Policy-Driven Memory Protection for Reconfigurable Hardware

Ted Huffmire, Shreyas Prasad, Tim Sherwood, and Ryan Kastner

University of California, Santa Barbara
Santa Barbara CA 93106, USA
{huffmire,sherwood}@cs.ucsb.edu
{shreyas,kastner}@ece.ucsb.edu
http://www.cs.ucsb.edu/~arch

Abstract. While processor based systems often 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 memory protection mechanism capable of enforcing 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. The efficiency of our access language design flow is evaluated in terms of area and cycle time across a variety of security scenarios.

Keywords: Computer Security, Embedded Systems, Reference Monitors, Separation Kernels, Security Policies, Policy Languages.

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 redaction.
– 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.

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 [4,6,8,15,20,29]. 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 [8]. 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 [5,7,33]. 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 [16]. To provide a security technique that can be used in practice, it must be both robust and efficient.

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 [22]. Segmented Memory [27] and Mondrian Memory Protection [35], 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 modules are active requires a new mechanism to ensure that the security properties of the system are enforced.

To provide separation in memory between multiple different interacting modules, we adapt some of the key concepts from separation kernels. Rushby originally proposed that a separation kernel [12] [18] [24] [25] 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. Unfortunately, separation kernels have high overhead and complexity due to the need to implement software virtualization, and the design complexity of modern out-of-order CPUs makes it difficult to implement separation kernels with a verifiable level of trust. A solution is needed that is located somewhere along a continuum between the two extremes of physical separation and software separation in order to have the best of both worlds.

We propose that the reconfigurable nature of FPGAs 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. We describe techniques we are working on to isolate the enforcement module in Section 5.2.

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 [32], 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 [30], 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 use the state of the enforcement mechanism as a covert channel. 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 5.2. The enforcement mechanism is also a Reference Validation Mechanism (RVM) [3]. Although Erlingsson et al. have proposed the idea of merging the reference monitor in-line with the target system [9], in a system with multiple interacting cores, this approach has the drawback that the reference monitors are distributed, which is problematic for stateful policies. Although there exist security policies that execution monitors are incapable of enforcing, such as *information flow* policies [26], 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. The *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 principals (M: modules), access rights (A: read, write, etc.), and objects (R: memory ranges[1]).

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 (M, A, R), and the universe of tuples forms an alphabet $\Sigma = M \times A \times R$. 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 $\forall xt \mid t \in \Sigma, xt \in L : x \in L$. 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 (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 $\exists s = s_0, s_1, ..., s_n \in L$ s.t. $\forall_{0 \leq i \leq n}$ 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 the first item through the use of a simple example. Consider a very straightforward compartmentalization policy. $Module_1$ 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]. Figure 2 shows this memory access policy expressed as a set of productions.

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 [19] 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

[1] An interval of the address space including high (R_{high}) and low (R_{low}) bounds.

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 a single 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 "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 specification language, 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 enforcement module. At a high level the technique partitions the module into two parts, range discovery and language recognition.

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:

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

Building and Transforming a Parse Tree – Next, we use Lex [17] and Yacc [14] to build a parse tree from our security policy. Internal nodes represent operators such as concatenation, alternation, and repetition. Figure 1 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 1 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

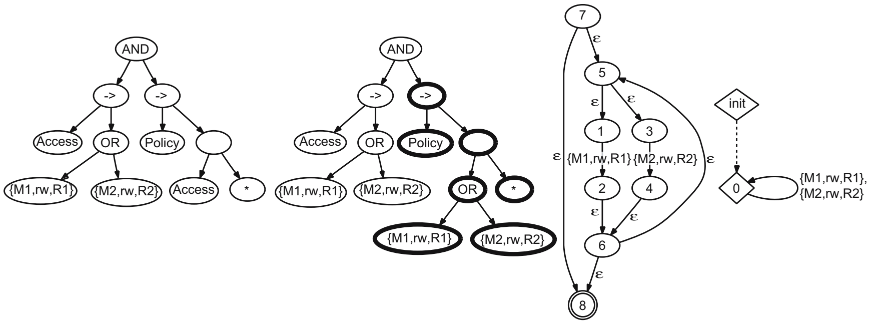

Fig. 1. Our policy compiler translates the policy to a regular expression by building, transforming, and traversing a parse tree. From the regular expression, an NFA is constructed, which is then converted into a minimized DFA.

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: $((\{Module_1, rw, Range_1\}) \mid (\{Module_2, rw, Range_2\}))*$.

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 1 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 [23] to minimize the DFA. Figure 1 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. 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.[2] 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 descrip-

[2] 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.

tion of state transitions into a hardware description language such as Verilog. Figure 3 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 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

Compartmentalization:

$rw \longrightarrow r \mid w;$

$Range_1 \longrightarrow [0x8e7b008, 0x8e7b00f];$

$Range_2 \longrightarrow [0x8e7b018, 0x8e7b01b];$

$Access_1 \longrightarrow \{Module_1, rw, Range_1\};$

$Access_2 \longrightarrow \{Module_2, rw, Range_2\};$

$Policy \longrightarrow (Access_1 \mid Access_2)^*;$

Access Control:

$Class_1 \longrightarrow Module_1 \mid Module_2;$

$Class_2 \longrightarrow Module_3 \mid Module_4;$

$List_1 \longrightarrow Class_1 \mid Class_2;$

$List_2 \longrightarrow Class_2;$

$Access_1 \longrightarrow \{List_1, rw, Range_1\};$

$Access_2 \longrightarrow \{List_2, rw, Range_2\};$

$Policy \longrightarrow (Access_1 \mid Access_2)^*;$

Secure Hand-Off:

$Module_{1|2} \longrightarrow Module_1 \mid Module_2;$

$Access_1 \longrightarrow \{Module_1, rw, Range_1\} \mid \{Module_{1|2}, rw, Range_2\};$

$Access_2 \longrightarrow \{Module_2, rw, (Range_1 | Range_2)\};$

$Trigger \longrightarrow \{Module_1, rw, Range_2\};$

$Policy \longrightarrow (Access_1^*)\ (\in \mid Trigger\ (Access_2)^*);$

Chinese Wall:

$Access_1 \longrightarrow \{Module_1, rw, (Range_1 \mid Range_3)\}^*;$

$Access_2 \longrightarrow \{Module_1, rw, (Range_1 \mid Range_4)\}^*;$

$Access_3 \longrightarrow \{Module_1, rw, (Range_2 \mid Range_3)\}^*;$

$Access_4 \longrightarrow \{Module_1, rw, (Range_2 \mid Range_4)\}^*;$

$Policy \longrightarrow Access_1 \mid Access_2 \mid Access_3 \mid Access_4;$

Redaction:

$rw \longrightarrow r \mid w;$

$Access_2 \longrightarrow \{Module_1, rw, Range_1\} \mid \{Module_1, r, Range_3\} \mid \{Module_2, rw, Range_2\} \mid \{Module_2, w, Range_4\} \mid \{Module_3, rw, Range_3\};$

$Access_1 \longrightarrow \{Module_2, r, Range_3\} \mid Access_2;$

$Trigger \longrightarrow \{Module_1, w, Range_4\};$

$Clear \longrightarrow \{Module_3, z, Range_3\};$

$SteadyState \longrightarrow (Access_2 \mid Clear\ Access_1^*\ Trigger)^*;$

$Policy \longrightarrow \in \mid Access_1^* \mid Access_1^*\ Trigger\ SteadyState \mid Access_1^*\ Trigger\ SteadyState\ Clear\ Access_1^*;$

Fig. 2. Several example policies expressed in our language

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, $Range_1$ 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 $Range_2$. Figure 2 shows this policy.

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 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 $Module_1$ wants to securely hand-off some data to $Module_2$. $Module_1$ writes some data to memory, to which it must have exclusive access, and then $Module_2$ 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 example policy in Figure 2. At first, $Module_1$ can access $Range_1$ or $Range_2$, and $Module_2$ can access $Range_2$. However, the first time $Module_1$ accesses $Range_2$ (indicating that $Module_1$ is ready to hand off), $Access_1$ 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 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 $Company_1$ should not view the set of files of $Company_2$ if $Company_1$ and $Company_2$ are in the same conflict-of-interest class. This lawyer may also view the files of $Company_3$ provided that $Company_3$ belongs to a different conflict-of-interest class than $Company_1$ and $Company_2$. Figure 2 shows a Chinese wall security policy expressed in our language. There are two conflict-of-interest 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.

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.

4.4 Redaction

Our security language can also be used to enforce instances of redaction [28], even at very high throughputs (such as for video). Military hardware such as avionics [34] 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 [31]. Figure 5 shows the architecture of a redaction scenario that is based on separation. A multilevel database contains both Top Secret and Unclassified data. $Module_1$ has a top secret (TS) clearance, but $Module_2$ has an unclassified (U) clearance. $Module_1$ and $Module_2$ are initially compartmentalized, since they have different clearance levels. 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 temporary storage. Any database query performed 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 temporary storage. Figure 2 shows our redaction policy, and Figure 4 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 redaction 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.

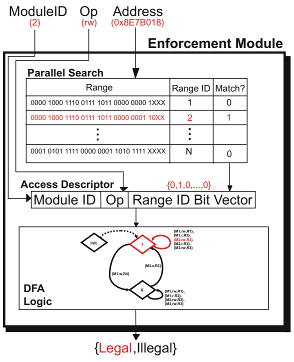

Fig. 3. A hardware reference monitor

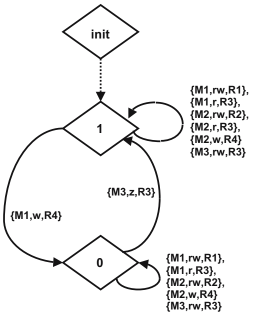
init
1
0
{M1,rw,R1}, {M1,r,R3}, {M2,rw,R2}, {M2,r,R3}, {M2,w,R4} {M3,rw,R3}
{M3,z,R3}
{M1,w,R4}
{M1,rw,R1}, {M1,r,R3}, {M2,rw,R2}, {M2,w,R4} {M3,rw,R3}

Fig. 4. DFA recognizing legal behavior for a redaction policy

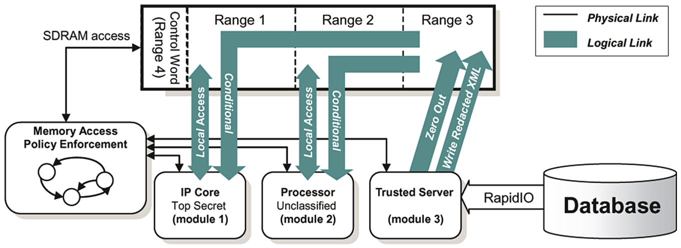

Fig. 5. A redaction architecture. IP stands for Intellectual Property.

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 6 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 (E) 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 (B) 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.

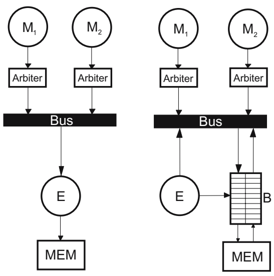

Fig. 6. Two alternative architectures for the enforcement mechanism

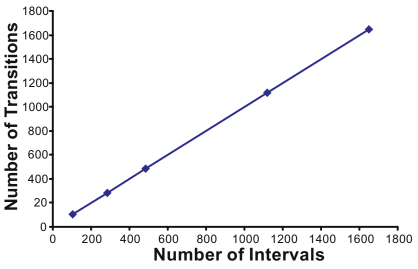

Fig. 7. DFA Transitions versus number of ranges for compartmentalization

5.2 Isolation of the Reference Monitor

It is critical that the reference module be isolated from other modules on the FPGA. Ensuring the physical separation of the modules entails distributing the computation spatially. We are working on methods to ensure that modules are placed in separate spatial areas and that there are no extraneous connections between the modules. Although we are working on addressing this problem by developing techniques that work at the gate level, this work is beyond the scope of this paper. In our attack model, there may be malicious modules or remote attacks that originate from the network, but we assume that the attacker does not have physical access to the device.

5.3 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 [21].

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[3] 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.

5.4 Synthesis Results

In general, a DFA for a compartmentalization policy always has exactly one state, and there is one transition for each $\{ModuleID, op, RangeID\}$ tuple. Figure 7 shows that there is a linear relationship between the number of transitions and the number of ranges.

Figure 8 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 9 shows the cycle time (T_{clock}) for machines of various sizes, and Figure 10 shows the setup time (T_{su}), which is primarily the time to determine the range to which the input address belongs. 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. Although T_{clock} is relatively stable, T_{su} increases linearly with the number of ranges. Fortunately, T_{su} can be reduced by pipelining the circuitry that determines what range contains the input address.

Figure 11 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 redaction, secure hand-off, access control list, and compartmentalization. Figure 12 shows that the cycle time is greatest for redaction, followed by compartmentalization, Chinese wall, secure hand-off, and access control list. Figure 13 shows that the setup time is greatest for redaction, followed by Chinese wall, compartmentalization, access control list, and secure hand-off.

5.5 Impact of the Reference Monitor on System Performance

FPGAs do not operate at high frequency. Since they operate at a lower frequency, they achieve their performance from spatial parallelism. FPGA applications such as DSPs, signal processing, and intrusion detection systems are

[3] The back-end handles netlist creation, placement, routing, and optimization for both timing and area.

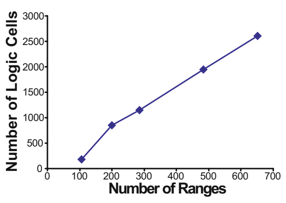

Fig. 8. Circuit area versus number of ranges. There is a nearly linear relationship between the circuit area and the number of ranges.

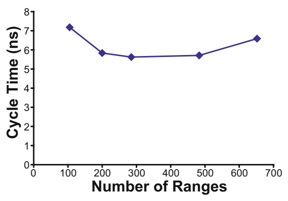

Fig. 9. Cycle time versus number of ranges. There is a nearly constant relationship between the cycle time and the number of ranges.

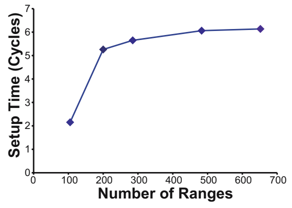

Fig. 10. Setup time versus number of ranges. There is a nearly linear relationship between the setup time and the number of ranges.

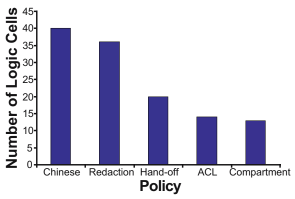

Fig. 11. Circuit area versus access policy. The area is related to the number of states, transitions, and ranges. The circuit area is greatest for the Chinese wall policy.

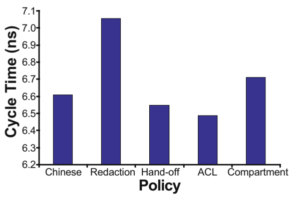

Fig. 12. Cycle time for each access policy. Cycle time is greatest for redaction, followed by compartmentalization, Chinese wall, secure hand-off, and access control list.

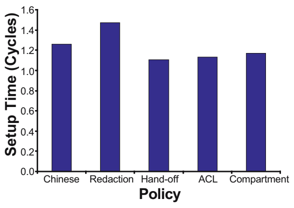

Fig. 13. Setup time for each access policy. Setup time is greatest for redaction, followed by Chinese wall, compartmentalization, access control list, and secure hand-off.

throughput-driven and therefore are latency-insensitive. These applications are designed using careful scheduling and pipelining techniques. For these reasons, we argue that our technique does not impact the 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 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 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 is invoked for every memory access, and we are currently developing gate-level techniques to ensure the physical isolation of the policy module.

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. In addition to the reconfigurable domain, our methods can be applied to systems-on-a-chip as part of a more general scheme.

Since usability is fundamental to system security [13] [11], we plan to provide an incremental method of constructing mathematically precise policies by building on the policy engineering work of Fong et al. [10]. In a correctly formed policy, there should be no intersection between legal and illegal behavior. Our tools will allow a policy engineer to check whether there is any conflict between a policy under construction that specifies legal behavior and a specific instance of behavior that is known to be illegal. If a conflict exists, the tool will inform the policy engineer of the exact problem that needs to be fixed.

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. K. Bondalapati and V.K. Prasanna. Reconfigurable computing systems. In *Proceedings of the IEEE*, volume 90(7), pages 1201–17, 2002.
5. D.A. Buell and K.L. Pocek. Custom computing machines: an introduction. In *Journal of Supercomputing*, volume 9(3), pages 219–29, 1995.
6. 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.
7. 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.
8. 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.
9. Ulfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: A retrospective. In *Proceedings of the 1999 Workshop on New Security Paradigms*, 1999.
10. Philip W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, 2004.

11. Peter Gutmann and Ian Grigg. Security usability. *IEEE Security and Privacy Magazine*, July/August 2005.
12. 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.
13. 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.
14. S. Johnson. Yacc: Yet another compiler-compiler. Technical Report CSTR-32, Bell Laboratories, Murray Hill, NJ, 1975.
15. Ryan Kastner, Adam Kaplan, and Majid Sarrafzadeh. *Synthesis Techniques and Optimizations for Reconfigurable Systems*. Kluwer Academic, Boston, MA, 2004.
16. 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.
17. M. Lesk and E. Schmidt. Lex: A lexical analyzer generator. Technical Report 39, Bell Laboratories, Murray Hill, NJ, October 1975.
18. 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.
19. Peter Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett, Sudbury, MA, 2001.
20. 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 computing. In *Computer*, volume 30(12), pages 38–43, 1997.
21. Giovanii De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.
22. 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.
23. D. Raymond and D. Wood. Grail: A C++ library for automata and expressions. *Journal of Symbolic Computation*, 11:341–350, 1995.
24. John Rushby. Design and verification of secure systems. *ACM Operating Systems Review*, 15(5):12–21, December 1981.
25. John Rushby. A trusted computing base for embedded systems. In *Proceedings 7th DoD/NBS Computer Security Conference*, pages 294–311, September 1984.
26. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.
27. J. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, July 1974.
28. O. Sami Saydjari. Multilevel security: Reprise. *IEEE Security and Privacy Magazine*, September/October 2004.
29. 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.
30. Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1), February 2000.

31. Richard E. Smith. Cost profile of a highly assured, secure operating system. In *ACM Transactions on Information and System Security*, 2001.
32. 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.
33. 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.
34. 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.
35. 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.