Designing Secure Systems on Reconfigurable Hardware

TED HUFFMIRE
Naval Postgraduate School
BRETT BROTHERTON
Special Technologies Laboratory
NICK CALLEGARI, JONATHAN VALAMEHR, and JEFF WHITE
University of California, Santa Barbara
RYAN KASTNER
University of California, San Diego
and
TIM SHERWOOD
University of California, Santa Barbara

The extremely high cost of custom ASIC fabrication makes FPGAs an attractive alternative for deployment of custom hardware. Embedded systems based on reconfigurable hardware integrate many functions onto a single device. Since embedded designers often have no choice but to use soft IP cores obtained from third parties, the cores operate at different trust levels, resulting in mixed-trust designs. The goal of this project is to evaluate recently proposed security primitives for reconfigurable hardware by building a real embedded system with several cores on a single FPGA and implementing these primitives on the system. Overcoming the practical problems of integrating multiple cores together with security mechanisms will help us to develop realistic security-policy specifications that drive enforcement mechanisms on embedded systems.

44

---

1. INTRODUCTION

Reconfigurable hardware, such as a field programmable gate array (FPGA), provides an attractive alternative to costly custom ASIC fabrication for deploying custom hardware. While ASIC fabrication requires very high nonrecurring engineering (NRE) costs, an SRAM-based FPGA can be programmed after fabrication to be virtually any circuit. Moreover, the configuration can be updated an infinite number of times.

Because they are able to provide a useful balance between performance, cost, and flexibility, many critical embedded systems make use of FPGAs as their primary source of computation. For example, the aerospace industry relies on FPGAs to control everything from the Joint Strike Fighter to the Mars Rover. We are now seeing an explosion of reconfigurable-hardware-based designs in everything from face recognition systems [Ngo et al. 2005], to wireless networks [Salefski and Caglar 2001], intrusion detection systems [Hutchings et al. 2002], and supercomputers [Bondhugula et al. 2006]. In fact, it is estimated that in 2005 alone there were over 80,000 different commercial FPGA design projects started [McGrath 2005].

Since major IC manufacturers outsource most of their operations to a variety of countries [Milanowski and Maurer 2006], the theft of IP from a foundry is a serious concern. FPGAs provide a viable solution to this problem, since the sensitive IP is not loaded onto the device until after it has been manufactured and delivered. This makes it harder for the adversary to target a specific application or user. In addition, device attacks are difficult on an FPGA, since the intellectual property is lost when the device is powered off. Modern FPGAs use bit-stream encryption and other methods to protect the intellectual property once it is loaded onto the FPGA or an external memory.

Although FPGAs are currently fielded in critical applications that are part of the national infrastructure, the development of security primitives for FPGAs is just beginning. Reconfigurable systems are often composed of several modules (called IP cores) on a single device. Since cost pressures necessitate

object reuse, a typical embedded system will incorporate soft IP cores that have been developed by third parties. Just as software designers must rely on third-party classes, libraries, and compilers, hardware designers must also cope with the reality of using third-party cores and design tools. This issue will grow in importance as organizations increasingly rely on incorporating commercial off-the-shelf (COTS) hardware or software into critical projects.

The goal of this article is to examine the practicality of recently proposed security primitives for reconfigurable hardware [Huffmire et al. 2007, 2006], by applying them to an embedded system consisting of multiple cores on a single FPGA. Through our red-black design example, we attempt to better understand how the application of these security primitives impacts the design, in terms of both complexity and performance, of a real system. Integrating several cores together with reconfigurable protection primitives enables the effective implementation of realistic security policies on practical embedded systems. We begin with a description of work related to reconfigurable security (Section 2), and then explain the underlying theory of separation in reconfigurable devices in Section 3. We then present our design example, a red-black system, and discuss how to secure this design through the application of moats, drawbridges, and reference monitors in Section 4.

2. RELATED WORK

While there is a large body of work relating to reconfigurable devices and their application to security, we can broadly classify the work related to *securing* reconfigurable designs into three broad categories: IP theft prevention; isolation and protection; and covert channels.

2.1 IP Theft

Most of the work relating to FPGA security targets the problem of preventing the theft of intellectual property and securely uploading bit-streams in the field, which is orthogonal to our work. Since such theft directly impacts their bottom line, industry has already developed several techniques to combat the theft of FPGA IP, such as encryption [Bossuet et al. 2004; Kean 2002, 2001], fingerprinting [Lach et al. 1999a], and watermarking [Lach et al. 1999b]. However, establishing a root of trust on a fielded device is challenging because it requires a decryption key to be incorporated into the finished product. Some FPGAs can be remotely updated in the field, and industry has devised secure hardware-update channels that use authentication mechanisms to prevent a subverted bit-stream from being uploaded [Harper et al. 2003; Harper and Athanas 2004]. These techniques were developed to prevent an attacker from uploading a malicious design that causes unintended functionality. Even worse, the malicious design could physically destroy the FPGA by causing the device to short-circuit [Hadzic et al. 1999].

2.2 Isolation and Protection

Besides our previous work [Huffmire et al. 2007, 2006], there is very little other work on the specifics of managing FPGA resources in a secure manner. Chien

and Byun have perhaps the closest work, where they addressed the safety and protection concerns of enhancing a CMOS processor with reconfigurable logic [Chien and Byun 1999]. Their design achieves process isolation by providing a reconfigurable virtual machine to each process, and their architecture uses hardwired translation look-aside buffers (TLBs) to check all memory accesses. Our work could be used in conjunction with theirs, using soft-processor cores on top of commercial off-the-shelf FPGAs, rather than a custom silicon platform. In fact, we believe one of the strong points of our work is that it may provide a viable implementation path to those that require a custom secure architecture, for example, execute-only memory [Lie et al. 2000] or virtual secure coprocessing [Lee et al. 2005].

A similar concept to moats and drawbridges is discussed in McLean and Moore [2007]. Though they do not provide great detail about much of their work, they use a similar technique to isolate regions of the chip by placing a buffer between them, which they call a fence. Gogniat et al. propose a method of embedded system design that implements security primitives such as AES encryption on an FPGA, which is one component of a secure embedded system containing memory, I/O, CPU, and other ASIC components [Gogniat et al. 2006]. Their security-primitive controller (SPC), which is separate from the FPGA, can dynamically modify these primitives at runtime in response to the detection of abnormal activity (attacks). In this work, the reconfigurable nature of the FPGA is used to adapt a crypto-core to situational concerns, although the concentration is on how to use an FPGA to help efficiently thwart system-level attacks, rather than chip-level concerns. Indeed, FPGAs are a natural platform for performing many cryptographic functions because of the large number of bit-level operations that are required in modern block ciphers. However, while there is a great deal of work centered around exploiting FPGAs to speed cryptographic or intrusion detection primitives, systems researchers are just now starting to realize the security ramifications of building systems around hardware which is reconfigurable.

2.3 Memory Protection on an FPGA

On a modern FPGA, the memory is essentially flat and unprotected by hardware mechanisms because reconfigurable architectures on the market today support a simple linear addressing of the physical memory. On a general-purpose processor, interaction via shared memory can be controlled through the use of page tables and associated TLB attributes. 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. Use of superpages, which are very large memory pages, makes it possible for the TLB to have a lower miss rate [Navarro et al. 2002]. Segmented memory [Saltzer 1974] and Mondrian memory protection [Witchel et al. 2002], a finer-grained scheme, address the inefficiency of providing per-process memory protection via global attributes by associating each process with distinct permissions on the same memory region.

2.4 Covert Channels, Direct Channels, and Trap Doors

Although moats provide physical isolation of cores, it is possible that cores could still communicate via a covert channel. In a covert-channel attack, classified information flows from a "high" core to a "low" core that should not access classified data. Covert channels work via an internal shared resource, such as processor activity, disk usage, or error conditions [Percival 2005]. There are two types of covert channel: storage channels and timing channels. Classical covert-channel analysis involves articulation of all shared resources on the chip, identifying the share points, determining whether the shared resource is exploitable, and determining the bandwidth of the covert channel as well as whether remedial action can be taken [Kemmerer 1983; Millen 1987]. Storage channels can be mitigated by partitioning the resources, while timing channels can be mitigated with sequential access. Examples of remedial action include decreasing the bandwidth (e.g., the introduction of artificial spikes (noise) in resource usage [Saputra et al. 2003]) or closing the channel. Unfortunately, an adversary can extract a signal from the noise, given sufficient resources [Millen 1987].

A slightly different type of attack is the side-channel attack, such as a power-analysis attack on a cryptographic system, which can extract the keys used by a crypto-core [Kocher et al. 1999; Standaert et al. 2003]. Finally, there are overt channels (i.e., trap doors or direct channels) [Thompson 1984]. One example of a direct channel is a system that lacks memory protection: A core simply writes data to a chunk of memory, and another core reads it. Another example of a direct channel is a tap that connects two cores. An unintentional tap is a direct channel that can be established due to implementation errors, faulty design, or malicious intent. For example, the place-and-route tool's optimization strategy may interleave the wires of two cores. Although the chances of this are small, CAD tools are not perfect and errors do occur. Much greater is the threat of designer errors, incorrect implementation of the specification, and malicious code or logic. We leave to future work the development of automated methods of detecting covert, side, and direct channels in embedded designs.

3. MOATS, DRAWBRIDGES, AND REFERENCE MONITORS

3.1 Motivation for Isolation and Separation

The concept of isolation is fundamental to computer security. Saltzer and Shroeder use diamonds as a metaphor for sensitive data [Saltzer and Schroeder 1974]. To protect the diamonds, you must isolate them by placing them in a vault. To access the diamonds, you must have a method of controlled sharing (a vault door with a combination lock). The term *separation* describes the controlled sharing of isolated objects. In a system with a mandatory access control (MAC) policy, objects may belong to different *equivalence classes*, such as classified and unclassified. Therefore, we must isolate the various equivalence classes and control their interaction.

Isolation and separation are crucial to the design of military avionics, which are designed in a federated manner so that a failure of one component (e.g.,
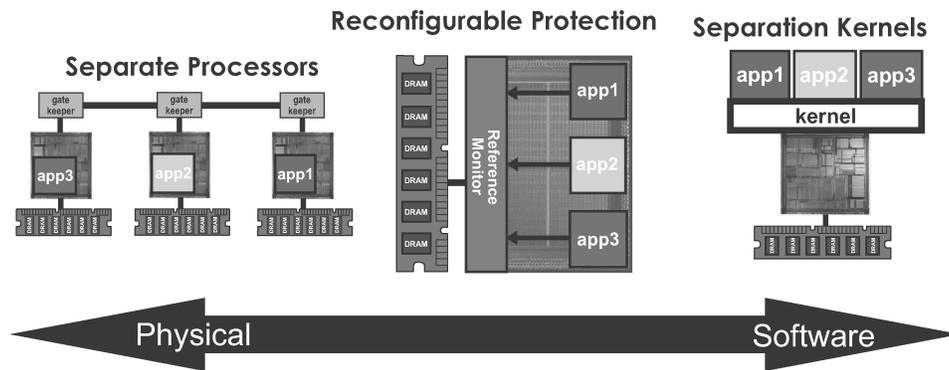
Fig. 1. Alternative strategies for providing protection on embedded systems. From a security standpoint, a system with multiple applications could allocate a dedicated physical device for each application, but economic realities force designers to integrate multiple applications onto a single device. Separation kernels use virtualization to prevent applications from interfering with each other, but these come with the overhead of software and are therefore restricted to general-purpose processor-based systems. The goal of this project is to evaluate reconfigurable isolation and controlled sharing mechanisms that provide separation for FPGA-based embedded systems.

by the enemy's bullet) is contained [Rushby 2000]. Since having a separate device for each function incurs a high cost in terms of weight, power, cooling, and maintenance, multiple functions must be integrated onto a single device without interfering with each other. Therefore, avionics was the drive behind the development of the first separation kernels [Rushby 1984]. In military-avionics systems, sensitive targeting data is processed on the same device as unclassified maintenance data, and keeping processing elements that are "cleared" for different levels of data, properly separated, is critical [Weissman 2003].

Separation and isolation are also fundamental to the design of cryptographic devices. In a red-black system, plaintext carried over red wires must be segregated from ciphertext carried over black wires, and the NSA has established requirements for the minimum distance and shielding between red and black circuits, components, equipment, and systems [National Security Telecommunications and Information Systems Security Committee 1995]. We extend the red-black concept in this article to an embedded system-on-a-chip with a red domain and a black domain.

3.2 Mechanisms for Isolation and Separation

One option for providing separation in embedded systems is purely physical separation, shown in the left of Figure 1. With physical separation, each application runs on its own dedicated device, and gate keepers provide a mechanism of controlled interaction between applications. Requiring a separate device for each application is very expensive and therefore impractical for embedded systems. In contrast to strictly physical protection, separation kernels [Rushby 1984; Irvine et al. 2004; Levin et al. 2004] use software virtualization to prevent applications from interfering with each other. A separation kernel, shown in the right of Figure 1, provides isolation of applications but also facilitates their

controlled interaction. However, separation kernels come with the overhead of software and can only run on general-purpose processors.

Reference monitors. In our prior work we proposed a third approach, called reconfigurable protection [Huffmire et al. 2006] and shown in the middle of Figure 1, that uses a reconfigurable reference monitor to enforce the legal sharing of memory among cores. A memory-access policy is expressed in a specialized language, and a compiler translates this policy directly to a circuit that enforces the policy. The circuit is then loaded onto the FPGA along with the cores. The benefit of using a language-based design flow is that a design change that affects the policy simply requires a modification to the policy specification, from which a new reference monitor can be automatically generated.

Moats and drawbridges. In our prior work [Huffmire et al. 2007], we proposed a spatial isolation mechanism called a moat and a controlled sharing mechanism called a drawbridge as methods for ensuring separation on reconfigurable devices. Moats exploit the spatial nature of computation on FPGAs to provide strong isolation of cores. A moat surrounds a core with a channel in which routing is disabled. In addition to isolation of cores, moats can also be used to isolate the reference monitor and provide tamper resistance. Drawbridges allow signals to cross moats, letting the cores communicate with the outside world. Finally, static analysis of the bit-stream is used to ensure that only specified connections between cores can be established. This analysis can also be used to ensure that the reference monitor cannot be bypassed and is always invoked.

4. AN APPLICATION OF SEPARATION THROUGH DESIGN

To test the practicality of moats, drawbridges, and reference monitors, we need to apply them to a real design. Our test system is a red-black system running on a single FPGA device. As discussed in Section 3, the red and black components must be separated. We will use two types of separation in our design: spatial separation using moats and drawbridges, and temporal separation using a reference monitor. The combination of moats, drawbridges, and a reference monitor allows us to develop a more secure system that can run on a single device and make use of shared resources to conserve power, cost, and area. Our design allows us to gain further knowledge about the ease of design and about performance in applying these mechanisms to a real system.

4.1 Red-Black System: A Design Example

The system we designed is a multicore system-on-a-chip which can be seen in Figure 3. There are two μBlaze processors in the system: one belongs to the red domain, and the other to the black. These processors communicate with the memory and various peripherals over a shared bus. A traditional shared bus is insecure because there is nothing to prevent one processor from reading the other processor's memory or accessing information from a peripheral that it is not supposed to access. To address this problem, the reference monitor was integrated into the on-chip peripheral bus (OPB), so that all bus accesses by the two processors must be verified by the reference monitor.

ModuleID Op Address
(2) (rw) (0x8E7B018)

Enforcement Module

Parallel Search

Range	Range ID	Match?
0000 1000 1110 0111 1011 0000 0000 1XXX	1	0
0000 1000 1110 0111 1011 0000 0001 10XX	2	1
⋮	⋮	
0001 0101 1111 0000 0001 1010 1111 XXXX	N	0

Access Descriptor {0,1,0,...,0}

| Module ID | Op | Range | ID | Bit | Vector |

init

1 {M1,rw,R1},
{M1,r,R3},
{M2,rw,R2},
{M2,r,R3},
{M3,rw,R3}

{M3,z,R3}

{M1,w,R4}

DFA Logic

0 {M1,rw,R1},
{M1,r,R3},
{M2,rw,R2},
{M3,rw,R3}

{Legal,Illegal}

Fig. 2. The inputs to the reference monitor are the module ID, op, and address. The range ID is determined by performing a parallel search over all ranges, similar to a content-addressable memory (CAM). The module ID, op, and range ID together form an access descriptor, which is the input to the state-machine logic. The output is a single bit: either grant or deny the access. Moats and drawbridges ensure that the reference monitor is tamper-proof and always invoked.

The design consists of seven different "cores": We have μBlaze$_0$, μBlaze$_1$, the OPB (along with its arbiter and the reference monitor), the AES core, DDR SDRAM, the RS-232 interface, and the Ethernet interface. These components share resources and interact with one another. The on-chip peripheral bus (OPB) was modified to create a custom OPB containing a reference monitor which must approve all memory accesses. Shared external memory (SDRAM),

Fig. 3. System architecture. The system is divided into two isolation domains to prevent the mixing of data of different sensitivity levels. The first domain (shown as a hatched pattern) contains μBlaze$_0$ and the local RS-232 interface, which can be connected to an iris or fingerprint scanner. The second domain (white) contains μBlaze$_1$ and the E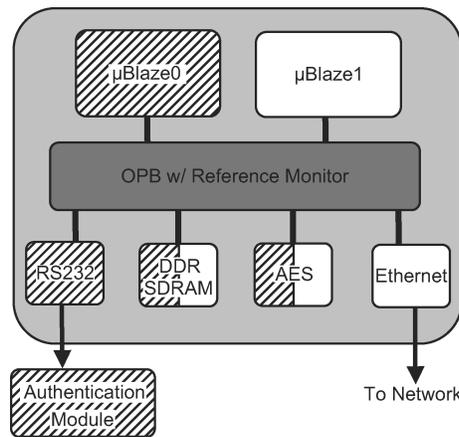thernet interface. Both processors share the AES core and external memory, and the reference monitor enforces the sharing of these resources as well as the isolation of domains.

the AES core, RS-232 interface, and Ethernet interface are also connected to the bus as slave devices, so access to these devices must go through the reference monitor. These seven different cores are then physically partitioned using moats and drawbridges.

The integration of the reference monitor into the OPB allows for ease of system design. This custom OPB is available to incorporate into any system using the Xilinx Platform Studio. The OPB is the most commonly used bus to connect the peripherals together in a system, so adding a reference monitor to a new system design is as simple as "dragging and dropping" the custom OPB into the design.

The AES core has a custom-designed controller that allows it to be controlled through shared memory. When a processor wants to encrypt or decrypt data, the processor places this data in the shared memory and writes several control words to indicate to the AES core what operation to perform, where the data is located, and how much data there is. When the AES core is done performing the requested operation, it signals the processor, and the processor can then retrieve the data from the shared memory buffer. The shared memory buffer allows the AES core to work like a coprocessor, freeing up the regular processor to perform other tasks while encryption/decryption is being performed.

In order to allow both processors to use the AES core, access to it is strictly regulated by our stateful security policy in the reference monitor. The shared memory buffer is divided into two parts, one for each processor. This keeps each processor's data separate and prevents one processor from reading data that has been decrypted by the other, but this does not solve the problem of regulating access to the core. Access to the core is controlled by restricting

access to the control words, and this will be discussed in further detail in the following sections.

All these components form our red-black system, which has two isolation domains. The red domain, shown in Figure 3 with a hatched pattern, consists of its own region of memory in the SDRAM and AES core, the RS232 interface along with the authentication module, and μBlaze$_0$. We currently have a Secugen fingerprint reader; however, other authentication methods such as retinal scanning or voice recognition could also be used. The second isolation domain (the black domain) is shown with no pattern in Figure 3 and consists of its own region of memory in the SDRAM and AES core, the Ethernet interface, and μBlaze$_1$. Since the Ethernet can be connected to the much less secure Internet, it is isolated from the red part of the system, which handles sensitive and authentication data. This separation is achieved through the use of moats, drawbridges, and a reference monitor.

4.2 A Reference Monitor

Commonly implemented in software, a reference monitor is used to control access to data or devices in a system. In our system, the reference monitor is implemented in hardware and used to regulate access to the memory and peripherals. When a core makes a request to access memory, the reference monitor (RM) makes a decision to either allow the access or deny it. The RM can provide protection for any device connected to the OPB. For example, a MicroBlaze CPU and an AES-encryption core can share a block of BRAM. The CPU encrypts plaintext by copying it to the BRAM and then signaling to the AES core via a control word. The AES core retrieves the plaintext from the BRAM and encrypts the plaintext using a symmetric key. After encrypting the plaintext, the AES core places the ciphertext into the BRAM and then signals to the CPU via another control word. Finally, the CPU retrieves the ciphertext from the BRAM. A similar process is used for decryption. A simple memory-access policy can be constructed with two states: one that gives the CPU exclusive access to the shared control buffer and another that gives the AES core exclusive access to the control buffer. Transitions between these two states occur when the cores signal to the reference monitor via performing a write to a reserved address. We extend this idea to construct a policy which will be applied to our red-black system, consisting of three states for a system with two CPU cores, a shared AES core, and shared external memory.

Typically, the different cores are connected to the memory and peripherals through a shared bus. This bus (OPB) can connect the CPU, external DRAM, RS232 (serial port), general-purpose I/O (to access the external pins), shared BRAM, and DMA. To prevent two cores from utilizing the bus at the same time, an arbiter sits between the modules and the bus. The reference monitor can be placed between the bus and the memory, or the reference monitor can snoop on the bus. Our goal is to make sure that our memory protection primitive achieves efficient memory-system performance. This will also be an opportunity to design meaningful policies for systems that employ a shared bus.

4.2.1 *A Hardware Implementation.* Figure 2 shows the hardware decision module we wish to build. An *access descriptor* specifies the allowed accesses between a module and a range. Each DFA transition represents an access descriptor consisting of a module ID, an op, and a range ID bit-vector. The range ID bit-vector contains a bit for each possible range, and the descriptor's range is indicated by the (one) bit that is set.

A *memory-access* request consists of three inputs: the module ID, the op {read, write, etc.}, and the address. The output is a single bit: 1 for grant and 0 for deny. First, the hardware converts the memory-access address to a bit-vector. To do this, it checks all the ranges in parallel and sets that bit corresponding to the range ID that contains the input address (if any). Then, the memory-access request is processed through the DFA. If an access descriptor matches the access request, the DFA transitions to the accept state and outputs a 1.

By means of the reference monitor, the system is divided into two systems which are isolated, yet share resources. The first system consists of μBlaze$_0$, the DDR SDRAM, and the RS-232 device. The second system consists of μBlaze$_1$, the DDR SDRAM, and the Ethernet device. Everything is interconnected with the OPB (on-board peripheral bus), which is the glue for the systems, and both systems make use of the AES core as well.

These two different systems save on power and area by sharing resources (the bus and the AES core); however, this can be a problem if we want to isolate the two systems. The Ethernet interface could be connected to the Internet, which has a lower security level than the RS-232 interface, which is a local connection. We want to prevent the mixing of data of different security levels. First, we assign a processor to each communication interface. Using the OPB that is provided with the EDK allows for both processors to share the peripherals but is very insecure, since they would have unregulated access to all regions of memory and all peripherals on the bus. Also, there is the issue of arbitrating access and preventing the mixing of data of different sensitivity levels in the shared AES core.

The reference monitor, which is integrated into the OPB, addresses these problems. Since we are using memory-mapped I/O, the reference monitor allows us to control access to the two I/O devices and to split the shared DDR SDRAM into two isolated blocks, one for each processor. In this way we restrict access so that each processor can access only that I/O device which is intended for its use. Access to the AES core is arbitrated by having multiple states in our memory-access policy. Our system can regulate access to any of the slave devices on the bus with little overhead. Furthermore, the system can easily be scaled to add more masters, and the policy implemented by the reference monitor can easily be modified.

4.2.2 *A Security-Policy Design Example.* While the reference monitor cannot be bypassed and can control access to all peripherals, it is useless without a good security policy. Our system makes use of a simple stateful policy to control access to the peripherals and to allow the sharing of the AES core. We will describe this policy and how it is transformed into a hardware reference monitor that can easily be added to any design.
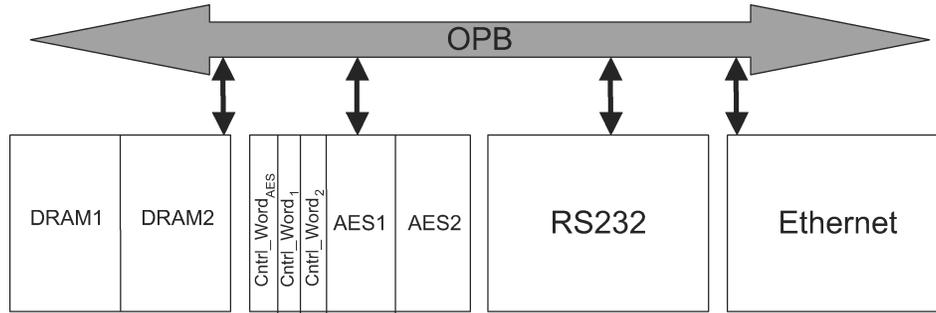
Fig. 4.   This diagram shows how the memory and different memory-mapped I/O devices are divided into regions by the reference monitor.

The designer expresses the access policy in our specialized language. The access policy consists of three states: one state for the case in which μBlaze_0 (or Module_1) has access to the AES core, one state for the case where μBlaze_1 (or Module_2) has access to the AES core, and one state for the case where neither has access to the AES core. A processor obtains access to the AES core by writing to a specific control word (Control Word 1), and a processor relinquishes access to the AES core by writing to another specific control word (Control Word 2). Therefore, the transitions between states occur when one of the processors writes to one of these specified control words.

In addition to permitting temporal sharing of the AES core, the policy isolates the two MicroBlaze processors such that Processor_1 and RS-232 data items are in a separate isolation domain from Processor_2 and Ethernet data items. Since each component of our system is assigned a specific address range, our reference monitor is well suited for enforcing a resource-sharing policy. We specify the policy for our system as follows. The first part of the policy specifies the ranges (a graphical depiction of the ranges can be seen in Figure 4).

$Range_1 \rightarrow$ [0x28000010,0x28000777]; (AES1)
$Range_2 \rightarrow$ [0x28000800,0x28000fff]; (AES2)
$Range_3 \rightarrow$ [0x24000000,0x24777777]; (DRAM1)
$Range_4 \rightarrow$ [0x24800000,0x24ffffff]; (DRAM2)
$Range_5 \rightarrow$ [0x40600000,0x4060ffff]; (RS-232)
$Range_6 \rightarrow$ [0x40c00000,0x40c0ffff]; (Ethernet)
$Range_7 \rightarrow$ [0x28000004,0x28000007]; ($Ctrl\_Word_1$)
$Range_8 \rightarrow$ [0x28000008,0x2800000f]; ($Ctrl\_Word_2$)
$Range_9 \rightarrow$ [0x28000000,0x28000003]; ($Ctrl\_Word_{AES}$)

The second part of the policy specifies the different access modes, one for each state.

$Access_0 \rightarrow \{Module_1, rw, Range_5\}$
    $| \{Module_2, rw, Range_6\}$
    $| \{Module_1, rw, Range_3\}$
    $| \{Module_2, rw, Range_4\}$

$Access_1 \rightarrow Access_0$
$| \{Module_1, rw, Range_1\}$
$| \{Module_1, rw, Range_9\};$
$Access_2 \rightarrow Access_0$
$| \{Module_2, rw, Range_2\}$
$| \{Module_2, rw, Range_9\};$

The third part of the policy specifies the transitions between states.

$Trigger_1 \rightarrow \{Module_1, w, Range_7\};$
$Trigger_2 \rightarrow \{Module_1, w, Range_8\};$
$Trigger_3 \rightarrow \{Module_2, w, Range_7\};$
$Trigger_4 \rightarrow \{Module_2, w, Range_8\};$

The final part of the policy uses regular expressions to specify the structure of the policy's state machine.

$Expr_1 \rightarrow Access_0 \mid Trigger_3\ Access_2{}^*\ Trigger_4;$
$Expr_2 \rightarrow Access_1 \mid Trigger_2\ Expr_1{}^*\ Trigger_1;$
$Expr_3 \rightarrow Expr_1{}^*\ Trigger_1\ Expr_2{}^*;$
$Policy \rightarrow Expr_1{}^* \mid Expr_1{}^*\ Trigger_3\ Access_2{}^*$
$\quad \mid Expr_3\ Trigger_2\ Expr_1{}^*\ Trigger_3\ Access_2{}^*$
$\quad \mid Expr_3\ Trigger_2\ Expr_1{}^* \mid Expr_3 \mid \epsilon;$

Since some designers may be uncomfortable with complex regular expressions, in Section 5.3 we describe our efforts to increase the usability of our scheme by developing a higher-level language. In this language, access policies can be expressed in terms of more abstract concepts such as isolation and controlled sharing.

Figure 5 shows a system-level view of the policy. From this policy, our policy compiler automatically generates a hardware description in Verilog of a reference monitor.

To further understand this security policy we will go through a simple example. The system starts out in $Access_0$, meaning neither processor can write to the AES core. Then, if $\mu Blaze_0$ needs to use the AES core, it first writes to cntrl_word$_1$, which triggers the reference monitor to transition to $Access_1$. Now that the reference monitor is in $Access_1$, the two processors can still access their peripherals and memory regions as they could in $Access_0$, except that $\mu Blaze_0$ can now access cntrl_word$_{AES}$ as well as AES1. This allows $\mu Blaze_0$ to place data into its portion of the shared AES-core memory and write the control words of the AES core, thus performing an encrypt/decrypt operation on the data. When the operation is done and $\mu Blaze_0$ has finished, it performs a write to cntrl_word$_2$, thus relinquishing control of the AES core and transferring the reference monitor back to $Access_0$. Similarly, $\mu Blaze_1$ can do the same to obtain use of the AES core. If one core tries to use or gain control of the AES core while it is being used by the other core, the reference monitor will simply deny access.
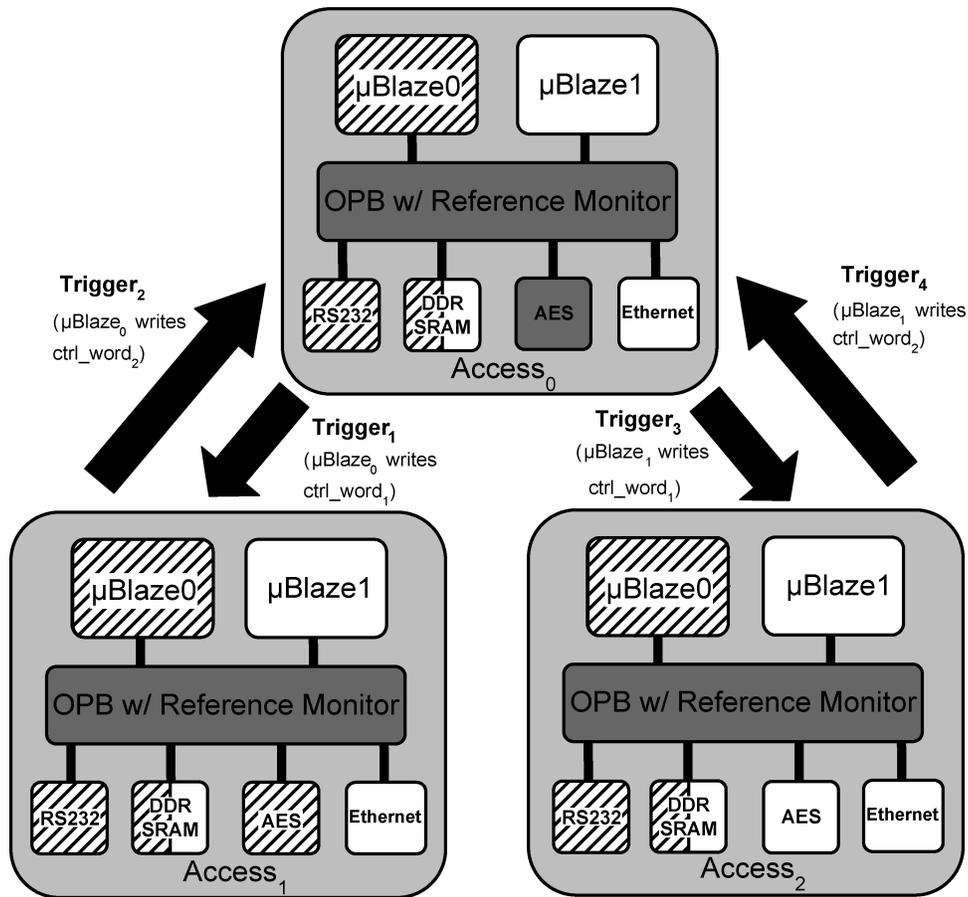
Fig. 5. This system-level diagram shows the three states of the reference monitor and what devices are in each isolation domain. The first domain is represented by the hatched pattern, and the second domain is represented by white background with no pattern. The SRAM is shared between the two and is therefore represented with half of each pattern.

Ensuring that the reference monitor cannot be bypassed is essential to the security of the system, since it regulates access to all the peripherals. The hardware must be verified to make sure that the reference monitor can in no way be tampered with or bypassed. Moats and drawbridges address this problem by allowing us to partition the system and then verify the connectivity of the various components in the system. For example, our tracing technique can detect an illegal connection between a core and memory that bypasses the reference monitor, an illegal connection between two cores, or an illegal connection that allows a core to snoop on the memory traffic of another core. In addition, the reference monitor itself can be isolated using a moat, which increases the reference monitor's resistance to tampering.

4.2.3 *Policy Compiler.* To understand how the access policy is converted to a reference monitor, we provide a condensed description of our policy compiler
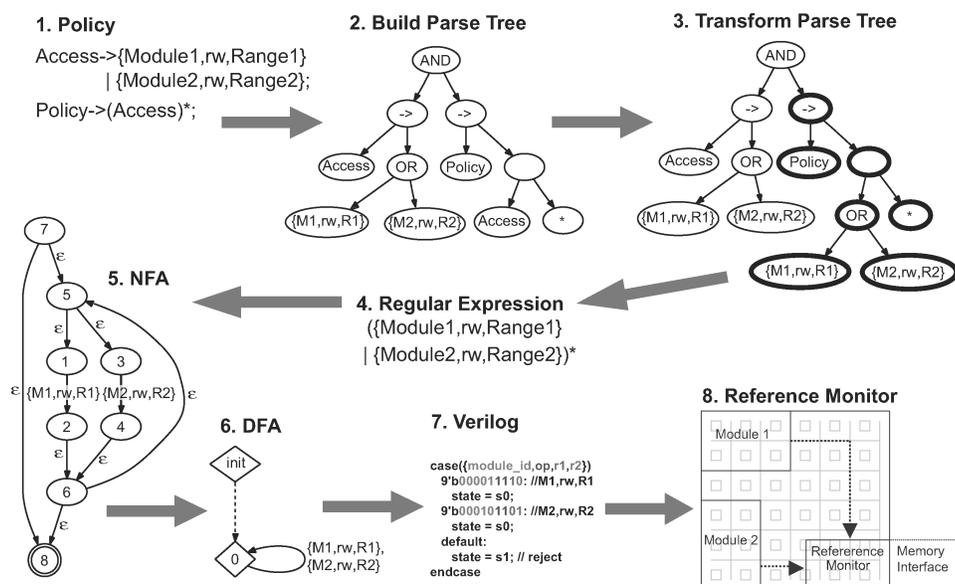
Fig. 6. Reference-monitor design flow for a toy policy. Our policy compiler first coverts the access policy to a regular expression, from which an NFA is contructed. Then, the NFA is converted to a minimized DFA, from which a hardware description of a reference monitor that enforces the policy is constructed.

here. Huffmire et al. [2006] provides a full description of our policy compiler. Figure 6 shows the reference-monitor design flow for a simple toy policy with one state. First, the access policy is converted to a regular expression by building and transforming a parse tree. Next, the regular expression is converted to a NFA using Thompson's algorithm. Then, the NFA is converted to a DFA using subset construction, and Hopcroft's minimization algorithm is used to produce a minimized DFA. The minimized DFA is then converted into a hardware description in Verilog HDL of a reference monitor that enforces the policy.

4.2.4 *Scalability.* In our design example, the system is protected by a single reference monitor. For larger, more complex systems, it may be necessary to have multiple reference monitors to ensure scalability. Reference monitors that enforce stateless policies (i.e., policies which have only one state) can simply be copied, since they can operate independently. However, for stateful policies, (i.e., those having more than one state) there is some communication overhead required so that all of the reference monitors share the same state. To reduce this overhead, system designers can make design decisions that minimize the amount of state that must be shared among all the reference monitors.

4.2.5 *Covert Channels.* There are several options for preventing a covert channel between the red and black domains. First, the AES core can be wiped between uses, and we describe a scrubbing technique in Huffmire et al. [2007] that exploits partial reconfiguration. To prevent a timing channel in which the red domain grabs and releases the AES core frequently (a behavior that can be

observed by the black domain), one way of limiting the bandwidth is to require that the AES core be used for a minimum amount of time. Another option is to use a statically scheduled sharing scheme in which each domain is allowed to use the AES core during a fixed interval. Another option is to introduce noise. Yet another option is to use counters to measure how many times the AES core is grabbed and released by the red domain, taking corrective action if this activity exceeds a predetermined threshold. We are also developing methods to prevent the internal state of the reference monitor from being used as a covert storage channel; these methods will include a policy checker that looks for cycles in the DFA that enforces the policy which indicates a possible covert channel, language features that prevent these cycles, dummy states that break up the cycles, only allowing a trusted module to change the state of the policy, and system-level techniques.

4.3 Moats and Drawbridges

Moats and drawbridges comprise two methods of providing spatial separation of cores on the chip. As previously discussed, spatial separation provides isolation, which in turn provides increased security and fault tolerance. Moats consist of a buffer zone of unused CLBs which are placed around cores. Their main purpose is to provide isolation and to enable the verification of this isolation. The size of the moat can be varied depending on the application, and is measured as the number of CLBs that are used as a buffer between cores. There is even the concept of a virtual moat (a moat of size 0), which occurs when the cores are placed right next to each other. Although they are touching and have no buffer zone around them, static analysis ensures that they are still isolated and placed in their own region. While this allows for lower area overhead, it requires greater verification effort.

Physically separating or partitioning the cores using moats and drawbridges provides increased security and fault tolerance, as discussed in Section 3. Physical separation is especially important if one or more of the cores was developed by a third-party designer (e.g., a COTS IP core). The third-party core may have a lower trust level than the other cores, resulting in a system with cores of varying trust levels. Physically separating the cores allows for isolation of the domains of trust. Communication with cores in a different domain of trust can go through a gatekeeper or reference monitor (as discussed before). This can all be verified using our verification technique. In addition to security, physical separation provides an additional layer of fault tolerance. If the cores are physically separated, it becomes more difficult for an invalid connection to be established between them. If the cores are intertwined and a bit is flipped by something such as an single-event upset (SEU) there is a chance that an invalid connection could be established between two cores; however, with the cores physically separated this chance is greatly reduced. In systems composed of cores of varying security levels, moats allow us to verify that only specified information flows are possible between cores. Moats prevent unintended information flows due to implementation errors, faulty design, or malicious intent.

Moats not only let us achieve physical separation, but also ease the process of verifying it. With moats and drawbridges, it is possible to analyze the information flow between cores and to ensure that the intended flow cannot be bypassed and is correctly implemented in hardware. The verification process would be very difficult if not impossible without them. Verification takes place at the bit-stream level. Since this is the last stage of design, there are no other design tools or steps that could introduce an error into the design. In a design without moats, the cores are intertwined, and trying to verify such a design at the bit-stream level is a hard problem because of the difficulty of determining where one core ends and another begins. Since modern FPGA devices have the capacity to hold designs with millions of gates, reverse engineering such a design is very complex. With the cores placed in moats, the task of verification becomes much simpler, and the physical isolation is stronger as well.

4.3.1 *Constructing Moats.*   The construction of moats is a fairly simple process. First, the design is partitioned into isolation domains. This step is highly design dependent. Once the design is partitioned, we can construct the moats using the Xilinx PlanAhead [Xilinx 2006] software. PlanAhead allows the designer to constrain cores to a certain area on the chip. The moats are constructed by placing the cores in certain regions on the chip. The remaining space not occupied by the cores effectively becomes the moat. The size of the moat changes based on the spacing between cores. PlanAhead then creates a user-constraints file which can be used to synthesize the design with the cores constrained to a certain area of the chip. Although the cores are constrained, the performance is not adversely affected. The tool simply confines the cores to a certain region, and the place-and-route tool can still choose an optimal layout within this region. One factor affecting performance is use of the drawbridges, which carry signals between cores. Since the cores are separated by a moat, a slightly longer delay may occur than if the cores were placed without a moat. However, this effect can be minimized if the drawbridge signals are properly buffered and the cores placed carefully.

Ensuring that a design is prepared to be partitioned using moats and drawbridges is very simple, and most designs should be ready with absolutely no modification. As long as the cores, or isolation domains, are separated into different design files (netlist or HDL) during the design phase, then the addition of moats using PlanAhead is trivial. We divided our test system into seven different cores: μBlaze$_0$, μBlaze$_1$, OPB with an integrated reference monitor, Ethernet, RS232, DDR SDRAM, and AES core. Since these were all separate cores added in XPS, the process of implementing the moats was as simple as selecting the core and then selecting a region for it on the chip with PlanAhead.

The separation of the design into seven different cores may seem unnecessary, since our design consists only of two isolation domains. However, since the cores all communicate through the OPB and since system security relies on the reference monitor, this is a necessary step. It allows us to verify that all cores go through the reference monitor and that there are no illegal connections between two cores. Doing this with only two isolation domains is not possible. It is also desirable to partition cores of different trust levels, since our design uses a mix

of third-party IP cores and custom-designed cores, resulting in different levels of trust. We can partition the third-party cores such as the Ethernet, RS232, and μBlaze processors away from our custom OPB and AES core, which have a higher level of trust. After one knows what cores to partition, it only remains to lay out the partitions on the chip.

The decision of where to place the cores and moats can involve some trial and error. We experimented with several different layouts before choosing the final one. Achieving a good layout is critical to the performance of the design. The key factors to achieving a good layout are placing the cores close to the I/O pins which they use, and placing cores which are connected close to each other. The moats were constructed for several different sizes so that the effect of moat size on performance could be observed. The size of the moat also affects the amount of required verification effort, the details of which are beyond the scope of this article.

5. DESIGN FLOW AND EVALUATION

The goal of this project was to analyze our secure design methods and determine the feasibility of implementing them in a real design. There are several main factors that determine the practicality of the methods. The two that we are concerned with are ease of design and the performance effect on design. Our techniques have been shown to be efficient and designer friendly.

5.1 Reference Monitor Implementation and Results

The actual implementation of the system was accomplished using Xilinx Platform Studio (XPS) software. The system was assembled using the graphical user interface in XPS; this entails separately loading the different components of our design into XPS, defining the interface between them, and specifying the device into which the design is to be loaded. The reference monitor was generated by the policy compiler described in Section 4.2.3. Integration of the reference monitor was accomplished by modifying the on-board peripheral bus (OPB) that came with the XPS software to create a custom OPB. Testing of the custom OPB as well as the other cores was performed through Modelsim simulations by specifying an array of inputs and verifying their respective outputs as correct. Once this was complete, the various components and the system's connectivity were synthesized to a hardware netlist and loaded into our FPGA.

The performance and area overheads of the design were analyzed with and without a reference monitor, and the results can be seen in Table I. The number of bus cycles was calculated by counting the number of cycles it took to perform 10,000 memory accesses to the DDR SRAM and then dividing by 10,000 to get the average cycles per access. The overhead due to our reference monitor was very small in terms of area and had little effect on performance.

The next step was the design of the software to run on the two μBlaze processors. The software was also developed and compiled using the XPS software. Testing and debugging of the software was done by downloading and running the software on the development board, using the Xilinx microprocessor

Table I. Area and Performance Effects of Reference Monitor
on System

Effects are shown on the synthesis of just the OPB and the
synthesis of the entire system. This table also shows the
average number of cycles per bus access, both with and
without the reference monitor.

Metric	W/O RM	With RM
OPB LUTs	158	208
System LUTs	9881	9997
OPB Max Clk (MHz)	300.86	300.86
System Max Clk (MHz)	73.52	65.10
Cycles/Bus Access	25.76	26.76

debugger (XMD). Software was also developed on the PC to allow send-ing/receiving of files to/from the board over RS-232 and Ethernet.

5.2 Moat Implementation and Results

The last stage in the design process is partitioning the design into moats. This is done by using the Xilinx PlanAhead software, which allows us to partition the chip into separate areas containing the cores, as shown in Figure 9. The moats are highlighted in Figure 9 as well, by the shaded areas surrounding each component. The design is then placed and routed using ten iterations in the multipass place-and-route for each different moat size and with no moats at all. Using multipass place-and-route allows us to find the best layout on the chip and to compare the tradeoffs of each chip layout generated.

Security is very important, but its cost must be managed; therefore, moats are only feasible if they do not have a significant impact on performance. The performance and area overheads of the system with various moat sizes were compared to the performance without moats. Figure 7 shows the performance effect of the moats, while Figure 8 shows the area overhead due to moats. For a moat size of 0, there was no effect on performance and none on area either, since there is no wasted moat area. A moat size of 6 would clearly consume more area since the moat occupies the unused CLBs. For this design, the extra overhead for the moat is over 1,000 CLBs, or 28% of the total chip. Performance overhead generally increases with moat size, but the impact is still very small, with a maximum decrease of less than 2%. Adding moats and a reference mon-itor to our system enhances the security with an almost negligible impact on performance and area. With a moat size of 0 there is no impact on the area, either.

5.3 Ease of Design

The cost of adding security is just as important as adding the security itself. No matter how many security advantages they provide, complex techniques will not be adopted unless they can easily be applied to a design. Although it cannot be quantified or tested, after evaluating our methods we believe that using moats, drawbridges, and reference monitors is effective and relatively simple.
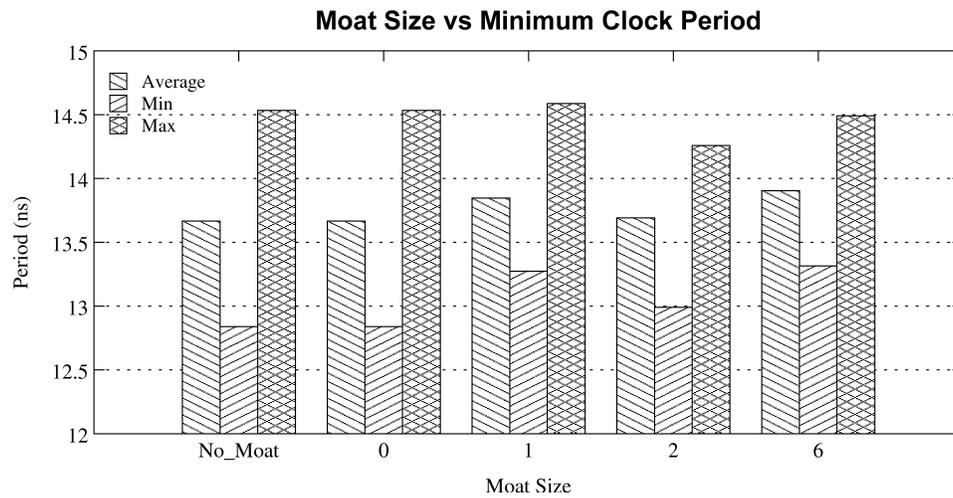
Moat Size vs Minimum Clock Period

Fig. 7.   Relationship between moat size and minimum clock period (performance) for the design. Performance is not greatly affected, with a maximum increase in clock speed of only 1.81% for a moat size of 6.

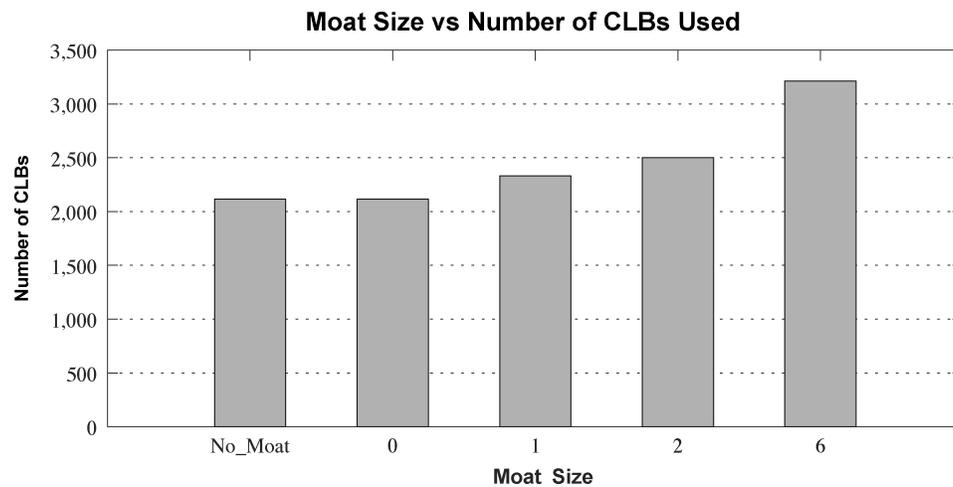**Moat Size vs Number of CLBs Used**

Fig. 8.   Relationship between number of CLBs used by the design and the moat size. Since no logic can be placed in the moat, the number of CLBs required increases with moat size. The area impact for larger moats can be quite significant.

Moats and drawbridges are very simple to add to a design because they are simply a form of floorplanning and can be implemented quickly and easily. While it may take a little work to get the right floorplan in order to achieve maximum performance, an experienced designer should have no trouble with this. The reference monitor is also very easy to add to a design. Since the reference monitor was integrated into the OPB, it is trivial to add it to any design using an on-chip bus. Futhermore, the designer does not have to worry about the low-level details of the reference monitor. The designer specifies the access
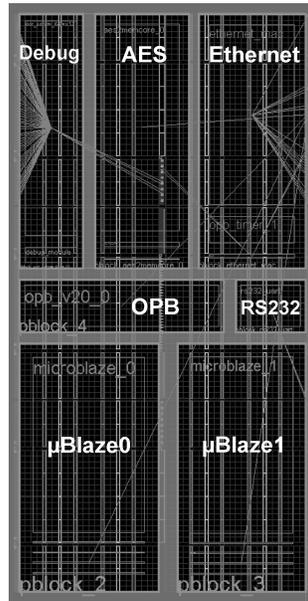
Fig. 9.   Floorplan view of our design in PlanAhead. The shaded areas between the cores are the moats.

policy in our laguage, and our policy compiler automatically generates the neccessary Verilog files for the reference monitor. We are developing a higher-level language for expressing access policies so that the designer does not have to be an expert with regular expressions. For example, this higher-level language allows designers to express access policies in terms of abstract concepts such as isolation and controlled sharing. We are also developing a compiler to translate the policy from this higher-level language.

6. CONCLUSIONS AND FUTURE WORK

Addressing the problem of security on reconfigurable-hardware designs is very important because reconfigurable devices are used in a wide variety of critical applications. We have built an embedded system for the purpose of evaluating security primitives for reconfigurable hardware. We have developed a stateful security policy that divides the resources in the system into two isolation domains. A reference monitor enforces the isolation of these domains, but also permits controlled sharing of the encryption core. A spatial isolation technique, using regions called moats, further isolates the domains, and a static analysis technique, using interconnections called drawbridges, facilitates the controlled interaction of isolated components. Together, moats and drawbridges comprise a separation technique that also helps to ensure that the reference monitor is tamperproof and cannot be bypassed. Our results show that these security primitives do not significantly impact the performance or area of the system.

We see many possibilities for future work. The DMA (direct memory access) controller introduces a new security challenge because of its ability to

independently copy blocks of memory. The development of a secure DMA controller with an integrated reference monitor requires understanding the tradeoffs between security and performance. In addition, memory-access policies may need to be constructed differently for systems that use a DMA controller. For example, the request to the DMA could include the requesting module's ID.

We leave to future work the problem of denial of service because the primary focus of this article is data protection. Although there is no overhead in denying a request, a subverted core could launch a denial-of-service attack against the system by repeatedly making an illegal request.

The state of computer security is grim, as increased spending on security has not resulted in fewer attacks. Embedded devices are vulnerable because few embedded designers even bother to think about security, and many people incorrectly assume that embedded systems are secure. A holistic approach to system security is needed, and new security technologies must move from the lab into widespread use by an industry which is often reluctant to embrace them. Fortunately, the reprogrammable nature of FPGAs allows security primitives to be immediately incorporated into designs. In order to be adopted by embedded designers, who are typically not security experts, security primitives need to be usable and understandable to those outside the security discipline. They must also be easy to use and have little performance impact. The primitives implemented in this article have been shown to have very low performance and area overheads, and would be rather easy to integrate into a design.

ACKNOWLEDGMENTS

REFERENCES

BONDHUGULA, U., DEVULAPALLI, A., FERNANDO, J., WYCKOFF, P., AND SADAYAPPAN, P. 2006. Parallel FPGA-based all-pairs shortest-paths in a directed graph. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

BOSSUET, L., GOGNIAT, G., AND BURLESON, W. 2004. Dynamically configurable security for SRAM FPGA bitstreams. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, NM.

CHIEN, A. AND BYUN, J. 1999. Safe and protected execution for the Morph/AMRM reconfigurable processor. In *7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA.

GOGNIAT, G., WOLF, T., AND BURLESON, W. 2006. Reconfigurable security support for embedded systems. In *Proceedings of the 39th Hawaii International Conference on System Sciences*.

HADZIC, I., UDANI, S., AND SMITH, J. 1999. FPGA viruses. In *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications (FPL)*, Glasgow, UK.

HARPER, S. AND ATHANAS, P. 2004. A security policy based upon hardware encryption. In *Proceedings of the 37th Hawaii International Conference on System Sciences*.

HARPER, S., FONG, R., AND ATHANAS, P. 2003. A versatile framework for FPGA field updates: An application of partial self-reconfiguration. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping*.

HUFFMIRE, T., BROTHERTON, B., WANG, G., SHERWOOD, T., KASTNER, R., LEVIN, T., NGUYEN, T., AND IRVINE, C. 2007. Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA.

HUFFMIRE, T., PRASAD, S., SHERWOOD, T., AND KASTNER, R. 2006. Policy-Driven memory protection for reconfigurable systems. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Hamburg, Germany.

HUTCHINGS, B., FRANKLIN, R., AND CARVER, D. 2002. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

IRVINE, C., LEVIN, T., NGUYEN, T., AND DINOLT, G. 2004. The trusted computing exemplar project. In *Proceedings of the 5th IEEE Systems, Man and Cybernetics Information Assurance Workshop*, West Point, NY. 109–115.

KEAN, T. 2001. Secure configuration of field programmable gate arrays. In *Proceedings of the 11th International Conference on Field Programmable Logic and Applications (FPL)*, Belfast, UK.

KEAN, T. 2002. Cryptographic rights management of FPGA intellectual property cores. In *10th ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey, CA.

KEMMERER, R. 1983. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Trans. Comput. Syst.*

KOCHER, P., JAFFE, J., AND JUN, B. 1999. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference*.

LACH, J., MANGIONE-SMITH, W., AND POTKONJAK, M. 1999a. FPGA fingerprinting techniques for protecting intellectual property. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, San Diego, CA.

LACH, J., MANGIONE-SMITH, W., AND POTKONJAK, M. 1999b. Robust FPGA intellectual property protection through multiple small watermarks. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation (DAC)*, New Orleans, LA.

LEE, R. B., KWAN, P. C. S., MCGREGOR, J. P., DWOSKIN, J., AND WANG, Z. 2005. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*. 2–13.

LEVIN, T. E., IRVINE, C. E., AND NGUYEN, T. D. 2004. A least privilege model for static separation kernels. Tech. Rep. NPS-CS-05-003, Naval Postgraduate School.

LIE, D., THEKKATH, C., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J., AND HOROWITZ, M. 2000. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA.

MCGRATH, D. 2005. Gartner dataquest analyst gives ASIC, FPGA markets clean bill of health. *EE Times*.

MCLEAN, M. AND MOORE, J. 2007. Securing FPGAS for red/black systems, FPGA-based single chip cryptographic solution. In *Military Embedded Systems*.

MILANOWSKI, R. AND MAURER, M. 2006. Outsourcing poses unique challenges for the U.S. military-electronics community. *Chip Des. Mag.*

MILLEN, J. 1987. Covert channel capacity. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA.

NATIONAL SECURITY TELECOMMUNICATIONS AND INFORMATION SYSTEMS SECURITY COMMITTEE. 1995. NSTISSAM Tempest/2-95 red/black installation guidance.

NAVARRO, J., IYER, S., DRUSCHEL, P., AND COX, A. 2002. Practical, transparent operating system support for superpages. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA.

NGO, H., GOTTUMUKKAL, R., AND ASARI, V. 2005. A flexible and efficient hardware architecture for real-time face recognition based on Eigenface. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*.

PERCIVAL, C. 2005. Cache missing for fun and profit. In *BSDCan*, Ottowa, Ontario, Canada.

RUSHBY, J. 1984. A trusted computing base for embedded systems. In *Proceedings of the 7th DoD/NBS Computer Security Conference*, 294–311.

RUSHBY, J. 2000. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. In *DOT/FAA/AR-99/58*.

SALEFSKI, B. AND CAGLAR, L. 2001. Reconfigurable computing in wireless. In *Proceedings of the Design Automation Conference (DAC)*.

SALTZER, J. 1974. Protection and the control of information sharing in Multics. *Commun. ACM 17,* 7 (Jul.), 388–402.

SALTZER, J. AND SCHROEDER, M. 1974. The protection on information in computer systems. *Commun. ACM 17,* 7 (Jul.).

SAPUTRA, H., VIJAYKRISHNAN, N., KANDEMIR, M., IRWIN, M., BROOKS, R., KIM, S., AND ZHANG, W. 2003. Masking the energy behavior of DES encryption. In *IEEE Design Automation and Test in Europe (DATE)*.

STANDAERT, F., OLDENZEEL, L., SAMYDE, D., AND QUISQUATER, J. 2003. Power analysis of FPGAs: How practical is the attack? *Field-Program. Logic Appl. 2778,* 2003 (Sept.), 701–711.

THOMPSON, K. 1984. Reflections on trusting trust. *Commun. ACM 27,* 8.

WEISSMAN, C. 2003. MLS-PCA: A high assurance security architecture for future avionics. In *Proceedings of the Annual Computer Security Applications Conference*, Los Alamitos, CA. 2–12.

WITCHEL, E., CATES, J., AND ASANOVIC, K. 2002. Mondrian memory protection. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA.

XILINX INC. 2006. Planahead methodology guide. Xilinx, San Jose, CA.