Hadzic, S. Udani, and J. Smith. FPGA viruses. In *Proceedings of the Ninth International Workshop on Field-Programmable Logic and Applications (FPL '99)*, Glasgow, UK, August 1999.

[23] J.T. Haigh, R.A. Kemmerer, J. McHugh, and W.D. Young. An experience using two convert channel analysis techniques on a real system design. *IEEE Transactions on Software Engineering*, 13(2):157–168, February 1987.

[24] Scott Harper and Peter Athanas. A security policy based upon hardware encryption. In *Proceedings of the 37th Hawaii International Conference on System Sciences*, 2004.

[25] Scott Harper, Ryan Fong, and Peter Athanas. A versatile framework for fpga field updates: An application of partial self-reconfiguration. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping*, June 2003.

[26] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, and R. Kastner. Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2007.

[27] Ted Huffmire, Shreyas Prasad, Tim Sherwood, and Ryan Kastner. Policy-driven memory protection for reconfigurable systems. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Hamburg, Germany, September 2006.

[28] C. Irvine, T. Levin, T. Nguyen, and G. Dinolt. The trusted computing exemplar project. In *Proceedings of the 5th IEEE Systems, Man and Cybernetics Information Assurance Workshop*, pages 109–115, West Point, NY, June 2004.

[29] Cynthia E. Irvine, Timothy E. Levin, Thuy D. Nguyen, David Shifflett, Jean Khosalim, Paul C. Clark, Albert Wong, Francis Afinidad, David Bibighaus, and Joseph Sears. Overview of a high assurance architecture for distributed multilevel security. In *Proceedings of the 2002 IEEE Workshop on Information Assurance and Security*, West Point, NY, June 2002.

[30] S. Johnson. Yacc: Yet another compiler-compiler. Technical Report CSTR-32, Bell Laboratories, Murray Hill, NJ, 1975.

[31] Ryan Kastner, Adam Kaplan, and Majid Sarrafzadeh. *Synthesis Techniques and Optimizations for Reconfigurable Systems*. Kluwer Academic, Boston, MA, 2004.

[32] T. Kean. Secure configuration of field programmable gate arrays. In *Proceedings of the 11th International Conference on Field Programmable Logic and Applications (FPL '01)*, Belfast, UK, August 2001.

[33] T. Kean. Cryptographic rights management of FPGA intellectual property cores. In *Tenth ACM International Symposium on Field-Programmable Gate Arrays (FPGA '02)*, Monterey, CA, February 2002.

[34] R.A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. In *ACM Transactions on Computer Systems*, 1983.

[35] R.A. Kemmerer. A practical approach to identifying storage and timing channels: Twenty years later. In *Proceedings of the 18th Annual Computer Security Applications Conference*, Las Vegas, Nevada, USA, December 2002.

[36] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi. Security as a new dimension in embedded system design. In *Proceedings of the 41st Design Automation Conference (DAC '04)*, San Diego, CA, June 2004.

[37] J. Lach, W. Mangione-Smith, and M. Potkonjak. FPGA fingerprinting techniques for protecting intellectual property. In *Proceedings of the 1999 IEEE Custom Integrated Circuits Conference*, San Diego, CA, May 1999.

[38] J. Lach, W. Mangione-Smith, and M. Potkonjak. Robust FPGA intellectual property protection through multiple small watermarks. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation (DAC '99)*, New Orleans, LA, June 1999.

[39] B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):842–856, October 1973.

[40] M. Lesk and E. Schmidt. Lex: A lexical analyzer generator. Technical Report 39, Bell Laboratories, Murray Hill, NJ, October 1975.

[41] Timothy E. Levin, Cynthia E Irvine, and Thuy D. Nguyen. A least privilege model for static separation kernels. Technical Report NPS-CS-05-003, Naval Postgraduate School, 2004.

[42] Peter Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett, Sudbury, MA, 2001.

[43] W.H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V.K. Prasanna, and H.A.E. Spaanenburg. Seeking solutions in configurable com-

puting. In *Computer*, volume 30(12), pages 38–43, 1997.

[44] Giovanii De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[45] J.K. Millen. Security kernel validation in practice. *Communications of the ACM*, 19(5):243–250, May 1976.

[46] J.K. Millen. Covert channel capacity. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, April 1987.

[47] J.K. Millen. Finite-state noiseless covert channels. In *Proceedings of the Computer Security Foundations Workshop II*, Franconia, NH, USA, June 1989.

[48] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002.

[49] D. Raymond and D. Wood. Grail: A C++ library for automata and expressions. *Journal of Symbolic Computation*, 11:341–350, 1995.

[50] John Rushby. A trusted computing base for embedded systems. In *Proceedings 7th DoD/NBS Computer Security Conference*, pages 294–311, September 1984.

[51] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[52] J. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[53] O. Sami Saydjari. Multilevel security: Reprise. *IEEE Security and Privacy Magazine*, September/October 2004.

[54] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh. A quick safari through the reconfiguration jungle. In *Proceedings of the Design Automation Conference*, pages 172–7, 2001.

[55] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1), February 2000.

[56] A.W. Senior, S. Pankanti, A. Hampapur, L. Brown, Y-L Tian, and A. Ekin. Blinkering surveillance: Enabling video privacy through computer vision. Technical Report RC22886, IBM, 2003.

[57] S. Shieh. Estimating and measuring covert channel bandwidth in multilevel secure operating systems. *Journal of Information Science and Engineering*, 15:91–106, 1999.

[58] Richard E. Smith. Cost profile of a highly assured, secure operating system. In *ACM Transactions on Information and System Security*, 2001.

[59] F. Standaert, L. Oldenzeel, D. Samyde, and J. Quisquater. Power analysis of FPGAs: How practical is the attack? *Field-Programmable Logic and Applications*, 2778(2003):701–711, September 2003.

[60] D.F. Stern. On the buzzword "security policy". In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 219–230, Oakland, CA, 1991.

[61] C.R. Tsai and V. Gligor. A bandwidth computation model for covert stroage channels and its applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 108–121, 1988.

[62] C.R. Tsai, V. Gligor, and C. Chandersekaran. On the identification of covert storage channels in secure systems. *IEEE Transactions on Software Engineering*, 16(6), June 1990.

[63] J.E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H.H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 4(1), pages 56–69, 1996.

[64] Clark Weissman. MLS-PCA: A high assurance security architecture for future avionics. In *Proceedings of the Annual Computer Security Applications Conference*, pages 2–12, Los Alamitos, CA, December 2003. IEEE Computer Society.

[65] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, October 2002.

[66] T. Wollinger, J. Guajardo, and C. Paar. Security on FPGAs: State-of-the-art implementations and attacks. *ACM Transactions on Embedded Computing Systems*, 3(3):534–574, August 2004.

[67] J. Woodward. Exploiting the dual nature of sensitivity labels. In *IEEE Symposium on Security and Privacy*, pages 23–30, Oakland, CA, USA, 1987.

[68] L. Zheng and A. Myers. Dynamic security labels and noninterference. Technical Report 2004-1924, Cornell University, 2004.