Implementing Memory Protection Primitives on Reconfigurable Hardware

Brett Brotherton and Nick Callegari
University of California, Santa Barbara
Department of Electrical and Computer Engineering
Santa Barbara, CA 93106
{bbrother,nick_callegari}@ece.ucsb.edu

Ted Huffmire
University of California, Santa Barbara
Department of Computer Science
Santa Barbara, CA 93106
huffmire@cs.ucsb.edu

Abstract

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 protection primitives for reconfigurable hardware by building a real embedded system with several cores on a single FPGA. 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.

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 non-recurring 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 for free. In addition, the performance gap between FPGAs and ASICs is narrowing. Although FPGAs are slower than ASICs, FPGAs can be fabricated using the latest deep sub-micron process technology, while many ASIC companies are forced to use more affordable technology that is several generations behind.

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 [24], to wireless networks [27], to intrusion detection systems [10], to supercomputers [1]. In fact it is estimated that in 2005 alone there were over 80,000 different commercial FPGA designs projects started. [21]

Since major IC manufacturers outsource most of their operations to a variety of countries [22], 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 typically cobbled together from a collection of existing modules (called cores) in order to save both time and money. Cost pressures often force designers to obtain cores from third parties, resulting in mixed trust designs.

The goal of this paper is to evaluate recently proposed security primitives for reconfigurable hardware [9] [8] by building an embedded system consisting of multiple cores on a single FPGA. Our testing platform will help us to better understand how these security primitives interact with cores on a real system. Overcoming the problems of integrating several cores together with reconfigurable protection primitives will provide an opportunity to develop more realistic security policy specifications that drive the enforcement mechanisms on embedded systems.

In addition to designing a security policy for the system, we will build a programmatic interface for user applications to communicate with the device. Since security primitives
on embedded devices need to be efficient, we also wish to ensure that the combined system achieves efficient memory system performance. To this end we quantitatively evaluate the impact of our methods on area and cycle time.

2 Reconfigurable Protection

This section motivates the problem of protection on reconfigurable systems. Figure 1 shows several different strategies for providing protection for embedded systems. Cost pressures force embedded designers to integrate multiple cores onto a single device, and these cores essentially run “naked” on the device. Since cores are often provided by third parties or created with third party tools, they may operate at different trust levels. Consider a system with two CPU cores and a shared crypto core. We must prevent a subverted core or a subverted design tool from allowing secret data such as keys from being leaked off-chip via a direct connection to the I/O pins.

Ideally, every application runs on its own dedicated device, but this is clearly inefficient from a cost perspective. In contrast to strictly physical protection, separation kernels [26] [11] [19] use software virtualization to prevent applications from interfering with each other, but they come with the overhead of software and can only run on general-purpose processors. Huffmire et al. proposed a third approach, reconfigurable protection [9], that uses a reconfigurable reference monitor that enforces 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.

Huffmire et al. proposed another protection primitive, moats and drawbridges [8], that exploits 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 use static analysis of the bit-stream to ensure that only specified connections between cores can be established. Drawbridges can also be used to ensure that the reference monitor cannot be bypassed and is always invoked. We have also developed a TDMA bus architecture to prevent snooping on the memory bus traffic.

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 both on-chip BRAM and off-chip DRAM. For example, a MicroBlaze CPU and an AES encryption core can share a block of BRAM. The CPU encrypts plaintext by copying the plaintext 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 state that gives the CPU exclusive access to the shared buffer and another state that gives the AES core exclusive access to the buffer. The transitions between these two states occur when the cores signal to each other via the control words. In Section 6, we extend this idea to construct a policy with three states for a system with two CPU cores, a shared AES core, and shared external memory.

In the above scenario involving BRAM, the cores are directly connected. Communication with external DRAM, on the other hand, requires utilizing a shared bus. A 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, a TDMA arbiter sits between the modules and the bus. Figure 2 shows two alternative memory protection architectures. The reference monitor can be placed between the bus and the DRAM. Alternatively, 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.

When a module violates the policy, the system denies the request. In this paper, we do not terminate a core that offends the policy since that would make it difficult to design a policy for sharing the AES core. There will be occasions when the AES core in in use by one of the processors, and the request by the other processor to use the AES core will be denied in this case.

Figure 3 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 comprises three inputs: the module ID, the op {read, write, etc.}, and the address. The output is a single bit: 1 for grant and 0 for deny. First, the hardware converts the memory access address to a bit vector. To do this, it checks all the ranges in parallel and sets the bit corresponding to the range ID that contains the in-
put 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.

3 Related Work

3.1 IP Theft

Most of the work relating to FPGA security targets the problem of preventing the theft of intellectual property and securely uploading bitstreams 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 [2] [12] [13], fingerprinting [16], and watermarking [17]. However, establishing a root of trust on a fielded device is challenging because it requires a decryption key to be incorporated into the finished product. Some FPGAs can be remotely updated in the field, and industry has devised secure hardware update channels that use authentication mechanisms to prevent a subverted bitstream from being uploaded [7] [6]. 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 [5].

3.2 Reconfigurable Protection

Besides the work of Huffmire et al. [9] [8], there appears to be little other work on the specifics of managing FPGA resources in a secure manner. Chien and Byun have perhaps the closest work, where they addressed the safety and protection concerns of enhancing a CMOS processor with reconfigurable logic [3]. Their design achieves process isolation by providing a reconfigurable virtual machine to each process, and their architecture uses hardwired Translation Lookaside 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 [20] or virtual secure co-processing [18].

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 [4]. 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 re-
Since the placement of the reference monitor affects memory bus performance, this paper explores various design alternatives. In the figure on the left, a memory access must pass through the reference monitor (RM) before going to memory. In the figure on the right, the reference monitor (RM) snoops on the bus, and a buffer (B) stores the data until the access is approved. An arbiter prevents the bus from being accessed by more than one module at a time.

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

Although moats provide physical isolation of cores, it is possible that cores could still communicate via a covert channel. Exploitation of a covert channel results in the unintended flow of information between cores. Covert channels work via an internal shared resource, such as power consumption, processor activity, disk usage, error conditions, or temperature of the device. Classical covert channel analysis involves the articulation of all shared resources on chip, identifying the share points, determining if the shared resource is exploitable, determining the bandwidth of the covert channel, and determining whether remedial action can be taken. 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) or closing the channel. Unfortunately, an adversary can extract a signal from the noise, given sufficient resources.

Of course the techniques we are evaluating in this paper [9] [8] are primarily about restricting the opportunity for direct channels and trap doors [30]. The memory protection scheme proposed in [9] is an example of that. Without any
memory protection, a core can leak secret data by writing the data directly to memory. Another example of a direct channel is a tap that connects two cores. An unintentional tap is a direct channel that can be established through luck. For example, the place-and-route tool’s optimization strategy may interleave the wires of two cores.

4 System Architecture

The system we designed is a multicore system on chip which can be seen in Figure 4. The design consists of seven different “cores”: We have \( \mu \text{Blaze}_0, \mu \text{Blaze}_1 \), the OPB along with its arbiter and the reference monitor, the AES core, the DDR SDRAM, the RS-232 interface, and the Ethernet interface. These components share resources and interact with one another. A reference monitor/arbiter sits between the MicroBlaze processors and the on-chip peripheral bus (OPB). Shared external memory (DRAM), the AES Core, the RS-232 interface, and the Ethernet interface also are connected to the bus. In addition to integrating the Ethernet core into the system, we have designed software to communicate over TCP with the processor. Our system has the ability to send data together with the desired operation to the device and receive either encrypted or decrypted data depending on the request. We have also modified the serial code to work with the new file format. Our system can receive and process files over both serial and Ethernet connections. In addition, we have configured our two-processor system to run applications simultaneously.

Through the reference monitor the system is divided into two systems which are isolated yet share resources. The first system consists of \( \mu \text{Blaze}_0 \), the DDR SDRAM, and the RS-232 device. The second system consists of \( \mu \text{Blaze}_1 \), the DDR SDRAM, and the Ethernet device. Everything is interconnected with the OPB (Onboard Peripheral Bus), which is the glue for the systems, and both systems make use of the AES core.

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 sensitivity level than the RS-232 interface, which is a local connection. We want to prevent the mixing of data of different sensitivity levels. First we assign a processor to each communication interface. The processor is responsible for implementing the driver and handling the I/O and data processing for each communication interface. This is a little better, but it is still not secure since there is nothing preventing one processor from accessing both I/O devices or from accessing the other processor’s data in the shared memory. 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 the I/O device which it is intended to use. Access to the AES core is arbitrated by having multiple states in our memory access policy, which we explain in Section 6.

In order to integrate the reference monitor with the OPB, we first integrated the reference monitor with the OPB block ram controller and ensured that it functioned correctly with low latency and overhead. Then, we tackled the more difficult problem of integrating the reference monitor with the OPB. Our system can regulate access to any of the slave processors on the bus, and our modifications only add one cycle to the latency.

5 Communication Interface

One of the design goals of our system is to improve on the technique of using HyperTerminal to connect to the AES core via the serial connection. We have incorporate a programmatic interface to allow a user application to send data and keys to the device and receive the result programmatically. This enables applications to provide a user interface that allows the user to select a data file and a key file and to specify where to save the file containing the result.

Our user interface is implemented in C++. The user can specify the desired operation, the input file, and the key file. To achieve modularized functionality, we implemented the serial socket coding to allow the user to connect to the Xilinx board. Functions are enabled to listen to the board and output the encrypted/decrypted data to a text file. Our interface can handle multiple forms of I/O (both serial and Ethernet). Each type of I/O is assigned to a specific processor core. Our system allows the user to specify which connection method to use when communicating with the device.

6 Security Policy

Our reference monitor will enforce a stateful memory access policy. There are three states: one state for the case in which Processor\(_1\) (or Module\(_1\)) has access to the AES Core, one state for the case where Processor\(_2\) (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 controllers such that Processor\textsubscript{1} and RS-232 data is in a separate isolation domain as Processor\textsubscript{2} and Ethernet data. 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:

\begin{itemize}
\item[Range\textsubscript{0}] \rightarrow \{0x41400000,0x4140ffff\}; (Debug)
\item[Range\textsubscript{1}] \rightarrow \{0x28000000,0x28000777\}; (AES1)
\item[Range\textsubscript{2}] \rightarrow \{0x28000800,0x28000fff\}; (AES2)
\item[Range\textsubscript{3}] \rightarrow \{0x24000000,0x24777777\}; (DRAM1)
\item[Range\textsubscript{4}] \rightarrow \{0x24800000,0x24fffff\}; (DRAM2)
\item[Range\textsubscript{5}] \rightarrow \{0x40600000,0x40600fff\}; (RS-232)
\end{itemize}

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

\begin{itemize}
\item[Access\textsubscript{0}] \rightarrow \{Module\textsubscript{1},rw,Ranges\textsubscript{5}\}
\item[Access\textsubscript{1}] \rightarrow Access\textsubscript{0}
\end{itemize}

\begin{itemize}
\item\text{Ranges}\textsubscript{5} \rightarrow \{0x40c00000,0x40c0ffff\}; (Ethernet)
\item\text{Ranges}\textsubscript{7} \rightarrow \{0x28000004,0x28000007\}; (Ctrl\_Word\textsubscript{1})
\item\text{Ranges}\textsubscript{8} \rightarrow \{0x28000008,0x2800000f\}; (Ctrl\_Word\textsubscript{2})
\item\text{Ranges}\textsubscript{9} \rightarrow \{0x28000000,0x28000003\}; (Ctrl\_Word\_AES)
\end{itemize}
Table 1. This table shows the area and performance effects of the reference monitor on the system. Effects are shown on synthesis of just the OPB and synthesis of the entire system. Also shows the average number of cycles per bus access with and without the reference monitor.

<table>
<thead>
<tr>
<th>Metric</th>
<th>W/O RM</th>
<th>With RM</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPB LUTs</td>
<td>158</td>
<td>208</td>
</tr>
<tr>
<td>System LUTs</td>
<td>9881</td>
<td>9997</td>
</tr>
<tr>
<td>OPB Max Clk(MHz)</td>
<td>300.86</td>
<td>300.86</td>
</tr>
<tr>
<td>System Max Clk(MHz)</td>
<td>73.52</td>
<td>65.10</td>
</tr>
<tr>
<td>Cycles/Bus Access</td>
<td>25.76</td>
<td>26.76</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\text{Trigger}_1 & \rightarrow \{ \text{Module}_1, \text{w}, \text{Range}_1 \}; \\
\text{Trigger}_2 & \rightarrow \{ \text{Module}_1, \text{w}, \text{Range}_8 \}; \\
\text{Trigger}_3 & \rightarrow \{ \text{Module}_2, \text{w}, \text{Range}_7 \}; \\
\text{Trigger}_4 & \rightarrow \{ \text{Module}_2, \text{w}, \text{Range}_8 \}; \\
\end{align*}
\]

The final part of the policy uses regular expressions to specify the structure of the policy’s state machine:

\[
\begin{align*}
\text{Expr}_1 & \rightarrow \text{Access}_0 | \text{Trigger}_3 \text{ Access}_2* \text{ Trigger}_4; \\
\text{Expr}_2 & \rightarrow \text{Access}_1 | \text{Trigger}_2 \text{ Expr}_1* \text{ Trigger}_1; \\
\text{Expr}_3 & \rightarrow \text{Expr}_1* \text{ Trigger}_1 \text{ Expr}_2*; \\
\text{Policy} & \rightarrow \text{Expr}_1* | \text{Expr}_1* \text{ Trigger}_3 \text{ Access}_2* \\
& | \text{Expr}_3 \text{ Trigger}_2 \text{ Expr}_1* \text{ Trigger}_3 \text{ Access}_2* \\
& | \text{Expr}_3 \text{ Trigger}_2 \text{ Expr}_1* | \text{Expr}_3 | \epsilon;
\end{align*}
\]

Figure 5 shows the Deterministic Finite Automaton (DFA) that enforces this policy. From this policy, we automatically generate a hardware description in Verilog of a reference monitor.

7 Results

7.1 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. The reference monitor was generated by the policy compiler, but it had to be slightly tweaked by hand to work with our system. A trivial change to the policy compiler is all that is needed to address this issue. Integration of the reference monitor was accomplished by modifying the On-board Peripheral Bus (OPB) that came with the XPS software. After this was complete, the system was synthesized to hardware. The performance and area overhead of the design was analyzed with and without a reference monitor, and the results can be seen in Table 1. The overhead due to our reference monitor was very small in terms of area and actually increased our clock speed.

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 Debugger (XMD).

7.2 Moat Implementation and Results

The last stage in the design process was partitioning the design into moats. This is done by using the Xilinx Plan Ahead software, which allows us to partition the chip into separate areas containing the cores as shown in Figure 6. Moats are a method of physical isolation to help improve the security of the design.

Security is very important but its cost must be managed, so moats are only feasible if they do not have a significant impact on performance. The performance and area overhead of the system with various moat sizes was compared to the performance without moats. Table 2 shows the results of this comparison. For a moat size of 0 and 6, the performance effect was almost nothing, and the area effect was also negligible. However, this does not take into account the area required for the moat. A moat size of six would clearly consume more area.

8 Conclusions and Future Work

Addressing the problem of security on reconfigurable hardware is very important because of their use in a wide variety of critical applications. We have built an embedded system for the purpose of evaluating security primitives for reconfigurable hardware. Our system consists of two CPU
cores, an encryption core, and interfaces for connecting the system to a local connection or network. We have developed a stateful security policy that divides the resources in the system into two isolation domains. A reference monitor enforces this policy which specifies the legal sharing of the encryption core as well as the isolation domains. Our results show that these security primitives do not significantly impact the performance 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 requires understanding 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 application of formal methods to verify each stage of the reference monitor design flow. We also leave to future work the problem of denial-of-service because the primary focus of this paper is data protection. Although there is no overhead of 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 industry, which is often reluctant to embrace them. 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.

References


Figure 6. Layout of the design in PlanAhead after it was partitioned into seven cores with moats.