Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems

Ted Huffmire⋆, Brett Brotherton†, Gang Wang†, Timothy Sherwood⋆,
Ryan Kastner†, Timothy Levin‡, Thuy Nguyen‡, and Cynthia Irvine‡

⋆ University of California, Santa Barbara
Department of Computer Science
Santa Barbara, CA 93106
{huffmire,sherwood}@cs.ucsb.edu

† University of California, Santa Barbara
Department of Electrical and Computer Engineering
Santa Barbara, CA 93106
{bbrother,wanggang,kastner}@ece.ucsb.edu

‡ Naval Postgraduate School
Department of Computer Science
Monterey, CA 93943
{levin,tdnguyen,irvine}@nps.edu

Abstract

Blurring the line between software and hardware, reconfigurable devices strike a balance between the raw high speed of custom silicon and the post-fabrication flexibility of general-purpose processors. While this flexibility is a boon for embedded system developers, who can now rapidly prototype and deploy solutions with performance approaching custom designs, this results in a system development methodology where functionality is stitched together from a variety of "soft IP cores," often provided by multiple vendors with different levels of trust. Unlike traditional software where resources are managed by an operating system, soft IP cores necessarily have very fine grain control over the underlying hardware. To address this problem, the embedded systems community requires novel security primitives which address the realities of modern reconfigurable hardware. We propose an isolation primitive, moats and drawbridges, that are built around four design properties: logical isolation, interconnect traceability, secure reconfigurable broadcast, and configuration scrubbing. Each of these is a fundamental operation with easily understood formal properties, yet maps cleanly and efficiently to a wide variety of reconfigurable devices. We carefully quantify the required overheads on real FPGAs and demonstrate the utility of our methods by applying them to the practical problem of memory protection.

1 Introduction

Reconfigurable hardware, such as a Field Programmable Gate Array (FPGA), provides a programmable substrate onto which descriptions of circuits can be loaded and executed at very high speeds. 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 satellites to the Mars Rover. Their circuit-level flexibility allows system functionality to be updated arbitrarily and remotely. Real-time and military projects, such as the Joint Strike Fighter, make frequent use of FPGAs because they provide both high-performance and well-defined timing behavior, but they do not require the costly fabrication of custom chips.

FPGA technology is now *the* leading design driver for almost every single foundry[1] meaning that they enjoy the benefits of production on a massive scale (reduced cost, better yield, difficult to tamper with), yet developers are free to deploy their own custom circuit designs by configuring the device in the appropriate ways. This has significantly lowered the primary impediment to hardware development, cost, and as such we are now seeing an explosion of reconfigurable hardware based designs in everything from face

[1] A foundry is a wafer production and processing plant available on a contract basis to companies that do not have wafer fab capability of their own

recognition systems [39], to wireless networks [42], to intrusion detection systems [20], to supercomputers [5]. In fact it is estimated that in 2005 alone there were over 80,000 different commercial FPGA designs projects started. [36] Unfortunately, while the economics of the semiconductor industry has helped to drive the widespread adoption of reconfigurable devices in a variety of critical systems, it is not yet clear that such devices, and the design flows used to configure them, are actually trustworthy.

Reconfigurable systems are typically cobbled together from a collection of exiting modules (called cores) in order to save both time and money. Although ideally each of these cores would be formally specified, tested, and verified by a highly trusted party, in reality, such a development model cannot hope to keep up with the exponential increases in circuit area and performance made possible by Moore's Law. Unlike uni-processor software development, where the programming model remains fixed as transistor densities increase, FPGA developers must explicitly take advantage of denser devices through changes in their design. Given that embedded design is driven in large part by the demand for new features and the desire to exploit technological scaling trends, there is a constant pressure to mix everything on a single chip: from the most critical functionality to the latest fad. Each of these cores runs "naked" on the reconfigurable device (i.e., without the benefit of an operating system or other intermediate layer), and it is possible that this mixing of trust levels could be silently exploited by an adversary with access to any point in the design flow (including design tools or implemented cores). In an unrestricted design flow, even answering the question of "are these two cores capable of communication" is computationally difficult to answer.

Consider a more concrete example, a system with two soft-processor cores and an AES encryption engine sharing a single FPGA. Each of these three cores requires access to off-chip memory to store and retrieve data. How can we ensure that the encryption key for one of the processors cannot be obtained by the other processor by either reading the key from external memory or directly from the encryption core itself? There is no virtual memory on these systems, and after being run through an optimizing CAD tool the resulting circuit is a single entangled mess of gates and wires. To prevent the key from being read directly from the encryption core itself, we must find some way to isolate the encryption engine from the other cores at the gate level. To protect the key in external memory, we need to implement a memory protection module, we need to ensure that each and every memory access goes through this monitor, and we need to ensure that all cores are communicating only through their specified interfaces. To ensure these properties hold at even the lowest levels of implementation (after all the design tools have finished their transformations), we argue that slight modifications in the design methods and tools can enable the rapid static verification of finished FPGA bitstreams[2]. The techniques presented in this paper are steps towards a cohesive reconfigurable system design methodology that explicitly supports cores with varying levels of trust and criticality – all sharing a single physical device.

Specifically, we present the idea of Moats and Drawbridges, a statically verifiable method to provide isolation and physical interface compliance for multiple cores on a single reconfigurable chip. The key idea of the Moat is to provide logical and physical isolation by separating cores into different areas of the chip with "dead" channels between them that can be easily verified. Note that this does not require a specialized physical device; rather, this work only assumes the use of commercially available commodity parts. Given that we need to interconnect our cores at the proper interfaces (Drawbridges), we introduce interconnect tracing as a method for verifying that interfaces carrying sensitive data have not been tapped or routed improperly to other cores or I/O pads. Furthermore, we present a technique, configuration scrubbing, for ensuring that remnants of a prior core do not linger following a partial reconfiguration of the system to enable object reuse. Once we have a set of drawbridges, we need to enable legal inter-core communication. We describe two secure reconfigurable communication architectures that can be easily mapped into the unused moat areas (and statically checked for isolation), and we quantify the implementation trade-offs between them in terms of complexity of analysis and performance. Finally, to demonstrate the efficacy of our techniques, we apply them to a memory protection scheme that enforces the legal sharing of off-chip memory between multiple cores.

2 Reconfigurable Systems

As mentioned in Section 1, a reconfigurable system is typically constructed piecemeal from a set of existing modules (called cores) in order to save both time and money; rarely does one design a full system from scratch. One prime example of a module that is used in a variety of contexts is a soft-processor. A soft-processor is simply a configuration of logical gates that implements the functionality of a processor using the reconfigurable logic of an FPGA. A soft-processor, and other intellectual property (IP) cores[3] such as AES implementations and Ethernet controllers, can

[2]bitstreams are the term for the detailed configuration files that encode the exact implementation of a circuit on reconfigurable hardware – in many ways they are analogous to a statically linked executable on a traditional microprocessor

[3]Since designing reconfigurable modules is costly, companies have developed several schemes to protect this valuable intellectual property, which we discuss in Section 6.

be assembled together to implement the desired functionality. Cores may come from design reuse, but more often than not they are purchased from third party vendors, generated automatically as the output of some design tool, or even gathered from open source repositories. While individual cores such as encryption engines may be formally verified [30], a malicious piece of logic or compromised design tool may be able to exploit low level implementation details to quietly eavesdrop on, or interfere with, trusted logic. As a modern design may implement millions of logical gates with tens of millions of interconnections, the goal of this paper is to explore design techniques that will allow the inclusion of both trusted and untrusted cores on a single chip, without the requirement that expensive static verification be employed over the entire finished design. Such verification of a large and complex design requires reverse engineering, which is highly impractical because many companies keep details about their bit-streams proprietary.

Increasingly we are seeing reconfigurable devices emerge as the flexible and high-performance workhorses inside a variety of high performance embedded computing systems [4, 9, 11, 22, 35, 45], but to understand the potential security issues, we need to build on an understanding of at least a simplified modern FPGA design flow. In this section we describe a modern device, a typical design flow, and the potential threats that our techniques are expected to handle.

2.1 Reconfigurable Hardware

FPGAs lie along a continuum between general-purpose processors and application-specific integrated circuits (ASICs). While general purpose processors can execute any program, this generality comes at the cost of serialized execution. On the other hand, ASICs can achieve impressive parallelism, but their function is literally hard wired into the device. The power of reconfigurable systems lies in their ability to flexibly customize an implementation down at the level of individual bits and logic gates without requiring a custom piece of silicon. This can often result in performance improvements on the order of 100x as compared to, per unit silicon, a similar microprocessor [7, 10, 50].

The growing popularity of reconfigurable logic has forced practitioners to begin to consider security implications, but as of yet there is no set of best design practices to guide their efforts. Furthermore, the resource constrained nature of embedded systems is perceived to be a challenge to providing a high level of security [26]. In this paper we describe a set of low level methods that a) allow effective reasoning about high level system properties, b) can be supported with minimal changes to existing tool flows, c) can be statically verified with little effort, d) incur relatively small area and performance overheads, and e) can be used with commercial off-the-shelf parts. The advantage of developing security primitives for FPGAs is that we can immediately incorporate our primitives into the reconfigurable design flow today, and we are not dependent on the often reluctant industry to modify the design of their silicon.

2.2 Mixed-Trust Design Flows

Figure 1 shows a few of the many different design flows used to compose a single modern embedded system. The reconfigurable implementation relies on a large number of sophisticated software tools that have been created by many different people and organizations. Soft IP cores, such as an AES core, can be distributed in the form of Hardware Description Language (HDL), netlists[4] or a bitstream. These cores can be designed by hand, or they can be automatically generated by computer programs. For example, the Xilinx Embedded Development Kit (EDK) [53] software tool generates soft microprocessors from C code. Accel DSP [17] translates MATLAB [48] algorithms into HDL, logic synthesis translates this HDL into a netlist, a synthesis tool uses a place-and-route algorithm to convert this netlist into a bitstream, with the final result being an implementation of a specialized signal processing core.

Given that all of these different design tools produce a set of inter-operating cores, you can only trust your final system as much as you trust your least-trusted design path. If there is a critical piece of functionality, e.g. a unit that protects and operates on secret keys, there is no way to verify that this core cannot be snooped on or tampered without a set of isolation strategies.

The subversion of design tools could easily result in malicious hardware being loaded onto the device. In fact, major design tool developers have few or no checks in place to ensure that attacks on specific functionality are not included. However, just to be clear, we are not proposing a method that makes possible the use of subverted design tools on a trusted core. Rather, we are proposing a method by which small trusted cores, developed with trusted tools (perhaps using in-house tools which are not fully optimized for performance[5]) can be safely combined with untrusted cores.

2.3 Motivating Examples

We have already discussed the example of a system with two processor cores and an encryption core. The goal of our methods is to prevent the encryption key for one of the processors from being obtained by the other processor by either

[4]Essentially a list of logical gates and their interconnections

[5]FPGA manufacturers such as Xilinx provide signed cores that can be trusted by embedded designers, while those freely available cores obtained from sources such as OpenCores are considered to be less trustworthy. The development of a trusted tool chain or a trusted core is beyond the scope of this paper.

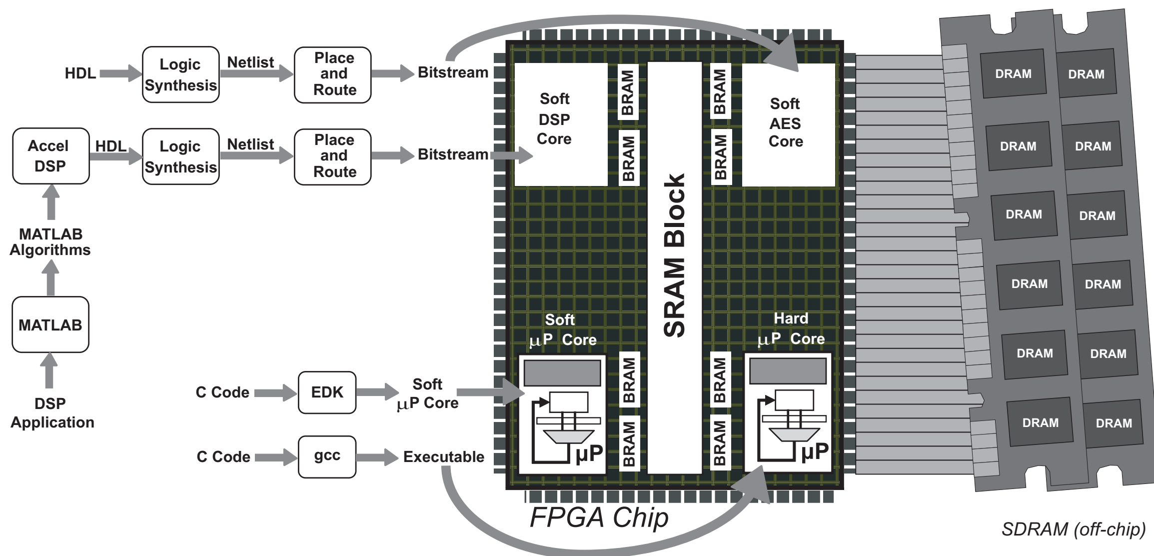

Figure 1. A Modern FPGA-based Embedded System: Distinct cores with different pedigrees and varied trust requirements find themselves occupying the same silicon. Reconfigurable logic, hard and soft processor cores, blocks of SRAM, and other soft IP cores all share the FPGA and the same off-chip memory. How can we ensure that the encryption key for one of the processors cannot be obtained by the other processor by either reading the key from external memory or directly from the encryption core itself?

reading the key from external memory or directly from the encryption core itself.

Aviation – Both military and commercial sectors rely on commercial off-the-shelf (COTS) reconfigurable components to save time and money. Consider the example of avionics in military aircraft in which sensitive targeting data is processed on the same device as less sensitive maintenance data. In such military hardware systems, certain processing components are "cleared" for different levels of data. Since airplane designs must minimize weight, it is impractical to have a separate device for every function. Our security primitives can facilitate the design of military avionics by providing separation of modules that must be integrated onto a single device.

Computer Vision – In the commercial world, consider a video surveillance system that has been designed to protect privacy. Intelligent video surveillance systems can identify human behavior that is potentially suspicious, and this behavior can be brought to the attention of a human operator to make a judgment [40] [21]. IBM's PeopleVision project has been developing such a video surveillance system [46] that protects the privacy of individuals by blurring their faces depending on the credentials of the viewer (e.g., security guards vs. maintenance technicians). FPGAs are a natural choice for any streaming application because they can provide deep regular pipelines of computation, with no shortage of parallelism. Implementing such a system would require at least three cores on the FPGA: a video interface for decoding the video stream, a redaction mechanism for blurring faces in accordance with a policy, and a network interface for sending the redacted video stream to the security guard's station. Each of these modules would need buffers of off-chip memory to function, and our methods could prevent sensitive information from being shared between modules improperly (e.g. directly between the video interface and the network). While our techniques could not verify the correct operation of the redaction core, they could ensure that only the connections necessary for legal communication between cores are made.

Now that we have described a high level picture of the problem we are attempting to address, we present our two concepts, moats and drawbridges, along with the details of how each maps to a modern reconfigurable device. In particular, for each approach we specify the threats that it addresses, the details of the technique and its implementation, and the overheads involved in its use. Finally, in Section 5, we show how these low-level protection mechanisms can be used in the implementation of a higher-level memory pro-

tection primitive.

3 Physical Isolation with Moats

As discussed in Section 2, a strong notion of isolation is lacking in current reconfigurable hardware design flows, yet one is needed to be certain that cores are not snooping on or interfering with each other. Before we can precisely describe the problem that moats attempt to solve, we need to begin with a brief description of how routing works (and the function it serves) in a modern FPGA.

On a modern FPGA, the vast majority of the actual silicon area is taken up by interconnect (approximately 90%). The purpose of this interconnect is to make it easy to connect logical elements together so that any circuit can be realized. For example, the output of one NAND gate may be routed to the input of another, or the address wires from a soft-processor may be routed to an I/O pad connected to external memory. The routing is completely static: a virtual wire is created from input to output, but that signal may be routed to many different places simultaneously (e.g., one output to many inputs or vice versa).

The rest of the FPGA is a collection of programmable gates (implemented as small lookup-tables called LUTs), flip-flops for timing and registers, and I/O blocks (IOB) for transferring data into and out of the device. A circuit can be mapped to an FPGA by loading the LUTs and switch-boxes with a *configuration*, a method that is analogous to the way a traditional circuit might be mapped to a set of logical gates. An FPGA is programmed using a *bitstream*. This binary data is loaded into the FPGA to execute a particular task. The bitstream contains all the information needed to provide a functional device, such as the configuration interface and the internal clock cycle supported by the device.

Without an isolation primitive, it is very difficult to prevent a connection between two cores from being established. Place-and-route software uses performance as an objective function in its optimization strategy, which can result in the logical elements and the interconnections of two cores to be intertwined. Figure 3 makes the scope of the problem more clear. The left hand of Figure 3 shows the floor plan of an FPGA with two small cores (soft processors) mapped onto it. The two processors overlap significantly in several areas of the chip. Ensuring that the two never communicate requires that we trace every single wire to ensure that only the proper connections are made. Such verification of a large and complex design requires reverse engineering, which is highly impractical because many companies keep the necessary details about their bitstreams secret. With moats, fewer proprietary details about the bitstream are needed to accomplish this verification. The difficulty of this problem is made more clear by the zoom-in on the right of Figure 3. The zoom-in shows a single switch box, the associated LUTs (to the right of the switch box), and all the wires that cross through that one small portion of the chip. A modern FPGA contains on the order of 20,000 or more such boxes.

Isolation is required in order to protect the confidentiality and integrity of a core's data, and helps to prevent interference with a core's functionality. Our technique allows a very simple static check to verify that, at least at the routing layer, the cores are sufficiently isolated.

3.1 Building Moats

Moats are a novel method of enhancing the security of FPGA systems via the physical isolation of cores. Our approach involves surrounding each core with a "moat" that blocks wiring connectivity from the outside. The core can only communicate with the outside world via a "drawbridge", which is a precisely defined path to the outside world.

One straightforward way to accomplish this is to align the routing tracks used by each of these modules and simply disable the switches near the moat boundaries. The problem with this simple approach is that, for the purposes of improving area and timing efficiency, modern FPGA architectures often support staggered, multiple track segments. For example, the Virtex platform supports track segments with lengths 1, 2 and 6, where the length is determined by measuring the number of Configuration Logic Blocks (CLBs) the segment crosses. For example, a length 6 segment will span 6 CLBs, providing a more direct connection by skipping unnecessary switch boxes along the routing path. Moreover, many platforms such as Virtex support "longline" segments, which span the complete row or column of the CLB array.

Figure 4 illustrates our moat architecture. If we allow the design tool to make use of segment lengths of one and two, the moat size must be at least two segments wide in order to successfully isolate two cores (otherwise signals could hop the moats because they would not *require* a switch box in the moat). To statically check that a moat is sound, the following properties are sufficient.

1. The target core is completely surrounded by moat of width at least w

2. The target core does not make *any* use of routing segments longer than length w

In fact, both of these properties are easy to inspect on an FPGA. We can tell if a switch box is part of a moat by simply checking that it is completely dead (i.e., all the routing transistors are configured to be disconnected). We can check the second property by examining all of the long line switch boxes to ensure that they are unused. These are easy

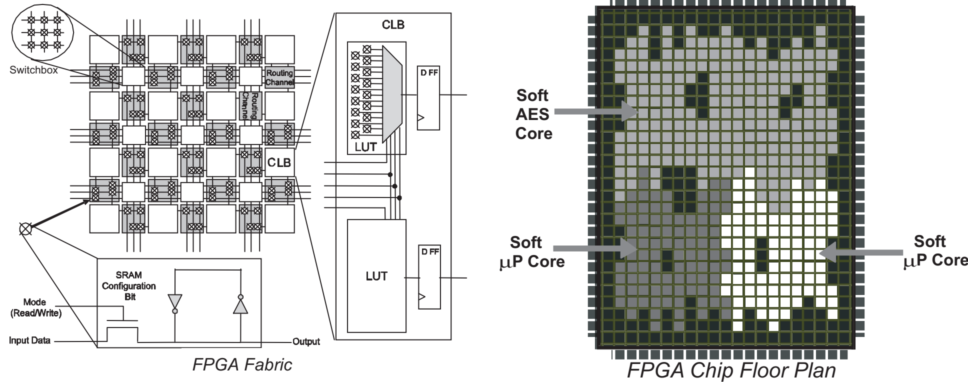

Figure 2. A simplified representation of an FPGA fabric is on the left. Configurable Logic Blocks (CLBs) perform logic level computation using Lookup Tables (LUTs) for bit manipulations and flip-flops for storage. The switch boxes and routing channels provide connections between the CLBs. SRAM configuration bits are used throughout the FPGA (e.g., to program the logical function of the LUTs and connect a segment in one routing channel to a segment in an adjacent routing channel). The FPGA floor plan on the right shows the layout of three cores – notice how they are intertwined.

to find because they are tied to the *physical* FPGA design and are not a function of the specific core on the FPGA.

3.2 A Performance/Area Trade-off

On an FPGA, the *delay* of a connection depends on the number of switch boxes it must pass through rather than the total length. Although large moats consume a great deal of chip area (because they reserve switch boxes without making use of them to perform an operation), they allow the design tools to make use of longer segments, which helps with the area and performance of each individual core. On the other hand, small moats require less chip area (for the moat itself), but having to use small segments negatively affects the area and performance of the cores.

A set of experiments is needed to understand the trade-offs between the size of the moats, the number of cores that can be protected using moats, and the performance and area implications for moat protection.

3.3 The Effect of Constrained Routing

We begin by quantifying the effect of constraining the tools to generate only configurations that do not use *any* routing segments longer than length w. The width of the moat could be any size, but the optimal sizes are dictated by the length of the routing segments. As mentioned before, FPGAs utilize routing segments of different sizes, most commonly 1, 2, 6 and long lines. If we could eliminate the long lines, then we would require a size 6 moat for protecting a core. By eliminating long lines and hex lines, we only need a moat of size 2, and so on.

In order to study the impact of eliminating certain long length segments on routing quality, we compare the routing quality of the MCNC benchmarks [32] on different segment configurations. We use the Versatile Placement and Routing (VPR) toolkit developed by the University of Toronto for such experiments. VPR provides mechanisms for examining trade-offs between different FPGA architectures and is popular within the research community [3]. Its capabilities to define detailed FPGA routing resources include support for multiple segment routing tracks and the ability for the user to define the distribution of the different segment lengths. It also includes a realistic cost model which provides a basis for the measurement of the quality of the routing result.

The effect of the routing constraints on performance and area can vary across different cores. Therefore, we route the 20 biggest applications from the MCNC benchmark set [32] (the de facto standard for such experiments) using four different configurations. The baseline configuration supports segments with length 1, 2, 6 and longlines. The distribution of these segments on the routing tracks are 8%, 20%, 60% and 12% respectively, which is similar to the Xilinx Virtex II platform. The other three configurations are derived from the baseline configurations by eliminating the segments with longer lengths. In other words, configuration 1-2-6 will have no longlines, configuration 1-2 will support segments of length 1 and 2, and configuration 1 will only support segments of length 1.

After performing placement and routing, we measure the

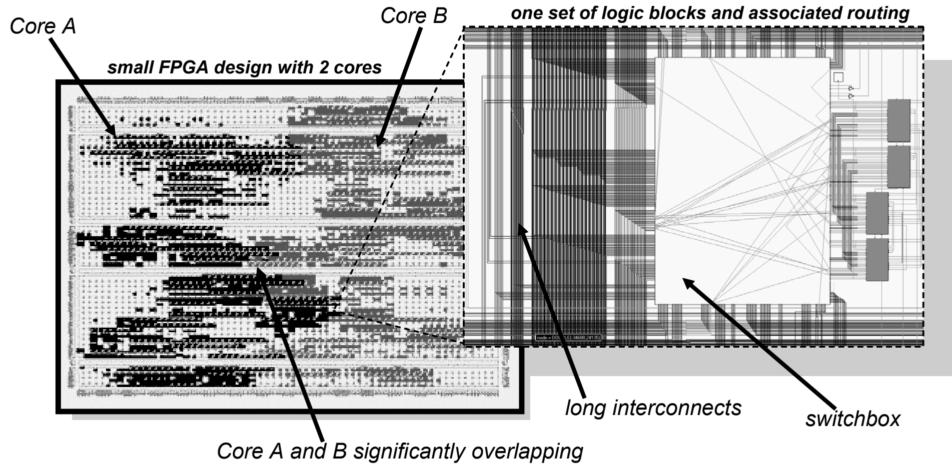

Figure 3. A simple two-core system mapped onto a small FPGA. The zoom-in to the right shows the wiring complexity at each and every switch-box on the chip. To statically analyze a large FPGA with 10s of cores and millions of logical gates, we need to restrict the degrees of freedom. Static verification of a large, complex design involving intertwined cores requires reverse engineering, which is highly impractical because many companies keep the necessary details about their bit-streams a closely guarded trade secret.

quality of the routing results by collecting the area and the timing performance based on the critical path of the mapped application. To be fair, all the routing tracks are configured using the same tri-state buffered switches with Wilton connection patterns [52] within the switch box. A Wilton switch box provides a good trade-off between routability and area, and is commonly used in FPGA routing architectures.

Figures 5 and 6 show the experimental results, where we provide the average hardware area cost and critical path performance for all the benchmarks over four configurations. The existence of longlines has little impact on the final quality of the mapped circuits. However, significant degradation occurs when we eliminate segments of length 2 and 6. This is caused by the increased demand for switch boxes, resulting in a larger hardware cost for these additional switch resources. Moreover, the signal from one pin to another pin is more likely to pass more switches, resulting in an increase in the critical path timing. If we eliminate hex and long lines, there is a 14.9% area increase and an 18.9% increase in critical path delay, on average. If the design performance is limited directly by the cycle time, the delay in critical path translates directly into slowdown.

3.4 Overall Area Impact

While the results from Figures 5 and 6 show that there is some area impact from constraining the routing, there is

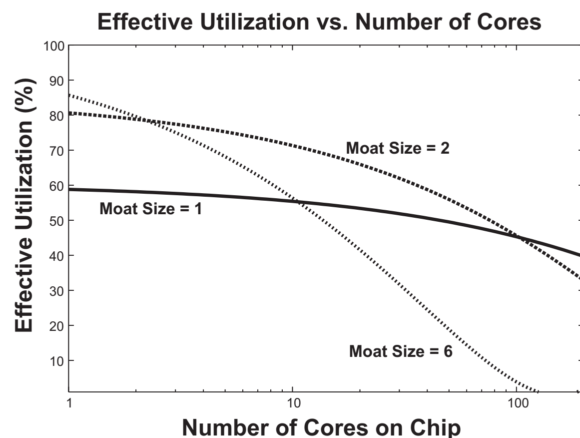

Figure 7. The trade-off between the number of cores, the size of the moat, and the utilization of the FPGA. An increasing number of cores results in larger total moat area, which reduces the overall utilization of the FPGA. Larger moat sizes also will use more area resulting in lower utilization.

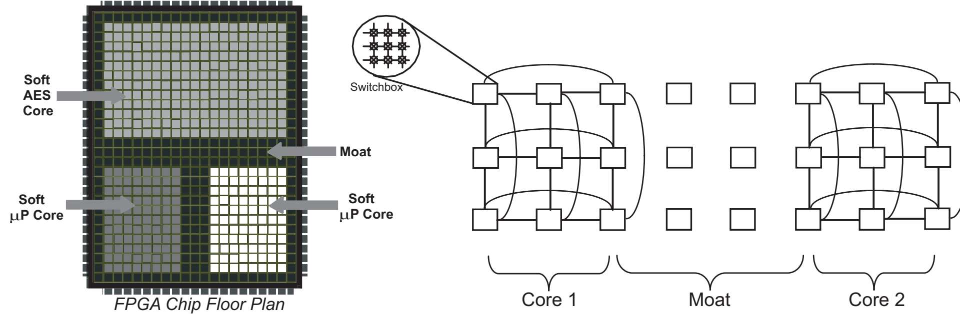

Figure 4. We use moats to physically isolate cores for security. In this example, segments can either span one or two switch boxes, which requires the moat size to have a length of two. Since the delay of a connection on an FPGA depends on the number of switch boxes it must pass through, restricting the length of segments reduces performance, but the moats can be smaller. Allowing longer segments improves performance, but the moats must waste more area.

also a direct area impact in the form of resources required to implement the actual moats themselves. Assuming that we have a fixed amount of FPGA real estate, we really care about how much of that area is used up by a combination of the moats and the core inflation due to restricted routing. We can call this number the effective utilization. Specifically, the effective utilization is:

$$U_{eff} = \frac{A_{AllRoutes}}{A_{RestrictedRoutes} + A_{Moats}}$$

Figure 7 presents the trade-offs between the moat size, the number of isolated cores on the FPGA, and the utilization of the FPGA. The FPGA used for these calculations was a Xilinx Virtex-4 Device which has 192 CLB rows and 116 CLB columns. The figure examines three different moat sizes: 1, 2 and 6 for a variable number of cores on the chip (conservatively assuming that a moat is required around all cores). As the number of cores increases, the utilization of the FPGA decreases since the area of the moats, which is unusable space, increases. However, when a small number of cores is used, a larger moat size is better because it allows us to make more efficient use of the non-moat parts of the chip. If you just need to isolate a single core (from the I/O pads) then a moat of width 6 is the best (consuming 12% of the chip resources). However, as the curve labeled "Moat Size = 2" in Figure 7 shows, a moat width of two has the optimal effective utilization for designs that have between two and 120 cores. As a point of reference, it should be noted that a modern FPGA can hold on the order of 100 stripped down microprocessor cores. The number of cores is heavily dependent on the application, and the trade-off presented here is somewhat specific to our particular platform, but our analysis method is still applicable to other designs. In fact, as FPGAs continue to grow according to Moore's Law, the percent overhead for moats should continue to drop. Because the moats are perimeters, as the size of a core grows by a factor of n, the cost of the moat only grows by $O(\sqrt{n})$.

3.5 Effective Scrubbing and Reuse of Reconfigurable Hardware

Moats allow us to reason about isolation without any knowledge of the inner workings of cores, which are far too complex to feasibly determine whether a particular element of a core is connected to another core. Furthermore, moats also allow us to isolate cores designed with a less trustworthy tool chain from cores that are the result of a more trustworthy tool chain. While these are both useful properties, we need to make sure we can actually implement them. In fact, a few of the latest FPGAs available have the ability to change a selective part of their configuration, one column at a time [34]. A specialized core on the FPGA can read one frame of the configuration, change part of this frame, and write the modified frame back. This core must therefore be part of the trusted computing base of the system.

Partial reconfiguration improves the flexibility of a system by making it possible to swap cores. If the number of possible configurations is small, then static verification is sufficient, but if the space of possible cores is infinite, then dynamic verification is necessary. For example, Baker et al. have developed an intrusion detection system based on reconfigurable hardware that dynamically swaps the detection cores [2] [1]. Since the space of intrusion detection rule sets is infinite, the space of detection cores is also infinite. Huffmire et al. have developed a memory protection scheme for reconfigurable hardware in which a reconfigurable reference monitor enforces a policy that specifies the legal sharing of memory [19]. Partial reconfiguration

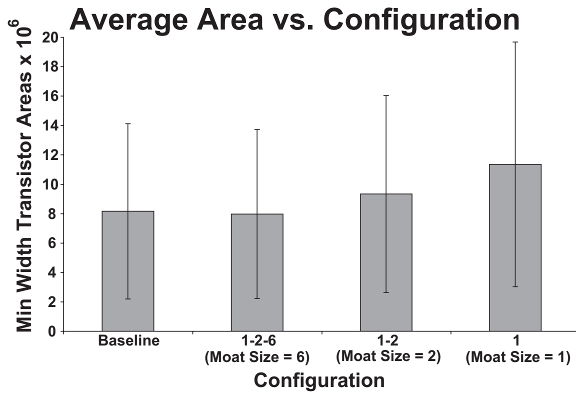

Figure 5. Comparison of area for different configurations of routing segments. The baseline system has segments with length 1, 2, 6 and longline. The distribution is close to that of Virtex II: 8% (1), 20% (2), 60% (6) and 12% (longline). Other configurations are created by eliminating one or more classes of segments. For example, configuration 1-2-6 removes the longlines and distributes them proportionally to other types of segments.

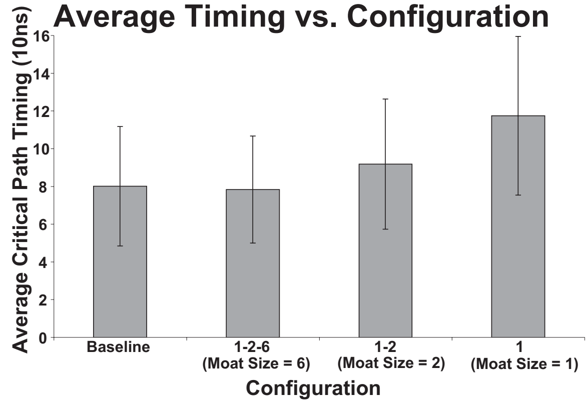

Figure 6. Comparison of critical path timing for different configurations of routing segments. Unlike Figure 7, the graphs in Figures 5 and 6 do not include the overhead of the moat itself. The error bars show one standard deviation.

could allow the system to change the policy being enforced by swapping in a different reference monitor. Since the space of possible policies is infinite, the space of possible reference monitors is also infinite. Lysaght and Levi have devised a dynamically reconfigurable crossbar switch [33]. By using dynamic reconfiguration, their 928x928 crossbar uses 4,836 CLBs compared to the 53,824 CLBs required without reconfiguration.

To extend our model of moats to this more dynamic case, we not only need to make sure that our static analysis must be simple enough to be performed on-line by a simple embedded core (which we argue it is), but we also need to make sure that nothing remains of the prior core's logic when it is replaced with a different core. In this section, we describe how we can enable object reuse through configuration cleansing.

By rewriting a selective portion of the configuration bits for a certain core, we can erase any information it has stored in memory or registers. The ICAP (Internal Configuration Access Port) on Xilinx devices allows us to read, modify, and write back the configuration bitstream on Virtex II devices. The ICAP can be controlled by a Microblaze soft core processor or an embedded PowerPC processor if the chip has one. The ICAP has an 8-bit data port and typically runs at a clock speed of 50 MHz. Configuration data is read and written one frame at a time. A frame spans the entire height of the device, and frame size varies based on the device.

Table 1 gives some information on the size and number of frames across several Xilinx Virtex II devices. The smallest device has 404 frames, and each frame requires 5.04 us to reconfigure, or equivalently, erase. Therefore, reconfiguring (erasing) the entire devices takes around 2 ms.

To sanitize a core we must perform 3 steps. First we must read in a configuration frame. The second step is to modify the configuration frame so that the flip-flops and memory are erased. The last step is to write back the modified configuration frame. The number of frames and how much of the frame we must modify depend on the size of the core that is being sanitized. This process must be repeated since each core will span the width of many frames. In general, the size of the core is linearly related to the time that is needed to sanitize it.

Our object reuse technique can also disable a core if extreme circumstances should require it, such as tampering. Embedded devices such as cell phones are very difficult to sanitize [38]. Smart phones contain valuable personal data, and the theft or loss of a phone can result in serious consequences such as identity theft. Embedded devices used by the military may contain vital secrets that must never fall into enemy hands. Furthermore, valuable IP information of the cores is stored in the form of the bitstream on the FPGA. A method of disabling all or part of the device is needed to protect important information stored on the FPGA in the

Table 1. Reconfiguration Time

Device	# Frames	Frame Length (32-bit words)	R/W time for 1 frame (ICAP@50 Mhz)
XC2V40	404	26	5.04 us
XC2V500	928	86	14.64 us
XC2C2000	1456	146	24.24 us
XC2V8000	2860	286	46.64 us

extreme case of physical tampering.

The IBM 4758 is an example of a cryptographic coprocessor that has been designed to detect tampering and to disable itself whenever tampering occurs [51]. The device is surrounded by specialized packaging containing wire mesh. Any tampering of the device disturbs this mesh, and the device can respond by disabling itself.

4 Drawbridges: Interconnect Interface Conformance with Tracing

In the previous section, we described an effective method for isolating cores using moats. Our moat methodology eliminates the possibility for external cores to tap into the information contained in a core surrounded by the moat. However, cores do not work in isolation and must communicate with other cores to receive and send data. Therefore, we must allow controlled entry into our core. The entry or communication is only allowed with prespecified transactions through a "drawbridge". We must know in advance which cores we need to communicate with and the location of those cores on the FPGA. Often times, it is most efficient to communicate with multiple cores through a shared interconnection (i.e., a bus). Again, we must ensure that bus communications are received by only the intended recipient(s). Therefore, we require methods to ensure that 1) communication is established only with the specified cores and 2) communication over a shared medium does not result in a covert channel. In this section, we present two techniques, interconnect tracing and a bus arbiter, to handle these two requirements.

We have developed an interconnect tracing technique for preventing unintended flows of information on an FPGA. Our method allows a designer to specify the connections on a chip, and a static analysis tool checks that each connection only connects the specified components and does not connect with anything else. This interconnect tracing tool takes a bitstream file and a text file that defines the modules and interconnects in a simple language which we have developed. The big advantage of our tool is that it allows us to perform the tracing on the bitstream file. We do not require a higher level description of the design of the core. Performing this analysis during the last stage of design allows us to catch illegal connections that could have originated from any stage in the design process including the design tools themselves.

In order for the tracing to work we must know the locations of the modules on the chip and the valid connections to/from the modules. To accomplish this we place moats around the cores during the design phase. We now know the location of the cores and the moats, and we use this information to specify a text file that defines: all the cores along with their location on the chip, all I/O pins used in the design, and a list of valid connections. Then our tool uses the JBits API [13] to analyze the bitstream and check to make sure there are no invalid connections in the design. The process of interconnect tracing is performed by analyzing the bitstream to determine the status of the switchboxes. We can use this technique to trace the path that a connection is routed along and ensure that it goes where it is supposed to. This tracing technique allows us to ensure that the different cores can only communicate through the channels we have specified and that no physical trap doors have been added anywhere in the design.

Ensuring that interconnects between modules are secure is a necessity to developing a secure architecture. This problem is made more complicated by the abundance of routing resources on an FPGA and the ease with which they can be reconfigured. Our proposed interconnect tracing technique allows us to ensure the integrity of connections on a reconfigurable device. This tool gives us the ability to perform checking in the final design stage: right before the bitstream is loaded onto the device.

4.1 Efficient Communication under the Drawbridge Model

In modern reconfigurable systems, cores communicate with each other via a shared bus. Unfortunately, the shared nature of a traditional bus architecture raises several security issues. Malicious cores can obtain secrets by snooping on the bus. In addition, the bus can be used as a covert channel to leak secret data from one core to another. The ease of reconfigurability on FPGAs allows us to address these problems at the hardware level.

To address this problem of covert channels and bus snooping, we have developed a *shared memory bus* with

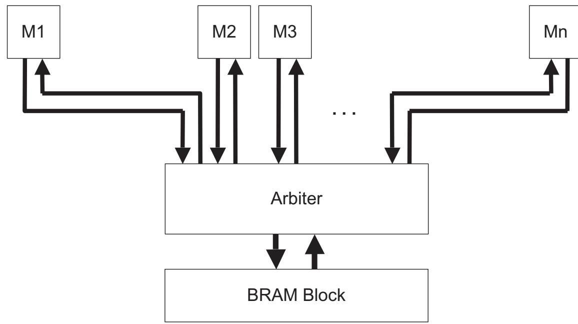

Figure 8. Architecture alternative 1. There is a single arbiter and each module has a dedicated connection to the arbiter.

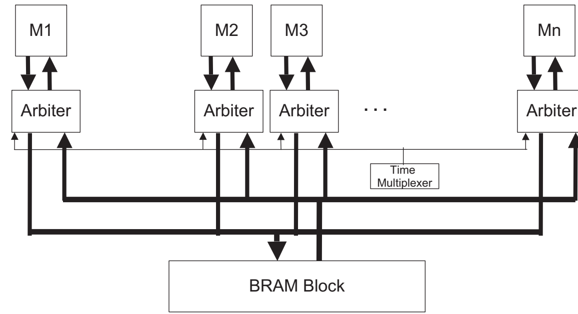

Figure 9. Architecture alternative 2. Each module has its own arbiter that prevents bus snooping and a central time multiplexer that connects to all the arbiters.

a *time division access*. The bus divides the time equally among the modules, and each module can read/write one word to/from the shared memory during its assigned time slice. Our approach of arbitrating by time division eliminates covert channels. With traditional bus arbitration, there is a possibility of a bus-contention covert channel to exist in any shared bus system where multiple cores or modules access a shared memory. Via this covert channel, a malicious core can modulate its bus references, altering the latency of bus references for other modules. This enables the transfer of information between any two modules that can access the bus [18]. This covert channel could be used to send information from a module with a high security clearance to a module with lower security clearance (write-down), which would violate a Bell-LaPadula multilevel policy and cannot be prevented through the use of the reference monitor. To eliminate this covert channel, we give each module an equal share of time to use the bus, eliminating the transfer of information by modulating bus contention. Since each module can only use the bus during its alloted time slice, it has no way of changing the bus contention. One module cannot even tell if any of the other modules are using the bus. While this does limit performance of the bus, it removes the covert channel. The only other feasible way that we see to remove this covert channel is to give each module a dedicated connection to all other modules. Requiring a dedicated direct connection between each set of modules that need to communicate would be inefficient and costly. Dedicated channels would require a worst case of $O(2^n)$ connections, where n is the number of modules in the design. Our architecture requires only $O(n)$ connections.

Bus snooping is another major concern associated with a shared bus. Even if we eliminate the covert channels there is nothing to prevent bus snooping. For example, let us consider a system where we want to send data from a classified module to another and where there are unclassified modules on the same bus. We need a way to ensure that these less trusted modules cannot obtain this information by snooping the bus. To solve this problem, we place an arbiter between the module and the memory. The arbiter only allows each module to read during its time share of the bus. In addition a memory monitor is required, but for this work we assume that such a configuration can be implemented on the FPGA using the results of Huffmire et. al.[19]

4.2 Architecture Alternatives

We devised two similar architectures to prevent snooping and to eliminate covert channels on the bus. In our first architecture, each module has its own separate connection to a single arbiter, which sits between the shared memory and the modules. This arbiter schedules access to the memory equally according to a time division scheduling (Figure 8). A module is only allowed to read or write during its aloted time, and when a module reads, the data is only sent to the module that issued the read request. The second architecture is more like a traditional bus. In this design, there is an individual arbiter that sits between each module and the bus. These arbiters are all connected to a central timing module which handles the scheduling (Figure 9). The individual arbiters work in the same way as the single arbiter in the first architecture to prevent snooping and to remove covert channels. To make interfacing easy, both of these architectures have a simple interface so that a module can easily read/write to the shared memory without having to worry about the timing of the bus arbiter.

During the design process, we found that the first architecture seemed easier to implement, but we anticipated that the second architecture would be more efficient. In our first architecture (Figure 8, everything is centralized, making the design of a centralized memory monitor and arbiter much easier to design and verify. In addition, a single moat could

be used to isolate this functionality. Our second architecture (Figure 9) intuitively should be more scalable and efficient since it uses a bus instead of individual connections for each module, but the arbiters have to coordinate, the memory monitor has to be split (if that is even possible), and each arbiter need to be protected by its own moat.

To test our hypotheses, we developed prototypes of both of the architectures. The prototypes were developed in VHDL and synthesized for a Xilinx Virtex-II device in order to determine the area and performance of the designs on a typical FPGA. We did not account for the extra moat or monitor overhead, but with this assumption results of the analysis of the two architectures, which can be seen in Table 2, were not what we first expected. During synthesis of the second architecture, the synthesis tool converted the tristate buffers[6] in the bus to digital logic. As a result, the second architecture used more area than the first and only had a negligible performance advantage. Contrary to what we expected, the first architecture used roughly 15% less area on the FPGA and is simpler to implement and verify. Since the peformance difference between the two was almost negligible, the first architecture is the better design choice.

This bus architecture allows modules to communicate securely with a shared memory and prevents bus snooping and certain covert channels. When combined with the reference monitor this secure bus architecture provides a secure and efficient way for modules to communicate.

5 Application: Memory Policy Enforcement

Now that we have described isolation and its related primitives, we provide an example of the application of isolation to memory protection, an even higher-level primitive. Saltzer and Schroeder identify three key elements that are necessary for protection: “Conceptually, then, it is necessary to build an impenetrable wall around each distinct object that warrants separate protection, construct a door in the wall through which access can be obtained, and post a guard at the door to control its use.” [43]. In addition, the guard must be able to identify the authorized users. In the case of protecting cores, our moat primitive is analogous to the wall, and our drawbridge primitive is analogous to the door. Our interconnect tracing and secure bus primitives act as the guard.

One way of protecting memory in an FPGA system is to use a reference monitor that is loaded onto the FPGA along with the other cores [19]. Here, the reference monitor is analogous to the guard because it decides the legality of every memory access according to a policy. This requires that every access go through the reference monitor. Without our isolation primitive, it is easy for a core to bypass the reference monitor and access memory directly. Since moats completely surround a core except for a small amount of logic (the drawbridge) for communicating with the rest of the chip, it is much easier to prevent a core from bypassing the reference monitor.

Saltzer and Schroeder describe how protection mechanisms can protect their own implementations in addition to protecting users from each other [43]. Protecting the reference monitor from attack is critical to the security of the system, but the fact that the reference monitor itself is reconfigurable makes it vulnerable to attack by the other cores on the chip. However, moats can mitigate this problem by providing physical isolation of the reference monitor.

Our isolation primitive also makes it harder for an unauthorized information flow from one core to another to occur. Establishing a direct connection between the two cores would clearly thwart the reference monitor. If moats surround each core, it is much harder to connect two cores directly without crossing the moat.

As we described above, a reference monitor approach to memory protection requires that every memory access go through the reference monitor. However, cores are connected to each other and to main memory by means of a shared bus. As we explained in Section 4.1, the data on a shared bus is visible to all cores. Our secure bus primitive protects the data flowing on the bus by controlling the sharing of the bus with a fixed time division approach.

A memory protection system that allows dynamic policy changes requires an object reuse primitive. It is often useful for a system to be able to respond to external events. For example, during a fire, all doors in a building should be unlocked without exception (a more permissive policy than normal), and all elevators should be disabled (a less permissive policy than normal). In the case of an embedded device, a system under attack may wish to change the policy enforced by its reference monitor. There are several ways to change polices. One way is to overwrite the reference monitor with a completely different one. Our scrubbing primitive can ensure that no remnants of the earlier reference monitor remain. Since cores may retain some information in their local memory following a policy change, our scrubbing primitive can also be used to cleanse the cores.

6 Related Work

There has always been an important relationship between the hardware a system runs on and the security of that system. Reconfigurable systems are no different, although to the best of our knowledge we are the first to address the problem of isolation and physical interface conformance on them. However, in addition to the related work we have already mentioned, we do build on the results of prior related

[6]tri-state buffers are gates that can output either a 0, 1, or Z – a high impedance state in which the gate acts as if it was disconnected from the wire.

Table 2. Comparison of Communication Architectures

	Architecture 1	Architecture 2	Percent Difference
Slices	146	169	15.75
Flip Flops	177	206	16.38
4 Input LUTs	253	305	20.55
Maximum Clock Frequency	270.93	271.297	0.14

efforts. In particular, we build on the ideas of reconfigurable security, IP protection, secure update, covert channels, direct channels, and trap doors. While a full description of all prior work in these areas is not possible, we highlight some of the most related.

6.1 Reconfigurable Hardware Security

The closest work to ours is the work of Huffmire et. al. To provide memory protection on an FPGA, Huffmire et al. propose the use of a reconfigurable reference monitor that enforces the legal sharing of memory among cores [19]. 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. While their work addresses the specifics of how to construct a memory access monitor efficiently in reconfigurable hardware, they do not address the problem of how to protect that monitor from routing interference, nor do they describe how to enforce that *all* memory accesses go through this monitor. This paper directly supports their work by providing the fundamental primitives that are needed to implement memory protection on a reconfigurable device.

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 [8]. Their design achieves process isolation by providing a reconfigurable virtual machine to each process, and their architecture uses hardwired 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 [31] or virtual secure co-processing [29].

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 [12]. 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.

Most of the work relating to FPGA security has been targeted at the problem of preventing the theft of intellectual property and securely uploading bitstreams in the field. Because such attacks directly impact their bottom line, industry has already developed several techniques to combat the theft of FPGA IP, such as encryption [6] [23] [24], fingerprinting [27], and watermarking [28]. 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 [16] [15]. 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 [14]. However, these authentication techniques merely ensure that a bitstream is authentic. An "authentic" bitstream could contain a subverted core that was designed by a third party.

6.2 Covert Channels, Direct Channels, and Trap Doors

The work in Section 4.1 directly draws upon the existing work on covert channels. 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, or error conditions [47] [41]. 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 [25]. Storage channels can be mitigated by partitioning the resources, while timing channels can be mitigated with sequential access, a fact we exploit in the construction of our bus architecture. Examples of remedial action include decreasing the bandwidth (e.g., the introduction of artificial spikes (noise) in resource usage [44]) or closing the channel. Unfortunately, an adversary can extract a signal from the noise, given sufficient resources [37].

Of course our technique is primarily about restricting the opportunity for direct channels and trap doors [49]. Our memory protection scheme 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.

7 Conclusion

The design of reconfigurable systems is a complex process, with multiple software tool chains that may have different trust levels. Since it is not cost-effective to develop an optimized tool chain from scratch to meet assurance needs, only the most sensitive cores should be designed using a trusted tool chain. To meet performance needs, most cores could be designed with commodity tools that are highly optimized but untrusted, which results in multiple cores on a chip with different trust levels. Our methodology will not lead to those less trusted portions becoming more dependable or correct, but it will isolate trusted portions from the effects of their subversion or failure. To address this situation, developers will need to build monitored or fail-safe systems on top of FPGAs to prevent the theft of critical secrets.

We have presented two low-level protection mechanisms to address these challenges, moats and drawbridges, and we have analyzed the trade-offs of each. Although larger moats consume more area than smaller moats, they have better performance because longer segments can be used. Our interconnect tracing primitive works together with our moat primitive in a complementary way by allowing smaller moats to be used without sacrificing performance. We have also described how these basic primitives are useful in the implementation of a higher-level memory protection primitive, which can prevent unintended sharing of information in embedded systems.

Acknowledgments

The authors would like to thank Virgil Gligor for his insightful comments on this paper. We also wish to thank the anonymous reviewers for their helpful feedback. This research was funded in part by National Science Foundation Grant CNS-0524771, NSF Career Grant CCF-0448654, and the SMART Defense Scholarship for Service.

References

[1] Z. Baker and V. Prasanna. Efficient architectures for intrusion detection. In *Twelfth Annual International Conference on Field-Programmable Logic and its Applications (FPL '04)*, August 2004.

[2] Z. Baker and V. Prasanna. Computationally-efficient engine for flexible intrusion detection. October 2005.

[3] V. Betz, J. S. Rose, and A. Marqardt. *Architecture and CAD for deep-submicron FPGAs*. Kluwer Academic, Boston, MA, 1999.

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

[5] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan. Parallel fpga-based all-pairs shortest-paths in a directed graph. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, April 2006.

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

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

[8] A. Chien and J. Byun. Safe and protected execution for the morph/AMRM reconfigurable processor. In *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, April 1999.

[9] 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.

[10] 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.

[11] 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.

[12] G. Gogniat, T. Wolf, and W. Burleson. Reconfigurable security support for embedded systems. In *Proceedings of the 39th Hawaii International Conference on System Sciences*, 2006.

[13] S. Guccione, D. Levi, and P. Sundararajan. Jbits: Java-based interface for reconfigurable computing. In *Proceedings of the Second Annual Conference on Military and Aerospace Applications of Programmable Logic Devices and Technologies (MAPLD)*, Laurel, MD, USA, September 1999.

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

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

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

[17] T. Hill. Acceldsp synthesis tool floating-point to fixed-point conversion of matlab algorithms targeting fpgas, April 2006.

[18] W.-M. Hu. Reducing timing channels with fuzzy time. In *IEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, May 1991.

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

[20] B. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, 2002.

[21] A. Jain, D. Koppel, K. Kaligian, and Y.-F. Wang. Using stationary-dynamic camera assemblies for wide-area video surveillance and selective attention. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2006.

[22] R. Kastner, A. Kaplan, and M. Sarrafzadeh. *Synthesis Techniques and Optimizations for Reconfigurable Systems*. Kluwer Academic, Boston, MA, 2004.

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

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

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

[26] 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.

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

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

[29] R. Lee, P. Kwan, J. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, June 2005.

[30] J. Lewis and B. Martin. Cryptol: High assurance, retargetable crypto development and validation. In *Military Communications Conference (MILCOM)*, October 2003.

[31] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA, November 2000.

[32] B. Lisanke. Logic synthesis and optimization benchmarks. Technical report, Microelectronics Center of North Carolina, Research Triangle Park, NC, USA, January 1991.

[33] P. Lysaght and D. Levi. Of gates and wires. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[34] P. Lysaght and J. Stockwood. A simulation tool for dynamically reconfigurable field programmable gate arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(3), September 1996.

[35] W. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. Prasanna, and H. Spaanenburg. Seeking solutions in configurable computing. In *Computer*, volume 30(12), pages 38–43, 1997.

[36] D. McGrath. Gartner dataquest analyst gives asic, fpga markets clean bill of health. *EE Times*, June 13 2005.

[37] J. Millen. Covert channel capacity. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, 1987.

[38] E. Nakashima. Used cellphones hold trove of secrets that can be hard to erase. *Washington Post*, October 21 2006.

[39] H. Ngo, R. Gottumukkal, and V. Asari. 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*, 2005.

[40] W. Niu, J. Long, D. Han, and Y.-F. Wang. Human activity detection and recognition for video surveillance. In *Proceedings of the IEEE Multimedia and Expo Conference*, Taipei, Taiwan, 2004.

[41] C. Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottowa, Ontario, Canada, 2005.

[42] B. Salefski and L. Caglar. Reconfigurable computing in wireless. In *Proceedings of the Design Automation Conference (DAC)*, 2001.

[43] J. Saltzer and M. Schroeder. The protection on information in computer systems. *Communications of the ACM*, 17(7), July 1974.

[44] H. Saputra, N. Vijaykrishnan, M. Kandemir, M. Irwin, R. Brooks, S. Kim, and W. Zhang. Masking the energy behavior of des encryption. In *IEEE DEsign Automation and Test in Europe (DATE '03)*, 2003.

[45] 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.

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

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

[48] The Math Works Inc. MATLAB User's Guide, 2006.

[49] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8), 1984.

[50] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, 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.

[51] S. Weingart and S. Smith. Building a high-performance, programmable secure coprocessor. *Computer Networks (Special Issue on Computer Network Security)*, 31:831–860, April 1999.

[52] S. Wilton. *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*. PhD thesis, University of Toronto, 1997.

[53] Xilinx Inc. Getting Started with the Embedded Development Kit (EDK), 2006.