Memory Protection for Reconfigurable Hardware Security

Abstract

While processor based systems often enforce some form of memory protection to prevent the unintended sharing of data between processes, current systems built around reconfigurable hardware typically offer no such protection. Reconfigurable hardware, such as an FPGA, allows designers to implement high speed circuits yet is capable of software-style updates. As the number of commercial systems deployed in reconfigurable hardware increases, the security of these systems becomes a serious design concern.

Several reconfigurable cores (which can include custom accelerators, off-the-shelf logic blocks, and even small processors) 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 memory protection mechanism capable of enforcing both fixed and transitional access policies expressed as a formal language. Our approach includes a specialized compiler that translates a policy of legal sharing to reconfigurable logic blocks which can be directly transferred to an FPGA. We demonstrate that our technique is powerful enough to enforce protection across a variety of classic security scenarios. The efficiency of our access language design flow is evaluated in terms of area and cycle time, and we show that our methods are scalable with the number of memory access ranges that must be recognized.

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. The bit-level reconfigurability of these devices can be used to implement highly optimized circuits for everything from encryption to FFT, or even entire customized processors. Because one device is used for so many different functions, special purpose circuits can be developed and deployed at a fraction of the cost associated with custom fabrication. Furthermore, if the design needs to be updated, 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 are simply nonexistent.

Due to Moore’s law, digital systems 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 has increased the possibility that security vulnerabilities may surface in one or more components, which could threaten the entire device. New methods that can provide separation and security in highly integrated reconfigurable devices are needed.

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 processor will typically use virtual memory and TLBs to enforce some form of memory protection, reconfigurable devices usually operate in the physical addresses space with no operating system support. Lacking these mechanisms, any hardware module can read or write to the memory of any other module at any time, whether purposefully, accidentally, or maliciously. This situation calls for a memory access policy 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 formal description that establishes what accesses to memory are legal and which are not. Our method rests on the ability to formally describe the access
policy using a specialized language. We present a 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 and enforce the policy.

The techniques presented in this paper are steps towards a cohesive methodology for those seeking to build reconfigurable systems with modules acting at different security clearance levels on a single chip. In order for such a methodology to be adopted by the embedded design community it is critical that the resulting hardware is both high performance and low overhead. Furthermore, it is important that our methods are both formally grounded and yet understandable to those outside the security discipline. Throughout this paper we strive to strike a balance between engineering and formal evaluation. Specifically, this paper makes the following contributions:

- We specify a memory access policy language, based on formal regular languages, which expresses the set of legal accesses and allowed policy transitions.
- We demonstrate how our language can express classical security scenarios, such as compartmentalization, secure hand-offs, Chinese walls, access control lists and an example of multi-level security.
- 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.

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 design flow. In Section 4, we describe our access policy language including several example policies. We present our synthesis results in Section 5. Finally, we conclude in Section 6 and show 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 [6, 9, 11, 19, 28, 36]. The power of re-
configurable 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 [11]. 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 [8, 10, 41]. 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 [22]. 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.

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.

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 [7, 21, 24] and to secure hardware update channels [15, 14]. However, few researchers have begun to consider the security ramifications of compromised hardware [13].

It is important to understand the different attacks against FPGAs that are possible in order to develop countermeasures [44]. 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 [39]. 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 [13]. Solutions to these problems include encryption [7] [20] [21], fingerprinting [23], and watermarking [24]. While there are a variety of attacks possible, our work is concerned with addressing the complete lack of memory protection available on most modern reconfigurable systems. In particular this paper is concerned with techniques to provide separation between multiple interacting cores and modules in the off-chip memory. In our attack
model, there may be malicious modules or remote attacks that originate from the network. We assume that the attacker does not have physical access to the device.

### 2.1.2 Protecting Memory on an FPGA

A successful run-time management system must protect different logical modules from interfering, intercepting, or corrupting any use of a shared resource. On an embedded system, the primary resource of concern is memory. Whether it is on-chip block RAM, off-chip DRAM, or backing-store such as Flash, a serious issue in the design of any high performance secure system is the allocation and reallocation of memory in a way that is efficient, flexible, and protected. On a high performance processor, security domains may be enforced through the use of a page table. Superpages, which are very large memory pages, can also be used to provide memory protection, and their large size makes it possible for the TLB to have a lower miss rate [30]. Segmented Memory [34] and Mondrian Memory Protection [43], a finer-grained scheme, address the inefficiency of providing memory protection at the granularity of a page (or a superpage) by allowing different protection domains to have different permissions on the same memory region.

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

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

To provide separation in memory between multiple different interacting modules we adapt some of the key concepts from separation kernels. Rushby originally proposed that a separation kernel [16] [26] [32] creates within a single shared machine an environment which supports the various components of the system, and it provides the communication channels between them in such a way that individual components of the system cannot distinguish this shared environment from a physically distributed one. A separation kernel divides all resources under its control into blocks such that the actions of a subject in one block are isolated from (viz., cannot be detected by or communicated to) a subject in another block, unless an explicit means for that communication has been established. For a multilevel secure system, each block typically represents a different classification level.

We propose that the reconfigurable nature of these gate arrays offers a new method by which the fine grain control of access to off-chip memory is possible. By building a specialized circuit that recognizes a *language of legal accesses*, and then by realizing that circuit directly onto the reconfigurable device as a specialized state machine, every memory access can be checked with only a small additional latency. Although incorporating the enforcement module into a separate hardware module would lessen the impact of covert channel attacks, this would introduce additional latency. The problem of covert channels in FPGAs can be mitigated by enforcing the physical separation of modules and by employing other techniques that work at the gate level, but this work is outside the scope of this paper.

### 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 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.

3.1 Memory Access Policy

Once a high level policy is developed based on the requirements of the system and the organizational security policy [40], it must be expressed in a concrete form to allow engineers to build 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 [37], which monitor the execution of a target, which in our case is one or more modules on the FPGA. An execution monitor must be able to monitor all memory accesses and able to halt or block the execution of the target if it attempts to violate the security policy. Allowing a misbehaving module to continue executing might let it figure out the inner workings of the DFA logic of the enforcement mechanism. In addition, all modules must be isolated from the enforcement mechanism so that they cannot interfere with the DFA transitions. We discuss techniques for module isolation in Section 5.1. The enforcement mechanism is also a Reference Validation Mechanism (RVM) [3], which must be tamperproof, always invoked, and small enough to be subject to analysis and test, the completeness of which can be assured.

Although there exist security policies that execution monitors are incapable of enforcing, such as information flow policies [33], 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. 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 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 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 units of separation of the FPGA as Modules. The Access Methods (A) are typically Read and Write, but may include special memory operators such as zeroing or incrementing if required. The set P is a partitioning of physical memory into ranges. Memory Range Specifier (R in P) describes a physical address or set of 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 access rights (A: read, write, etc.), principals (M: modules), and objects (R: memory ranges1).

The terminals of the language are memory accesses descriptors which ascribe a specific right to a specific module for a specific object for the duration of the next memory access. Formally, the terminals of the productions are tuples of the form (A, M, R), and the universe of tuples forms an alphabet \( \Sigma = A \times M \times R \). The memory access policy description precisely defines a formal language \( L \subseteq \Sigma^* \) which is almost always infinite (unless the device only supports a fixed number of accesses). \( L \) needs to satisfy the property that

---

1 An interval of the address space including high (\( R_{\text{high}} \)) and low (\( R_{\text{low}} \)) bounds
∀xt | t ∈ Σ, xt ∈ L : x ∈ L. This has the effect that any legal access pattern 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. A memory access (A, M, k), where k is a particular address, is contained in a memory access descriptor (A′, M′, R) iff A = A′, M = M′, and R_{low} ≤ k ≤ R_{high}. A sequence of memory accesses a = a_0, a_1,..., a_n is said to be legal iff ∃s = s_0, s_1,..., s_n ∈ L s.t. ∀0 ≤ i ≤ n s_i contains a_i. In order to turn this into an enforceable method we need two things.

1. A method by which L can be precisely defined

2. An automatically created circuit which recognizes memory access sequences that are legal under L

We begin with a description of (1) through the use of a simple example. Consider a very straightforward 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 input format, this is coded as the following set of productions:

\[
\begin{align*}
\text{rw} & \rightarrow r \mid w; \\
\text{Range}_1 & \rightarrow [0x8e7b008,0x8e7b00f]; \\
\text{Range}_2 & \rightarrow [0x8e7b018,0x8e7b01b]; \\
\text{Access}_1 & \rightarrow \{\text{Module}_1, \text{rw}, \text{Range}_1\}; \\
\text{Access}_2 & \rightarrow \{\text{Module}_2, \text{rw}, \text{Range}_2\}; \\
\text{Policy} & \rightarrow (\text{Access}_1 | \text{Access}_2)^*;
\end{align*}
\]

Each of these productions is a re-writing rule as in a standard grammar. The non-terminal Policy is the start symbol of the grammar and defines the overall access policy. Note that Policy is essentially a regular expression that describes L. Through the use of a grammar we allow the hierarchical composition of more complex policies. In this case Access_1 and Access_2 are simple access descriptors, but in general they could be more complex expressions that recognize a set of legal memory access.

Since we eventually want to compile the access policy to hardware, we limit our language to constructs with computational power no greater than a regular expression [27] with the added ability to detect ranges. Although a regular language must have a type-3 grammar in the Chomsky hierarchy, it is inconvenient for security administrators to express policies in right-linear or left-linear form. Since a language can be recognized by many grammars, any grammar that can be transformed into type-3 form is acceptable. This transformation can be accomplished 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 size. The smallest granularity that we enforce currently is at the word boundary, and we can support any sized range from word to the entire address space. There is no reason that ranges have to be of the same size or even close, unlike 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 compartmentalization and Chinese wall in order to demonstrate the versatility and efficiency of our approach. In Section 4.4 we describe a “multilevel security policy” in which multiple security clearance levels are interacting within a single embedded system. However, now that we have introduced our memory access policy specification language, we describe how it can be transformed automatically to a efficient circuit for implementation on an FPGA.
Figure 2: We use Lex and Yacc to build a parse tree from the simple access policy. Next, we transform the parse tree by applying the productions of the policy to the tree. We substitute the left hand side of a production with its right hand side. From the resulting regular expression, we use Thompson’s Algorithm to construct the NFA. Transitions are checked in parallel in an NFA. We use subset construction and Hopcroft’s Algorithm to convert the NFA to the minimized DFA on the right.

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 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:

- Create the access policy (described above).
- Build a syntax tree from the policy.
- Transform syntax tree to an expanded intermediate form.
- Expand Policy to a regular expression defined over the alphabet $\Sigma$.
- Convert the regular expression to a non-deterministic finite automaton (NFA).
- Construct an equivalent minimized state machine from the NFA.
- Break down the ranges into sizes that are a power of two.
- Organize the set of ranges as a trie$^2$, and create a logic tree that recognizes them.
- Export the state machine and range detection logic as Synthesizable Verilog.
- Synthesize, Place, and Route Circuit
- Load 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 consider a simple compartmentalization policy with two modules, which can only access their own single range:

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

**Building and Transforming a Parse Tree** — Next, we use Lex [25] and Yacc [18] 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.

\[\text{an ordered tree data structure for storing lookup tables}\]
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 2 shows the transformed parse tree for our policy.

**Building the Regular Expression** – Next, we find the subtree corresponding to Policy and traverse this subtree to obtain the regular expression. By this stage we have completely eliminated all of the 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, an NFA can be constructed from this regular expression using Thompson’s Algorithm [1]. Figure 2 shows the NFA for our policy.

**Converting the NFA to a DFA** – From this NFA we can construct a DFA through subset construction [1]. Following the creation of the DFA, we apply Hopcroft’s Partitioning Algorithm [1] as implemented by Grail [31] to minimize the DFA. Figure 2 shows the minimized DFA for our policy on the right.

**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. Here we express Range_{1} and Range_{2} in binary:

Range_{1}: [0000 1000 1110 0111 1011 0000 0000 1000, 0000 1000 1110 0111 1011 0000 0000 1111]

Range_{2}: [0000 1000 1110 0111 1011 0000 0001 1000, 0000 1000 1110 0111 1011 0000 0001 1111]

Notice that for each of the addresses in the example, all of the bits are the same except for the least significant bits. We can express the intervals more concisely by using \(X\) as a “don’t care” bit to denote either 0 or 1:

Range_{1}: 0000 1000 1110 0111 1011 0000 0000 1XXX

Range_{2}: 0000 1000 1110 0111 1011 0000 0001 10XX

Our system converts the ranges to an internal format using these 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.\(^3\) Using this optimization, any aligned power of two range 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 8 shows the hardware module we wish to build. There are 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. The DFA transitions are the concatenation of the module ID, op, and a range ID bit vector. The range ID bit vector contains one bit for each range ID in the policy. The hardware will check all the ranges in parallel and set to 1 the bit corresponding to the range ID that contains the input address. If there is no

\(^3\)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
transition for an input character, 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 insert illegal characters into the input.

**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 how to compartmentalize access to different modules, and it is trivial to extend the above policy 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. In particular we demonstrate how data may be securely handed off between modules, and we also show the Chinese wall policy. Before we do that let us first discuss another more 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, $\text{Range}_1$ and $\text{Range}_2$. $\text{Class}_1$ is a class of principals ($\text{Module}_1$ and $\text{Module}_2$), and $\text{Class}_2$ is another class of principals ($\text{Module}_3$ and $\text{Module}_4$). Either $\text{Class}_1$ or $\text{Class}_2$ may access $\text{Range}_1$, but only $\text{Class}_2$ may access $\text{Range}_2$:

\[
\text{Class}_1 \rightarrow \text{Module}_1 \mid \text{Module}_2; \\
\text{Class}_2 \rightarrow \text{Module}_3 \mid \text{Module}_4; \\
\text{List}_1 \rightarrow \text{Class}_1 \mid \text{Class}_2; \\
\text{List}_2 \rightarrow \text{Class}_2; \\
\text{Access}_1 \rightarrow \{\text{List}_1, \text{rw}, \text{Range}_1\}; \\
\text{Access}_2 \rightarrow \{\text{List}_2, \text{rw}, \text{Range}_2\}; \\
\text{Policy} \rightarrow (\text{Access}_1 \mid \text{Access}_2)^*;
\]

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, $\text{Module}_1$, $\text{Module}_2$, $\text{Module}_3$, and $\text{Module}_4$ may access $\text{Range}_1$, and $\text{Module}_3$ and $\text{Module}_4$ may access $\text{Range}_2$. The total number of transitions in this example is $4+2=6$.

4.2 Secure Hand-off

Many protocols require the ability to securely hand-off information from one party to another. Embedded systems often implement these protocols, and our language makes these transfers possible. 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 $\text{Module}_1$ wants to securely hand-off some data to $\text{Module}_2$. $\text{Module}_1$ writes some data to memory, to which it must have exclusive access, and then $\text{Module}_2$ reads the data from memory. Rather than communicating the data, an access policy can be compiled that will allow the critical transition of permissions in synchronization with the hand-off. Using formal languages to express security policies makes such a temporal hand-off possible.

After a certain trigger event occurs, it is possible to revoke the permissions of a module so that it may no longer access one or more ranges. Consider the following example:
Figure 3: The Venn Diagram on the left shows two conflict-of-interest classes, ClassA and ClassB. A principal that accesses Range4 (black) is subsequently prohibited from accessing Range3 (dark gray), but it may access either Range1 (white) or Range2 (light gray), because they are in a different class. The DFA on the right recognizes legal accesses for this Chinese Wall policy. An access to Range4 results in a transition to state 2 (black), from which an access to Range1 results in a transition to state 1 (black or white).

Module1|2 → Module1 | Module2;
Access1 → {Module1,rw,Range1} | {Module1|2,rw,Range2};
Access2 → {Module2,rw,(Range1|Range2)};
Trigger → {Module1,rw,Range2};
Policy → (Access1)*(ε | Trigger(Access2)*);

At first, Module1 can access Range1 or Range2, and Module2 can access Range2. However, the first time Module1 accesses Range2 (indicating that Module1 is ready to hand off), Access1 is deactivated by this trigger event, revoking the permissions for Module1 from both Ranges. As a result of the trigger, Module2 has exclusive access to Range1 and Range2, where the exclusivity is represented by parentheses.

4.3 Chinese Wall

Another security scenario that can be efficiently expressed using a policy language is the Chinese wall. Consider an example of this scenario, in which a lawyer who looks at the set of documents of Company1 should not view the set of files of Company2 if Company1 and Company2 are in the same conflict-of-interest class. This lawyer may also view the files of Company3 provided that Company3 belongs to a different conflict-of-interest class than Company1 and Company2. Figure 3 shows a Venn Diagram for this situation. We can express a Chinese wall security policy using our language:

Access1 → {Module1,rw,(Range1 | Range3)}*;
Access2 → {Module1,rw,(Range1 | Range4)}*;
Access3 → {Module1,rw,(Range2 | Range3)}*;
Access4 → {Module1,rw,(Range2 | Range4)}*;
Policy → Access1 | Access2 | Access3 | Access4;

In the above policy, there are two conflict-of-interest classes. One contains Range1 and Range2, and the other contains Range3 and Range4. For simplicity, we have restricted this policy to one module. Figure 3 shows the DFA that recognizes legal accesses for this policy.

In general, for Chinese wall security policies, the number of states scales exponentially in the number of conflict-of-interest classes. This occurs because the number of possible legal accesses is the product of the number of sets in each conflict-of-interest class. The number of transitions also scales exponentially in the number of classes for the same reason. Fortunately, the number of states scales linearly in both the number of sets and the number of modules. Even better, the number of states is not affected by the number of ranges. The number of transitions scales linearly in the number of sets, ranges, and modules.
In order to prove that this policy specification has the property that Module1 will never be able to access both Range1 and Range2 or both Range3 and Range4, we first determine the language of illegal behavior:

\[
\begin{align*}
\text{Anything} & \rightarrow (\text{Range}_1 \mid \text{Range}_2 \mid \text{Range}_3 \mid \text{Range}_4 \mid \epsilon)^*; \\
\text{Access}_{\text{s}1} & \rightarrow \text{Anything} \text{Range}_1 \text{Anything} \text{Range}_2 \text{Anything}; \\
\text{Access}_{\text{s}2} & \rightarrow \text{Anything} \text{Range}_2 \text{Anything} \text{Range}_1 \text{Anything}; \\
\text{Access}_{\text{s}3} & \rightarrow \text{Anything} \text{Range}_3 \text{Anything} \text{Range}_4 \text{Anything}; \\
\text{Access}_{\text{s}4} & \rightarrow \text{Anything} \text{Range}_4 \text{Anything} \text{Range}_3 \text{Anything}; \\
\text{Illegal} & \rightarrow \text{Access}_{\text{s}1} \mid \text{Access}_{\text{s}2} \mid \text{Access}_{\text{s}3} \mid \text{Access}_{\text{s}4};
\end{align*}
\]

The DFA that recognizes illegal accesses should be identical to the complement of the DFA that recognizes legal accesses. This ensures that an illegal access will never be recognized by an enforcement mechanism with DFA logic that recognizes legal accesses.

### 4.4 Multilevel Security

Our security language can also be used to enforce instances of Multilevel Security (MLS) [35]. Military hardware such as avionics [42] may contain components with different clearance levels, and a component with a top secret clearance must not leak sensitive information to a component with a lower clearance [38]. Separation can form the basis for the enforcement of multilevel security. Figure 4 shows the architecture of such a MLS scenario. Module1 has a top secret (TS) clearance, and Module2 has an unclassified (U) clearance. Module1 and Module2 are initially compartmentalized, since they have different clearance levels. Therefore, Range1 belongs to Module1, and Range2 belongs to Module2. Module3 acts as a trusted server of information contained in the multilevel database, which contains both TS and U data. Therefore, the trusted server must have a security label range from U to TS. Range3 is temporary storage used for holding information that has just been retrieved from the database by Module3. Range4 (the control word) is used for performing database queries: a module writes to Range4 to request that Module3 retrieve some information from the database and then write the query result to Range3. If a request is made by Module1 for top secret information, it is necessary to revoke Module2’s read access to Module3, and this access must not be reinstated until Module3 zeroes out the sensitive information contained in Range3.

We express our MLS policy as follows:

\[
\begin{align*}
\text{rw} & \rightarrow r \mid w; \\
\text{Access}_{\text{s}2} & \rightarrow \{\text{Module}_{1,\text{rw},\text{Range}_1}\} \mid \{\text{Module}_{1,r,\text{Range}_3}\} \\
& \mid \{\text{Module}_{2,\text{rw},\text{Range}_2}\} \mid \{\text{Module}_{2,w,\text{Range}_4}\} \mid \{\text{Module}_{3,\text{rw},\text{Range}_3}\}; \\
\text{Access}_{\text{s}1} & \rightarrow \{\text{Module}_{2,\text{w},\text{Range}_3}\} \mid \text{Access}_{\text{s}2}; \\
\text{Trigger} & \rightarrow \{\text{Module}_{1,\text{w},\text{Range}_4}\}; \\
\text{Clear} & \rightarrow \{\text{Module}_{3,z,\text{Range}_3}\}; \\
\text{SteadyState} & \rightarrow (\text{Access}_{\text{s}2} \mid \text{Clear} \text{Access}_{\text{s}1} \text{*} \text{Trigger})^*; \\
\text{Policy} & \rightarrow \epsilon \mid \text{Access}_{\text{s}1}^* \mid \text{Access}_{\text{s}1}^* \text{Trigger} \text{SteadyState} \\
& \mid \text{Access}_{\text{s}1}^* \text{Trigger} \text{SteadyState} \text{Clear} \text{Access}_{\text{s}1}^*;
\end{align*}
\]

Figure 5 shows the DFA that recognizes this policy. State 1 corresponds to a less restrictive mode (Access$_1$), and State 0 corresponds to a more restrictive mode (Access$_2$). The Trigger event causes the state machine to transition from State 1 to State 0, and the Clear event causes the machine to transition from State 0 to State 1. In general, the DFA for a MLS policy will have one state for each access mode. For example, if we have three different modules that each have a different clearance level, there will be three access modes and three states. This example also demonstrates some of the weaknesses of our language. While the hardware to implement this technique is quite straightforward, the language based representation is hard to understand by a human. We discuss this and other problems in Section 6.
Figure 4: A Multilevel Security (MLS) architecture. Module1 has a Top Secret clearance, and Module2 has an Unclassified clearance. Module3 is a trusted server. Module3 performs a database query by writing to the control word (Range4). Before Module3 can write the query result to temporary storage (Range3), Module2’s permission to read from Range3 must be temporarily revoked because it only has an unclassified clearance, and this permission can only be restored after Module3 zeroes out the sensitive information in Range3.

Figure 5: This DFA recognizes legal behavior for a Multilevel Security (MLS) policy. State 1 corresponds to a less restrictive mode (Access1), and State 0 corresponds to a more restrictive mode (Access2).

Figure 6: DFA Transitions versus number of ranges. There is a linear relationship between the number of DFA transitions and the number of ranges for compartmentalization.

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, or is based on the fact 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 the data is sent 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 the write...
request is sent to memory. Thus, both architectures provide to the enforcement mechanism 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, the arbiters will allow one module to access the bus on even clock cycles and the other module to access the bus on odd clock cycles.

Our results quantify the efficiency of our policy compiler. First we discuss our methodology, followed by a discussion of the synthesis results.

5.2 Evaluation

Of the different policies we discussed in Section 4, we focus primarily on characterizing compartmentalization as this isolates 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 compartmentalization. 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 [29].

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\(^4\) 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.

\(^4\)the back-end handles netlist creation, placement, routing, and optimization for both timing and area
5.3 Synthesis Results

In general, a DFA for a compartmentalization policy always has exactly one state, and there is one transition for each \{\text{ModuleID}, \text{op}, \text{RangeID}\} tuple. Figure 6 shows that there is a linear relationship between the number of transitions and the number of ranges.

Figure 9 shows that the area of the resulting circuit scales linearly with the number of ranges for the compartmentalization policy. The slope is approximately four logic cells for every range. Figure 10 shows the cycle time \(T_{\text{clock}}\) for machines of various sizes, and Figure 11 shows the setup time \(T_{\text{su}}\), which is primarily the time to determine the range to which the input address belongs. \(T_{\text{clock}}\) is primarily the time for one DFA transition, and it is very close to the maximum frequency of this particular Altera Stratix device. Although \(T_{\text{clock}}\) is relatively stable, \(T_{\text{su}}\) increases linearly with the number of ranges. Fortunately, \(T_{\text{su}}\) can be reduced by pipelining the circuitry that determines what range contains the input address.

Figure 12 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 compartmentalization circuits above because the example policies have very few ranges. The complexity of the circuit is a combination of the number of ranges and the number of DFA states and transitions. Since the circuit for the Chinese wall policy has the most states, transitions, and ranges, it has the greatest area, followed by multilevel security, secure hand-off, access control list, and compartmentalization. Figure 13 shows that the cycle time is greatest for multilevel security, followed by compartmentalization, Chinese wall, secure hand-off, and access control list. Figure 14 shows that the setup time is greatest for multilevel security, followed by Chinese wall, compartmentalization, access control list, and secure hand-off.

6 Conclusions

Due to the increased use of reconfigurable logic in mission-critical applications, a new set of synthesizable security techniques 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 cannot be corrupted and is always invoked.

The formal access policy language provides a convenient and precise way to describe the fine-grained memory separation of modules on an FPGA. The flexibility of our language allows modules to communicate with each other securely by precisely transferring the privilege to access a buffer from one module to another. We have used our policy compiler to translate a variety of security policies to hardware enforcement modules, and we have analyzed the area and performance of these circuits. Our synthesis data show that the enforcement module is both efficient and scalable in the number of ranges that must be recognized.

We plan to enhance our technique by allowing information flow policies, which provide end-to-end security for sensitive data. In order to make this possible, it will be necessary to combine our execution monitor approach with static analysis techniques. This will make it easier for security administrators to develop security policies because the objects will be the actual data rather than ranges of memory. Arora et al. have proposed a scheme in which an embedded program first undergoes static analysis to determine legal behavior, and the results of this analysis are then used to build a hardware enforcement mechanism that monitors the application at run-time [4]. We would like to see if a similar approach will be successful for information flow policies.

We would also like to make it easier for security people to write more intuitive policies because usability is fundamental to system security [17] [12]. For example, a policy that is the conjunction of several complex regular expressions might not be the most intuitive way of expressing a much simpler idea, such as “Don’t let this module access top secret information.” We would like to use ideas from language theory to automatically translate these higher-level concepts into the appropriate regular expressions.
Figure 9: Circuit area versus number of ranges. There is a nearly linear relationship between the circuit area and the number of ranges.

Figure 10: Cycle time versus number of ranges. There is a nearly constant relationship between the cycle time and the number of ranges.

Figure 11: Setup time versus number of ranges. There is a nearly linear relationship between the setup time and the number of ranges. This time can be reduced by pipelining the logic that determines which range contains the input address.

Figure 12: Circuit area versus access policy. The area is related to the number of states, transitions, and ranges. The circuit area is greatest for the Chinese wall policy, which has the most states, transitions, and ranges, followed by multilevel security, secure hand-off, access control list, and compartmentalization.

Figure 13: Cycle time for each access policy. Cycle time is greatest for multilevel security, followed by compartmentalization, Chinese wall, secure hand-off, and access control list.

Figure 14: Setup time for each access policy. Setup time is greatest for multilevel security, followed by Chinese wall, compartmentalization, access control list, and secure hand-off.
References


ence on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, CA, October 2002.