Security Primitives for Reconfigurable Hardware-Based Systems

TED HUFFMIRE, TIMOTHY LEVIN, THUY NGUYEN, and CYNTHIA IRVINE
Naval Postgraduate School
BRETT BROTHERTON
Special Technologies Laboratory
GANG WANG
Intuit
TIMOTHY SHERWOOD
University of California, Santa Barbara
and
RYAN KASTNER
University of California, San Diego

10

Computing systems designed using reconfigurable hardware are increasingly composed using a number of different Intellectual Property (IP) cores, which are often provided by third-party vendors that may have different levels of trust. Unlike traditional software where hardware resources are mediated using an operating system, IP cores have fine-grain control over the underlying reconfigurable hardware. To address this problem, the embedded systems community requires novel security primitives that address the realities of modern reconfigurable hardware. In this work, we propose security primitives using ideas centered around the notion of "moats and drawbridges." The primitives encompass four design properties: logical isolation, interconnect traceability, secure reconfigurable broadcast, and configuration scrubbing. Each of these is a fundamental

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.
Authors' addresses: T. Huffmire, T. Levin, T. Nguyen, and C. Irvine, Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943; email: {tdhuffmi, televin, tdnguyen, irvine}@nps.edu; B. Brotherton, Special Technologies Laboratory, Santa Barbara, CA 93111; email: brett.brotherton@gmail.com; G. Wang, Intuit, San Diego, CA 92122; email: Gang_Wang@intuit.com; T. Sherwood, Department of Computer Science, University of California, Santa Barbara, CA 93106; email: sherwood@cs.ucsb.edu; R. Kastner, Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093; email: kastner@cs.ucsd.edu.

operation with easily understood formal properties, yet they map cleanly and efficiently to a wide variety of reconfigurable devices. We carefully quantify the required overheads of the security techniques on modern FPGA architectures across a number of different applications.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Virtual memory*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate arrays*; B.7.2 [**Integrated Circuits**]: Design Aids—*Placement and routing*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Authentication*

General Terms: Design, Security

Additional Key Words and Phrases: Field Programmable Gate Arrays (FPGAs), Advanced Encryption Standard (AES), memory protection, separation, isolation, controlled sharing, hardware security, reference monitors, execution monitors, enforcement mechanisms, security policies, static analysis, security primitives, Systems-on-a-Chip (SoCs)

ACM Reference Format:
Huffmire, T., Levin, T., Nguyen, T., Irvine, C., Brotherton, B., Wang, G., Sherwood, T., Kastner, R. 2010. Security primitives for reconfigurable hardware-based systems. ACM Trans. Reconfig. Technol. Syst. 3, 2, Article 10 (May 2010), 35 pages. DOI = 10.1145/1754386.1754391. http://doi.acm.org/10.1145/1754386.1754391.

1. INTRODUCTION

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 trustworthy. Reconfigurable systems are typically designed using a collection of Intellectual Property (IP) cores in order to save both time and money. Ideally each of these cores would be formally specified, tested, and verified by a highly trusted party. However, in reality, this is rarely the case. Unlike uniprocessor 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 has explicit control of 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 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: "Are these two cores capable of communication?" is not simple.

Consider a more concrete example, a cryptographic system that performs local authentication to encrypt and decrypt network traffic. This system is designed using two soft-processor cores—one to interface with an authentication device (e.g., fingerprint reader), the other to control the ethernet IP core—and an AES encryption engine used by both of the processor cores. These cores are all implemented using a single FPGA. Further details about this system can be found in Huffmire et al. [2008], and in Section 4.3.1. Each of 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 an obfuscated network 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 guarantee that all cores are communicating only through their specified interfaces. To confirm that 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. The techniques presented in this article 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 such that this separation can be easily verified. 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. Once we have a set of drawbridges, we need to enable legal intercore communication. We describe two secure reconfigurable communication architectures, and we quantify the implementation trade-offs between them in terms of complexity of analysis and performance.

In this article, we extend our preliminary work [Huffmire et al. 2007] to incorporate an alternative form of moats that does not require "dead" areas or the disabling of longer routing segments. Rather than using physical/spatial separation, this improved form of moats uses logical separation. We compare these two versions of moats to determine which is optimal.

2. SEPARATION

The concepts of isolation and separation are fundamental to computer security. Saltzer and Schroeder [1974] define complete isolation as a "protection system that separates principals into compartments between which no flow of information or control is possible." However, no system can function if all of its components are completely isolated from each other. Therefore, a separation technique that allows the controlled sharing of data among isolated components is needed. In a system with a mandatory access control policy, we need to isolate *equivalence classes* of objects (e.g., all top secret objects) and

control their interaction (e.g., with unclassified objects). To achieve separation in FPGA systems, we propose a solution where moats provide the isolation, and drawbridges provide a means of controlled sharing.

Consider again the aforementioned cryptographic network computing 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 potentially sensitive 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? This system consists of two compartments with an AES core that is shared between the two domains. One domain (gray) contains one of the processors as well as an RS-232 (serial) interface used to communicate with the local authentication device. The other domain (black) contains the other processor as well as an Ethernet interface. All of these components are connected over a shared bus. Since RS-232 is a local connection, the domain containing the serial interface can serve a highly trusted function, such as biometric authentication with an iris scanner or fingerprint reader.

3. RECONFIGURABLE SYSTEMS

We are seeing reconfigurable devices emerge as the flexible and high-performance workhorses inside a variety of high-performance embedded computing systems [Bondalapati and Prasanna 2002; Compton and Hauck 2002; DeHon and Wawrzynek 1999; Kastner et al. 2004; Mangione-Smith et al. 1997; Schaumont et al. 2001]. To understand the potential security issues, we describe a modern device, a typical design flow, and the potential threats that our techniques are expected to handle.

3.1 The Composition Problem

Increasingly, soft-processors and other IP cores[1] are composed to implement the desired functionality of a reconfigurable system. Cores are often 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 can be formally verified [Lewis and Martin 2003], 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 research 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 design.

3.2 Reconfigurable Hardware Security

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

[1]Since designing reconfigurable modules is costly, companies have developed several schemes to protect this valuable intellectual property, which we discuss in Section 8.

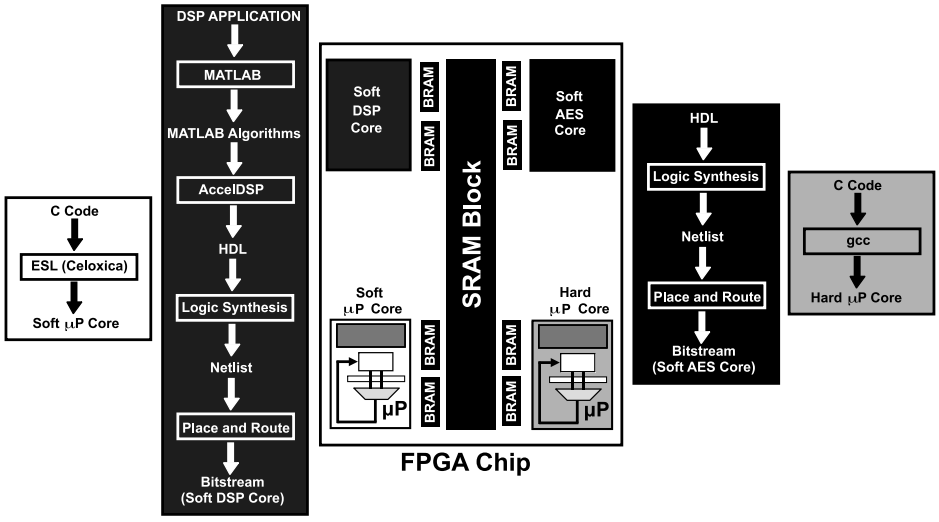

Fig. 1. FPGA design flows: Distinct cores with different provenance trust properties occupy the same chip. Reconfigurable logic, hard- and soft-processor cores, blocks of on-chip SRAM and BRAM, and other IP cores all share the FPGA and the same off-chip memory.

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 [Kocher et al. 2004]. In this article 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.

3.3 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 across many organizations. IP cores, such as an AES core, can be distributed in the form of Hardware Description Language (HDL), netlists, 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) [Xilinx, Inc. 2006] software tool generates custom microprocessors cores and compiles C code to these soft or hard microprocessors. Accel DSP [Hill 2006] translates MATLAB [The Math Works, Inc. 2006] algorithms into HDL, logic synthesis translates this HDL into a netlist, physical synthesis converts this netlist into a bitstream, with the final result being an implementation of a synthesized IP processing core.

Given that all of these different design tools produce a set of interoperating cores, you can only trust your final system as much as you trust your least-trusted design path. A separation mechanism is needed to protect sensitive data from being compromised. For example, we must prevent secret keys and plaintext from being extracted from a crypto processing core.

3.4 Assumptions, Threats, and Scope

Many critical hardware components are manufactured in foundries that are located in countries where the cost to build and run a foundry is competitive. The problem of hardware subversion is a serious concern that has led to the establishment of the DARPA Trust in Integrated Circuits Program [Adee 2008], which applies reverse engineering to circuits to detect malicious circuity intentionally added to test chips by a "red team." A very small amount of extra logic can give an attacker full control of a system.

We are not attempting to solve the trusted foundry problem. Nor are we trying to solve the problem of subverted design tools, which is another very difficult problem in which the attacker has a huge advantage. The subversion of design tools could easily result in malicious hardware being loaded onto the device, and security is not yet a primary goal of the major design tool manufacturers.

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[2]), can be safely combined with untrusted cores.

Our method works by putting constraints on the layout function of the design tools so that the design obeys a separation policy. In order for our method to be effective, the tools must be used properly, and the policy must be correct. Ensuring the correctness of the policy can be achieved using formal methods.

3.5 Motivating Examples

Encryption. We have been motivating our discussion so far with the example of a networked cryptographic authentication system to send/receive encrypted data from the Internet using local authentication [Huffmire et al. 2008]. In general, cryptographic systems contain sensitive data (both the encrypted data and the cryptographic keys) that must be protected. The goal of our methods is to ensure that this sensitive data does not fall into the wrong hands. In our example, this means that it should not be possible to read any unauthorized key or data either from external memory or directly from the encryption core itself.

Avionics. Isolation has long been a fundamental requirement in the design of avionics. Consider the example of avionics in military aircraft [Weissman 2003] in which sensitive targeting data is processed on the same device as less sensitive maintenance data. In such military hardware systems, certain

[2]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 article.

processing components are "cleared" for different levels of data, and proper separation of these components is needed to protect sensitive data. Isolation in avionics is also required because an aircraft must continue to fly even if one component malfunctions. To achieve fault containment, avionics are designed with a federated architecture [Rushby 1999]. However, since airplane designs must minimize weight, power, cooling, and maintenance costs, it is impractical to have a separate device for every function. Therefore, avionics was the impetus for the development of the first separation kernels [Rushby 1984]. A separation kernel isolates multiple applications executing on a processor and facilitates the controlled sharing of data among processes.

Video surveillance. 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 [Niu et al. 2004; Jain et al. 2006]. For example, IBM's PeopleVision project has been developing such a video surveillance system [Senior et al. 2003] that protects the privacy of individuals by blurring their faces depending on the credentials of the viewer (e.g., security guards versus maintenance technicians). FPGAs are a natural choice for any streaming application because they 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. Then, in Section 7, we show how these low-level protection mechanisms can be used in the implementation of a higher-level memory protection primitive.

4. PHYSICAL ISOLATION WITH MOATS

As discussed in Section 3, 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. Without some assurances of isolation, it is very difficult to prevent a connection between two cores from being established.

In general, physical synthesis tools use performance as an objective function in their optimization strategy, which can result in the logical elements and the

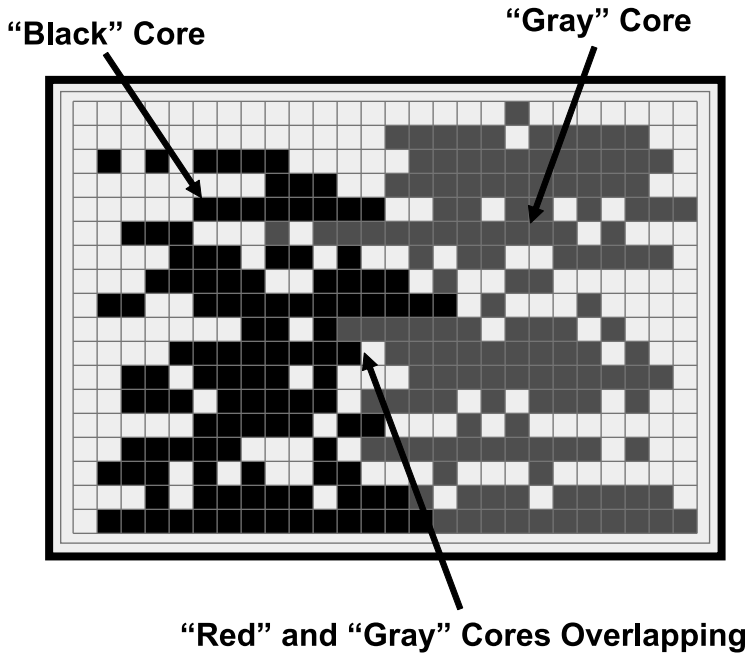

Fig. 2. A simple two-core system mapped onto a small FPGA. Normally, the design tools will place the cores in a manner that optimizes performance, which often leads to the intertwined situation shown here. Our method constrains the design tools to spatially separate the cores.

interconnections of two cores becoming intertwined. Figure 2 makes this problem more clear. The left-hand of Figure 2 shows the placement of a design with two small cores (soft processors) mapped onto an FPGA. The two processors overlap significantly in several areas of the chip. The difficulty of this problem is made more clear by the zoom-in on the right of Figure 2. The zoom-in shows a single switchbox, the associated LUTs (to the right of the switchbox), and all the wires that cross through that one small portion of the chip. The largest current FPGAs contain on the order of 20,000 to 30,000 switchboxes.

Isolation is required in order to protect the confidentiality and integrity of a core's data, and it helps to prevent interference with a core's functionality. In the following, we describe a design methodology for ensuring isolation of an IP core on a FPGA. We utilize a static check of the bitstream to verify that the cores are isolated.

4.1 Building Moats

Moats are a novel method of enhancing the security of FPGA systems, providing physical isolation of cores while requiring only small changes to the design tools. We propose two approaches to building moats. For both methods, we require that the cores be contained within distinctly rectangular regions. The ability to restrict cores to a rectangular area is a feature in today's FPGA tool chains, such as the Xilinx PlanAhead tool.

The first approach, which we call the *gap* method, involves surrounding each core with a "dead" area (i.e., a moat) while restricting the design to only use routing segments that are smaller than the length of the moat. We show that this approach ensures isolation of the cores. With the gap method, we are actually changing the architecture of the FPGA fabric, eliminating some kinds of routing segments.

In the second approach, which we call the *inspection* method, we show how to reduce or eliminate the moat by performing smart checking of the routing segments near the border of the core. Unlike the gap method, there is no need to change the design of the FPGA fabric. With the inspection method, we can utilize segments of any size except for those segments that lie close to the border of the isolated core. We describe the type of static analysis that is required to check the switchboxes near the border to determine if any connections violate the isolation of the core. For both the gap and inspection methods, a core can only communicate with the outside world via a precisely defined path called a "drawbridge," which we explain in Section 5. In this section, we provide analysis of the gap and inspection methods to determine which technique is better.

We note that the gap method is a physical/spatial isolation technique, while the inspection method is a logical isolation technique. We developed the inspection method after we developed the gap method.[3] Evolution from physical to logical isolation is a common design pattern in software. For example, separation kernels evolved out of prior isolation techniques that used separate computers [Rushby 1981].

4.1.1 *The Gap Method.* The gap method provides isolation by surrounding each core of interest with a moat. In other words, we disable the switchboxes outside the border of the core.

Modern FPGA architectures use staggered, multiple-track routing 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. A length 6 segment will span 6 CLBs, allowing a more direct connection by skipping unnecessary switchboxes along the routing path. Moreover, many FPGA architectures provide "longline" segments which span the complete row or column of the CLB array.

Figure 3 illustrates the gap technique of constructing moats. If we allow the design tool to make use of segment lengths of one and two, the moat must have a width of at least two to ensure isolation (otherwise signals could hop the moats because they would not *require* a switchbox in the moat). The following two properties are sufficient to statically check that a moat is sound:

(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.

[3]Our paper in IEEE Symposium on Security and Privacy only discusses the gap method [Huffmire et al. 2007].

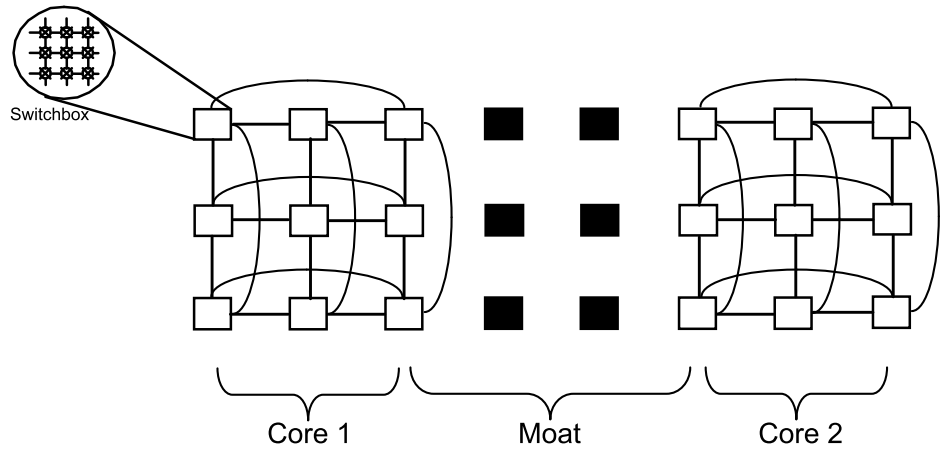

Fig. 3. Gap method: The gap method of building moats uses "dead" areas between the cores for isolation. In this example, routing segments can either span one or two switchboxes, which requires the moat to have a width of two. Since the delay of a connection depends on the number of switchboxes it must pass through, restricting the length of segments reduces performance, but the moats can be smaller. Allowing longer segments improves performance, but the moats must be wider, which wastes more area.

Both of these properties are easy to verify on an FPGA. We can tell if a switchbox is part of a moat by checking that it is completely dead, that is, all the routing transistors are programmed to be disconnected. We can check the second property by examining all the switches that connect to segments with length greater than w and verifying that they are turned off. This requires knowledge about the bitstream, for example, we must know the location of the configuration bit for each switch. All of the knowledge that we require is available using the JBits API [Guccione et al. 1999]. Although Xilinx has replaced JBits with XDL for its latest models, our ideas are easy to extend, and we argue that it should not be difficult to target them to any FPGA given sufficient knowledge about the bitstream. XDL, a text format, is similar to JBits but without the GUI interface. The XDL file contains all of the programmable elements, including vertical carry chains and horizontal sum-of-products.

4.1.2 *The Inspection Method.* By employing a smarter static checking scheme, we can reduce the size of the moat or even eliminate it altogether. The key insight behind this technique relies on the fact that we can utilize a segment that is longer than the moat width, as long as that segment is located far enough away from the border of the IP core. For example, assume that we have a design with moat width of two, and the core's dimension is 20×20. The gap technique requires that we not use any segment longer than two. However, there is no reason why the core could not use a hex segment that resides in the middle of the core for routing internal signals, for example, a segment from $(10, 10)$ to $(16, 10)$.

With the inspection method, we can have smaller moats than with the gap method, and we can even have no gap at all, which we call a *seamless* moat. Without a sufficiently large gap, we must use static analysis to ensure that

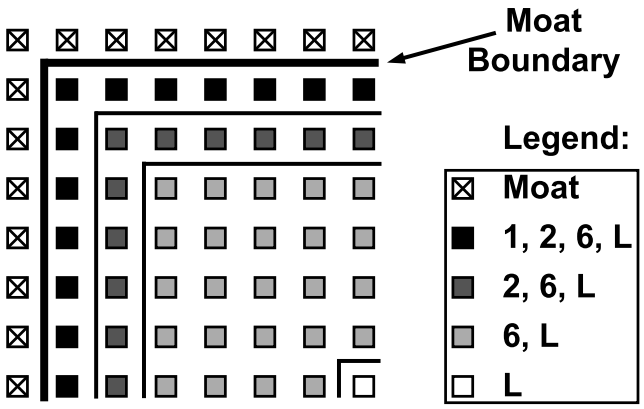

Fig. 4. Inspection method: The inspection method of building moats does not require a large gap (or any gap at all, which we call a "seamless" moat). Instead, the design tools check along the boundary for illegal connections. Since a seamless moat has no gap between the cores, all connections must be traced *near the border*: single lines (1), double lines (2), hex lines (6), and long lines (L). The "deeper" we go into the center of the core, the fewer connections that must be checked.

illegal connections do not cross the boundary. Consider a seamless moat: a connection that is one CLB away from the border of a core could connect to a neighboring core through the use of a double, hex, or long line. A single line, however, could not reach a neighboring core unless it was on the border of the core. Therefore, only a *subset* of the connections within a core must be examined, a fact that we can exploit to limit the amount of static checking.

Figure 4 shows the subset of connections that must be checked in order to verify the design for a seamless moat. The "deeper" we go into the center of the core, the fewer connections that must be checked. For example, even with seamless moats, we don't have to worry about checking single lines once we are at least one CLB away from the boundary. We don't have to check double lines once we are at least two CLBs away from the boundary. We don't have to check hex lines once we are at least six CLBs away from the boundary. Long lines must always be checked, as they span the full height or width of the device.

The smaller the width of the moat, the more checking we must do. The depth (in CLBs) at which a connection must be searched is a function of the moat width M and is defined as

$$D(L, M) = \begin{cases} 0 & \text{if L} < \text{M}, \\ L - M & \text{if L} \geq \text{M}, \end{cases} \tag{1}$$

where D is the search depth, L is the length of the connection (1, 2, or 6), and M is the size of the moat. Using this equation, it is evident that a moat width of one eliminates the need to check for single line connections inside of cores. Because there is a minimum of one CLB separating all cores in the design, any connection of length one cannot span from one core to another. Even if the connection is on the border of the core it will have to use some other connection inside the moat to go any further, so this connection will be found during the verification of the moat and does not need to be checked again. Similarly, with

a moat width of two, checking of double lines is eliminated, and with a moat width of six, checking of hex lines is eliminated.

4.2 Quantitative Analysis of the Gap Method

To understand the trade-offs of these two methods of constructing moats, we first compare their costs quantitatively. For moats constructed using the gap technique, we analyze the trade-off between circuit performance and "dead" area. The gap technique involves restricting the use of longer segments, while the inspection technique allows all segment lengths but performs static checking instead. Since restricting the length of segments hurts performance, we will show that smaller moats will result in worse performance than larger moats using the gap technique, but smaller moats will result in better performance using the inspection technique. For moats constructed using the inspection technique, we also consider the cost of checking for illegal connections near the boundary.

On an FPGA, the *delay* of a route largely depends on the number of switchboxes that it passes through rather than the total distance that it spans. Although large moats consume a great deal of chip area (because they reserve switchboxes 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 restricting the routing architecture to use only small routing 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.

4.2.1 *The Effect of Constrained Routing Architectures.* 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 six moat for protecting a core. By eliminating long lines and hex lines, we only need a moat of size two, and so on.

In order to study the impact of the gap method, we compare the quality of the MCNC benchmarks [Lisanke 1991] using different routing architectures. We do this to model the effect of eliminating routing segments longer than the moat width. We will use this to calculate the overall cost of employing the gap method over differing moat widths. We use the Versatile Placement and Routing (VPR) toolkit developed by the University of Toronto for our experiments. VPR provides mechanisms for examining trade-offs between different FPGA architectures [Betz et al. 1999]. 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 result.

The effect of the routing architecture's constraints on performance and area can vary across different cores. Therefore, we route the 20 biggest applications from the MCNC benchmark set [Lisanke 1991] 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 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 tristate buffered switches with Wilton connection patterns [Wilton 1997] within the switchbox. A Wilton switchbox 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 hardware area cost and critical path performance for all the benchmarks over four configurations. We can see that the existence of longlines has little impact on the final quality of the mapped circuits (compare Baseline and 1,2,6). However, significant degradation occurs when we eliminate segments of length two and six. This is caused by the increased demand for switchboxes, 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 through 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.

4.2.2 *Overall Area Impact.* While the results from Figures 5 and 6 show that there is some area impact from constraining the routing, there is 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 call this number the effective utilization, explained in Figure 7. Specifically, the effective utilization is as follows.

$$U_{eff} = \frac{A_{AllRoutes}}{A_{RestrictedRoutes} + A_{Moats}} \tag{2}$$

Figure 8 presents the trade-offs between the moat size, the number of isolated cores on the FPGA, and the utilization of the FPGA. We used a large FPGA for these calculations; one with 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

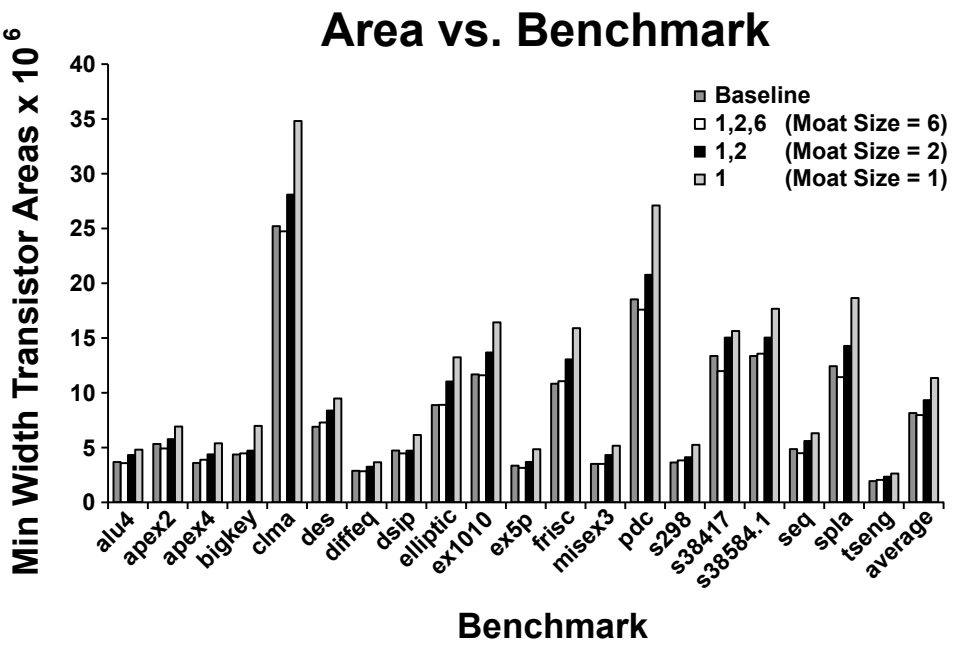

Fig. 5. Comparison of area for different configurations of routing segments. The baseline system has segments with length 1, 2, 6, and longline. The distribution is close to that of Virtex II: 8% (1), 20% (2), 60% (6), and 12% (longline). Other configurations are created by eliminating one or more classes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

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 nonmoat 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 8 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. While the number of cores is heavily dependent on the application, and the trade-off presented here is somewhat specific to our particular platform, 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 $O(\sqrt{n})$.

4.3 Quantitative Analysis of the Inspection Method

In this section, we analyze the inspection method. Unlike the gap method, the inspection method can use all segment lengths for routing. This requires a more complex method of static checking to ensure isolation. We will show that smaller moats achieve better performance using the inspection technique. We

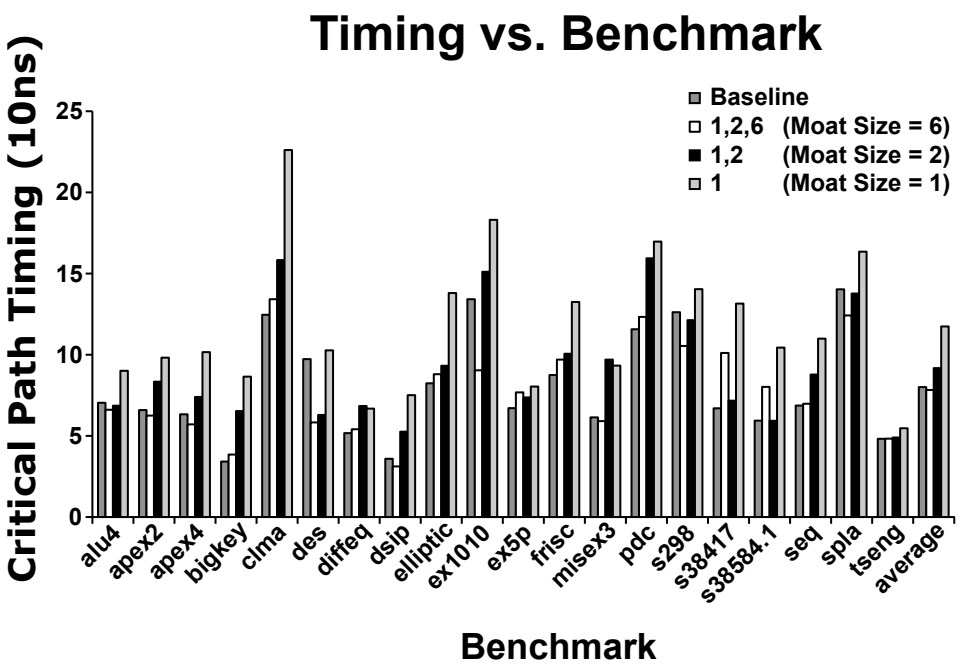

Fig. 6. Comparison of critical path timing for different configurations of routing segments. Unlike Figure 8, the graphs in Figures 5 and 6 do not include the overhead of the moat itself.

also analyze the cost of performing the checking. This cost is a one-time-only cost, since checking is performed at design time. The exception to this is the case of systems that employ partial reconfiguration, in which some checking must be performed at runtime.

The numbers of each type of routing segment connecting to a switchbox for a Xilinx Virtex-II device are as follows: 16 single, 80 double, 240 hex, and 48 longlines [Xilinx, Inc. 2005]. Since hex lines account for a majority of the connections, if we can avoid checking hex lines then there will be a significant savings in the verification effort. However, this results in a larger moat area to check, making less area available to use for logic on the chip. The timing of the design may also be adversely affected since the modules will be spaced further apart, thus giving a longer delay for signals routed between them.

Using Eq. (1), along with the number of each routing segment discussed before, we can estimate the number of connections that must be checked. The graph in Figure 9 shows the results. Since the moat area is significantly smaller than the area used by the cores, a larger moat size requires less checking. Even though you have a larger moat area in which all connections must be checked, the savings realized in checking the cores outweighs the cost of extra moat area to check. The benefit of moats is even greater when you have a larger number of cores. Having a larger number of cores will result in more overlap of the moat space between cores. The extra overhead of having to search all the connections in the moat is effectively divided among the cores that border the

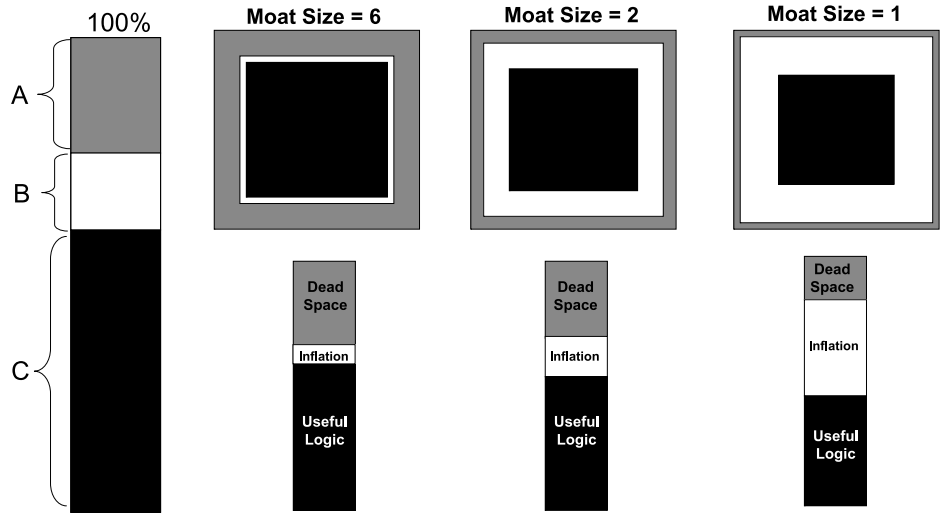

Fig. 7. Our effective utilization metric $U_{eff} = \frac{C}{A+B+C}$, where A is the dead area for the moats (which depends on the number of cores), B is the inflation due to restricted routing (on the order of 10%), and C is the useful logic with no inflation (unrestricted routing). This figure is not drawn to scale.

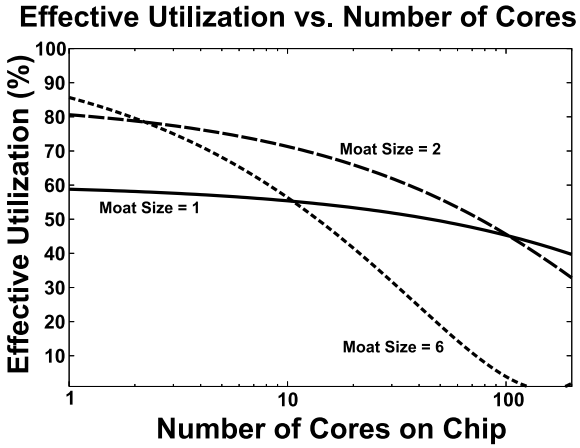

Fig. 8. The trade-off between the number of cores, the size of the moat, and the utilization of the FPGA. An increasing number of cores results in larger total moat area, which reduces the overall utilization of the FPGA. Larger moat sizes also will use more area, resulting in lower utilization.

moats. The area of overlap between cores increases as you have more isolation domains on the chip, resulting in greater savings in verification effort.

If you ignore the potential savings from using moats of size one or larger, it may seem to make more sense to use seamless moats, which would allow for the maximum possible utilization of the FPGA area. However, there are several arguments for using a moat of size one or greater. For example, the moat area can act as a communication channel for the routing of drawbridges (signals between cores and I/O pins).

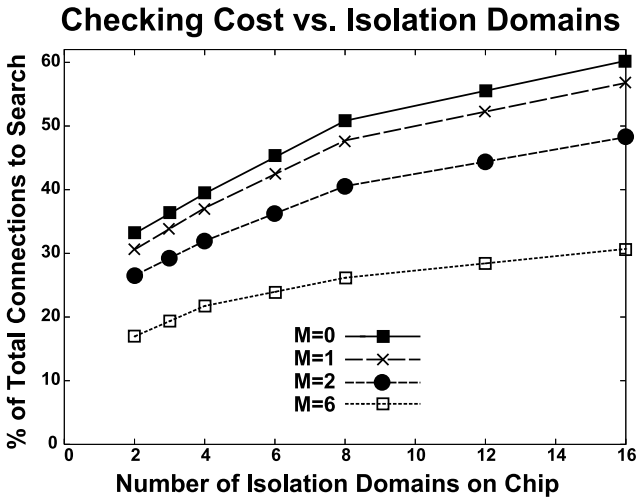

Fig. 9. This graph shows how many connections we must check in the switchboxes to verify our design based on the number of isolation domains on the chip. It shows this for moat sizes of m = 0, 1, 2, and 6.

4.3.1 *Benchmarks.* To evaluate the performance of the inspection method, we partitioned several different systems with moats and drawbridges. All designs were placed and routed on a Xilinx Virtex-4 XC4VSX35-FF668-10C device. The designs were synthesized using multipass place and route with the effort level set to high for ten different runs.

The first test system is a distributed Multiple Input Multiple Output (MIMO) transceiver [Mirzaei et al. 2008]. It is designed to wirelessly receive data from a set of distributed sensor nodes over multiple channels. The design was partitioned into four different cores: digital up converter, digital down converter, channel tracker, and timing and frequency estimation core.

Next was the networked cryptographic authentication system [Huffmire et al. 2008]. As previously discussed, this design contains two μBlaze processors connected to a shared onboard peripheral bus with an integrated reference monitor to regulate access to the shared memory and peripherals. The bus had the following slave devices: DDR SRAM, RS232, Ethernet, and AES encryption/decryption core. The system is divided into seven different cores: the AES, RS232, DDR SRAM, Ethernet, OPB, $Processor_0$, and $Processor_1$.

The third system was a hardware JPEG encoder. The encoder was partitioned into 5 different cores: DCT, ITU 656 data stream decoder, quantization and rounding, run length encoder, and discrete cosine transform.

4.3.2 *Experimental Results.* The first question that we study is: "How much area do the cores require?" On one hand, we could try to minimize the rectangular area where the core resides. This will save area; however, it also can severely restrict the physical synthesis tools. On the other hand, giving the core too much area would be wasteful. We conducted a series of tests to understand this trade-off. First, we study the relationship between utilization and clock frequency. To study this effect, we took two designs, each with only one

Table I. The Relationship Between How Much Area a Core is Given and the Performance of the Core

Slices utilization (%)	Maximum Frequency (MHz)	
	μBlaze	AES
50	85.21	76.49
60	83.55	69.93
72	79.11	75.71
80	81.68	70.85
90	75.55	68.24
96	76.83	69.58

Slice utilization is the total number of slices in the area where the core is placed divided by the number of slices that the core actually uses. This is done for two cores, an AES encryption core, and a μBlaze processor core.

core, and constrained the core to different size areas. The logic utilization (the amount of slices required by the core over the total number of slices available) was compared with the maximum clock frequency. As shown in Table I, even at 96% slice utilization, a performance penalty of less than 10% was observed. With less area, the physical synthesis tools are more constrained with respect to where to place the logic. Since the tools cannot place the logic for optimal performance, both design size and performance are impacted. In the following experiments, we make sure to give each core enough area to limit the impact on performance.

The performance tests for the inspection method analyzed the effect of moat size on performance. We used several moat sizes for the four different designs described in the previous section. To minimize the effect discussed earlier, the cores were constrained to as large an area as possible, and when possible cores were placed as close as possible to the I/O pins that they used. The results of these tests can be seen in Table II. The results show that the effect of moat size on the design is minimal. In the worst case we saw only a 2.61% decrease in performance, and in the best case there was a 1.05% increase in performance. Just as with regular floorplanning, if the different IP cores are properly placed, the performance can be improved. As expected, larger moat sizes hurt the performance because of the longer delay caused by increased wire length between the cores.

From our performance tests, it is evident that there is a direct correlation between moat size and performance when using the inspection method. The larger the moat size, the greater the minimum clock period. Larger moat sizes also require more area overhead for the moat. However, they provide a significant reduction in the complexity of verification, which is especially important if we want to perform runtime checking. Furthermore, the effect on performance was minimal, with a maximum average increase of 2.61% from the baseline case of no moat (unconstrained placement).

The impact of the moat area, however, was much more significant. In the cryptographic design with seven cores, the moat area took up 29% of the chip area for a moat size of six. Clearly, unless ample extra space was available on chip, a moat size of six would not be feasible. The space-saving properties of

Table II. The Relationship between Moat Size and Design Performance (T=clock period) for Several Different Systems

Moat Size	$\overline{T}$ (Avg) (ns)	% Δ	σ	T (Min) (ns)	T (Max) (ns)	# CLBs	% Total Connections to check
MIMO							
No Moat	6.8791	N/A	0.3233	6.5	7.419	2148	N/A
0	6.8068	-1.05%	0.1546	6.532	7.025	2148	60.8%
1	6.8356	-0.63%	0.2986	6.456	7.32	2276	54.3%
2	6.9113	0.47%	0.1846	6.656	7.232	2400	48.2%
6	6.9558	1.11%	0.2258	6.61	7.343	2880	30.1%
Crypto							
No Moat	13.6563	N/A	0.5605	12.838	14.533	2115	N/A
0	13.6563	0.00%	0.5605	12.838	14.533	2115	72.0%
1	13.845	1.38%	0.4349	13.273	14.586	2332	65.8%
2	13.6913	0.26%	0.496	12.99	14.258	2501	58.1%
6	13.9038	1.81%	0.4247	13.314	14.49	3213	37.5%
JPEG							
No Moat	7.2	N/A	0.0868	7.021	7.342	1074	N/A
0	7.2473	0.66%	0.1103	7.081	7.446	1074	72.8%
1	7.2894	1.24%	0.1605	7.066	7.578	1212	67.2%
2	7.3861	2.58%	0.2546	7.049	7.778	1377	61.9%
6	7.3876	2.61%	0.1892	7.063	7.701	1851	46.4%

The clock period T was averaged over 10 different place-and-route runs for each design, each run using a different seed. The percent change Δ in average clock period from the baseline of no moat is shown for each design.

seamless moats makes them highly advantageous. However, this comes at the expense of more costly verification. With the exception of systems that must perform verification at runtime (e.g., systems that employ partial reconfiguration), seamless moats are optimal, as the results in Table II show. We can also see that the performance impact of moats constructed using the inspection method is extremely small, and in some designs there is even a slight increase in performance.

5. DRAWBRIDGES: INTERCONNECT INTERFACE CONFORMANCE WITH TRACING

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 a core. The entry or communication is only allowed through a prespecified interface called a "drawbridge." We must know in advance how the cores communicate (i.e., all possible connections between cores) and the location of these 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 that (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. The major 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 final stage of design allows us to catch illegal connections that could have originated from any stage in the design process, including subversion by the design tools.

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.

We perform tracing at the bitstream level, using the JBits API [Guccione et al. 1999] 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 only to valid cores. 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.

The route tracing tool takes two inputs: a bitstream file and a file specifying all modules along with their boundaries and a list of connections. Connections are specified in terms of a source (pin or module) and destination (pin or module). To follow is an example input file.

```
# denotes a comment
# first declare the device type
#D device
D XC2V6000 FF1517

#N modules pins connections
N 4 5 12

#M  modulename xmin xmax
# ymin ymax
M MB1 11 35 57 80
M MB2 11 35 13 35
M MB3 54 78 57 80
M MB4 54 78 13 35
```

```
#P  pinname in/out
P B25 rst #Reset
P C36 in #rs_232_rx_pin
P J30 out #rs_232_tx_pin
P C8 in #rs_232_rx2_pin
P C9 out #rs_232_tx2_pin

#C source destination width
C B25 MB1 1
C C36 MB1 1
C MB1 J30 1
C B25 MB2 1
C MB1 MB2 32
C MB2 MB1 32
C B25 MB3 1
C MB3 C9 1
C C8 MB3 1
C B25 MB4 1
C MB4 MB3 32
C MB3 MB4 32
```

The tracing tool analyzes the bitstream to ensure that all connections are valid. Connections to external pins are the simplest because the program simply starts at the pin and searches the path until it arrives at the destination module. On the other hand, checking connections between modules is more complicated because placement of the module during floor planning does not place the gates and connections; it merely constrains the module to a particular region of the chip. Since we don't have precise knowledge of where the design tools placed the connections, the program must search the entire area of the module for them. Searching the entire area is necessary because hex lines and longlines can begin inside a module and end outside of it.

The tracing algorithm starts with a list of input and output pins (some pins may be able to do both) that can enter or leave a CLB. After performing a trace on all the input pins, it next traces all outgoing connections from all the CLBs in the modules. Finally, it performs a reverse trace on all outgoing connections from the modules (although if the design is correct there will be none, since the connections will have been found in the previous steps). The route tracing algorithm is a simple breadth-first search with a few modifications: it maintains a list of every pin it has visited to prevent searching the same path twice. The search is terminated once another module is reached. The following pseudocode describes the tracing process.

```
RouteTree trace(pin, module) {
   add pin to routeTree
   for all sinks of wire this pin is on {
      if sink is connected to pin
         if sink has already been searched
            return
```

Table III. Performance Results of Tracing Program

Design	Area(CLBs)	#Connections	#Gates	#LUTs	Time(s)
1 μBlaze	396	3	332489	1592	5.66
2 μBlaze	1050	68	663128	3153	9.7
3 μBlaze	1800	71	987631	4678	13.88
4 μBlaze	2400	135	1319492	6349	17.03
Encoder	4	12	54	9	2.26
Crypto	4200	140	780577	19150	71.53
Game	3697	92	910242	6129	22.82

```
            if sink is in another module
               check if connection is valid
               return
            add sink to list of searched pins
            trace(sink, module)
      }
}
```

The tracing program outputs all the connections it finds, and it can optionally display the route tree showing the entire path of a connection. When finished, it outputs whether or not the design was successfully verified.

```
.
.
.
Found Valid connection:MB1 to MB2
CLB.S6BEG5[57][33]
.  [CLB.S6END5[51][33]]
.  .  CLB.S6BEG5[51][33]
.  .  .  [CLB.S6END5[45][33]]
.  .  .  .  CLB.S6BEG3[45][33]
.  .  .  .  .  [CLB.S6END3[39][33]]
.  .  .  .  .  .  CLB.S2BEG3[39][33]
.  .  .  .  .  .  .  [CLB.S2END3[37][33]]
.  .  .  .  .  .  .  .  CLB.S2BEG1[37][33]
.  .  .  .  .  .  .  .  .  [CLB.S2END_S1[34][33]]
Found Valid connection:MB3 to MB4
CLB.OMUX0[58][58]
.  CLB.LV12[58][58]
.  .  [CLB.LV18[28][58]]
Found Valid connection:MB3 to C9
.
.
.
Design Successfully verified!
```

We evaluated the performance of the tracing program on several test designs to determine how different parameters affect the runtime, using a commodity Dell Inspiron notebook computer running Windows XP. Table III shows the

results. Since these designs are not using partial reconfiguration, the analysis can be performed offline during the design phase. Even if they use partial reconfiguration, the reconfigurable modules can always be checked at compile time.

If a design uses partial reconfiguration, there is an opportunity to realize much greater efficiency. We found that it is possible to trace routes in the reconfigurable portion after the rest of the design has been verified. If a design uses partial reconfiguration, we only have to store two pins for each connection that passes through the reconfigurable area. Connections that only enter but do not leave the area only require us to store one pin and the direction (in or out). In this case, we only have to search the pins that enter or leave the module. This greatly reduces our search area, although it may add some overhead to the initial tracing. We found that for one of the static designs (Game), which had seventy connections passing through the reconfigurable area, the runtime was 147 times faster (22.8161s versus 0.15485s), and the increase in the overhead of the initial tracing of the entire chip was very small (0.0155s).

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.

5.1 Efficient Communication under the Drawbridge Model

In modern reconfigurable systems, busses are a common method of communication between cores. 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. In the following, we propose a secure bus architecture for FPGAs.

To address this problem of covert channels and bus snooping, we have developed a *shared memory bus* with 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. 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 [Hu 1991]. This covert channel could be used to send information from a module with a high security label to a module with lower security label (write-down), which would violate the Bell-LaPadula model [Bell and LaPadula 1973] 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 allotted time slice, it has no way of manipulating the bus to send or infer information. One module cannot tell if any of the other modules is 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 completely eliminate this covert channel is to give each module a dedicated connection to all other modules. Requiring a dedicated direct connection between all modules that need to communicate would be inefficient and costly. Dedicated channels would require a worst case of $O(n^2)$ 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 one classified module to another classified module and 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 all the modules on the bus. The modules can only use the bus through this arbiter, which only allows each module to read during its designated time slice. If we want to connect the bus to a memory, then a memory monitor is also required; for this work we assume that such a configuration can be implemented on the FPGA using the results of our previous work [Huffmire et al. 2006].

5.2 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 scheme (Figure 10). A module is only allowed to read or write during its alloted 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 11). 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 10), 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 11) intuitively should be more scalable and efficient since it uses a bus instead of individual connections for

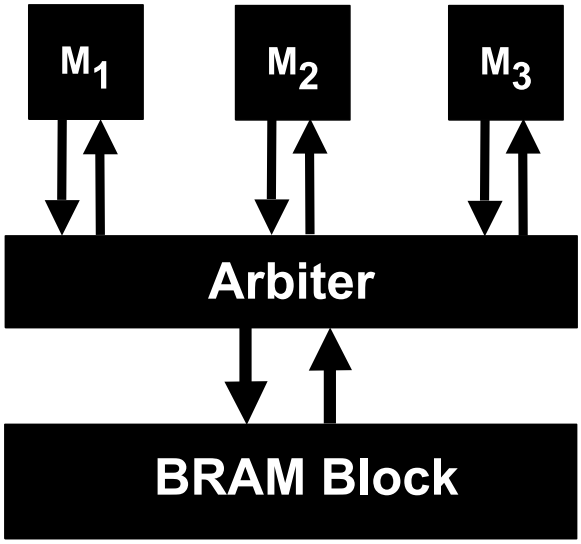

Fig. 10. Architecture alternative 1. There is a single arbiter, and each module has a dedicated connection to the arbiter.

each module. However, the arbiters have to coordinate, the memory monitor has to be split, and each arbiter needs 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. The results of the analysis of the two architectures, which can be seen in Table IV, were not what we first expected. During synthesis of the second architecture, the synthesis tool converted the tristate buffers 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 performance difference between the two was almost negligible, we recommend using the first architecture. 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.

6. EFFECTIVE SCRUBBING AND REUSE OF RECONFIGURABLE HARDWARE

Modern FPGA architectures have begun to incorporate the ability to perform partial reconfiguration of the logic and routing resources. This makes it possible to to selectively change part of the FPGA's configuration, one column at a time [Lysaght and Stockwood 1996]. Partial reconfiguration is performed through a special interface on the FPGA which can read and write the bitstream while the FPGA is running. This interface must therefore be part of the trusted computing base of the system.

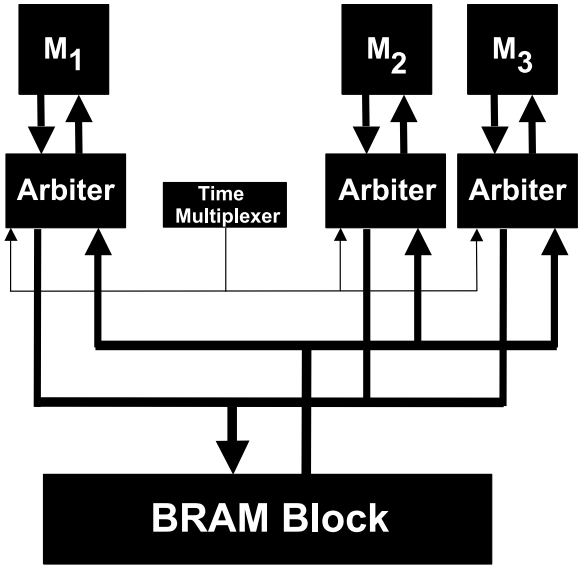

Fig. 11. Architecture alternative 2. Each module has its own arbiter that prevents bus snooping and a central time multiplexer that connects to all the arbiters.

Table IV. Comparison of Communication Architectures

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

Partial reconfiguration improves the flexibility of a system by making it possible to swap cores. For example, Lysaght and Levi [2004] have devised a dynamically reconfigurable crossbar switch. By using dynamic reconfiguration, their 928x928 crossbar uses 4,836 CLBs compared to the 53,824 CLBs required without reconfiguration. In another example, Baker and Prasanna [2004, 2005] have developed an intrusion detection system based on reconfigurable hardware that dynamically swaps the detection cores. Since the space of intrusion detection rule sets is infinite, the space of detection cores is also infinite. In our earlier work [Huffmire et al. 2006], we developed a memory protection scheme for reconfigurable hardware in which a reconfigurable reference monitor enforces a policy that specifies the legal sharing of memory. Partial reconfiguration could allow the system to change the policy being enforced by swapping in a different reference monitor, assuming that you could overcome the very difficult challenge of ensuring the security and correctness of the handover. Since the space of possible policies is infinite, the space of possible reference monitors is also infinite.

To extend our model of moats to this more dynamic case, we need to make sure that nothing remains of the prior core's logic when it is replaced with

Table V. Reconfiguration Time

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

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 V 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 μs to reconfigure, or equivalently, erase. Therefore, reconfiguring (erasing) the entire device 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.

After the sanitization process is complete, it is a good idea to apply integrity checks to ensure the correctness of the new design. A straightforward and efficient way to do this is to compute a checksum of the new design and compare against a list of known good checksums [Glas et al. 2008].

To perform scrubbing, all you have to do is perform a reset per flip-flop. Clearly, the scrubbing controller needs to be trusted since its function is highly sensitive. For example, a malicious scrubbing module could change the reconfigurable logic or fail to reset the logic. Each flip-flop should have a reset, and if you need to, you can put your data back in following the reset. In order to scrub other kinds of state, such as I/O registers, you could use a scrubbing controller.

Since partial reconfiguration increases design complexity, practitioners often avoid it, opting instead to wait a couple years for Moore's Law to double the number of available logic blocks. In addition to increasing design complexity, the security analysis is also more complicated with partial reconfiguration. Another limitation is that enabling partial reconfiguration disables bitstream decryption mechanisms. If this were not done, any encrypted bitstream could

be obtained via the ICAP interface. To overcome this problem it is possible to use a reconfigurable crypto core together with the ICAP interface to protect the bitstream [Harper et al. 2003]. Many of the features of PlanAhead were originally developed to enable partial reconfiguration, and our moats and drawbridges work exploits some of them. However, PlanAhead does not perform checking similar to our tracing algorithm.

7. APPLICATION: MEMORY POLICY ENFORCEMENT

Now that we have described a set of isolation and separation primitives, we provide an example of utilizing these primitives to perform memory protection, an even higher-level primitive. Saltzer and Schroeder [1974] 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." 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 [Huffmire et al. 2006]. 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. We can use our isolation primitives to ensure that this is the case. For example, we can surround the reference monitor with a moat, specify the allowable connections using drawbridges, then use interconnect tracing to ensure that the memory I/O blocks are only connected to the reference monitor. Without these primitives, it is easy for a core to bypass the reference monitor and access memory directly.

Saltzer and Schroeder [1974] describe how protection mechanisms can protect their own implementations in addition to protecting users from each other. 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 primitives also make 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. Using moats and drawbridge makes it impossible to connect two cores directly without crossing the moat.

As we described earlier, a reference monitor approach to memory protection requires that every memory access pass 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 5.1, 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.

8. RELATED WORK

There has always been an important relationship between the hardware a system runs on and the security of that system, and reconfigurable systems are no different. In addition to the related work we have already mentioned, 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 relevant.

8.1 Reconfigurable Hardware Security

To provide memory protection on an FPGA, we proposed the use of a reconfigurable reference monitor that enforces the legal sharing of memory among cores [Huffmire et al. 2006]. 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 our work addressed the specifics of how to construct a memory access monitor efficiently in reconfigurable hardware, we did not address the problem of how to protect that monitor from routing interference, nor did we describe how to enforce that *all* memory accesses go through this monitor. This article directly supports our prior work by providing the fundamental primitives that are needed to implement memory protection on a reconfigurable device.

There is similar concurrent work by McLean and Moore [2007]. Though they do not provide extensive details, they appear to be using a similar technique to isolate regions of the chip by placing a buffer between them which they call a *fence*.

Trimberger [2007] explains how FPGAs address the trusted foundry problem. When an ASIC is manufactured, the sensitive design could be stolen by malicious employees of the foundry. For defense sensitive content, this issue concerns the national interest. Although FPGAs address this problem for the fabrication phase, the design could be stolen from the FPGA itself by circumventing bitstream protection mechanisms. ASICs are also susceptible to theft of the design after fabrication.

Since techniques that make ASIC circuits resistant to side-channel attacks do not necessarily translate to FPGAs, Yu and Schaumont [2007] have developed a routing technique to reduce the risk of side-channel attacks on FPGAs. Their technique involves creating a duplicate circuit that has symmetrical power consumption to cancel out the original circuit. Our work could be used in conjunction with this technique to provide a higher level of security.

Chien and Byun [1999] address the safety and protection concerns of enhancing a CMOS processor with reconfigurable logic. 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 [Lie et al. 2000] or virtual secure coprocessing [Lee et al. 2005].

Gogniat et al. [2006] 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. 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 their 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 [Bossuet et al. 2004; Kean 2001; 2002], fingerprinting [Lach et al. 1999a], and watermarking [Lach et al. 1999b]. However, establishing a root of trust on a fielded device is challenging because it requires a decryption key to be incorporated into the finished product. Some FPGAs can be remotely updated in the field, and industry has devised secure hardware update channels that use authentication mechanisms to prevent a subverted bitstream from being uploaded [Harper et al. 2003; Harper and Athanas 2004]. These techniques were developed to prevent an attacker from uploading a malicious design that causes unintended functionality. Even worse, the malicious design could physically destroy the FPGA by causing the device to short-circuit [Hadzic et al. 1999]. 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.

8.2 Covert Channels, Direct Channels, and Trap Doors

The work in Section 5.1 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 [Standaert et al. 2003; Percival 2005].

We are not the first to propose security enhancements for SoCs employing a bus communication architecture. Coburn et al. [2005] propose a Security Enforcement Module (SEM) that provides stateful policy enforcement and resource arbitration, including memory protection. However, they do not address the problem of covert timing channels resulting from bus contention.

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 [Kemmerer 1983]. 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 [Saputra et al. 2003]) or closing the channel. Unfortunately, an adversary can extract a signal from the noise, given sufficient resources [Millen 1987].

Of course, our technique is primarily about restricting the opportunity for direct channels and trap doors [Thompson 1984]. Our memory protection scheme is an example of this. 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.

9. CONCLUSION

The design of reconfigurable systems is a complex process, consisting of multiple software tool chains that may have different trust levels. Composing a trustworthy system from untrusted components remains an open challenge in computer security. The tools necessary for high-assurance FPGA system development do not yet exist. Our methods represent an initial attempt at verifying security properties of FPGA designs. In particular, we described a set of primitives that can be used to isolate trusted cores from the effects of the subversion or failure of other cores in the system.

We have presented a protection scheme called moats and drawbridges that provides separation of multiple cores on an FPGA. Moats isolate the cores, and drawbridges facilitate their controlled interaction. We have presented two alternative methods of implementing moats. The gap method uses physical/spatial isolation and restricts the use of longer routing segments, while the

inspection method uses logical isolation and static checking instead. We have shown that the inspection method results in better performance than the gap method. We have also described how moats and drawbridges can be used in conjunction with a reference monitor to provide memory protection.

There are many opportunities for future work. To provide for heterogeneous provisioning of bandwidth for our TDMA bus architecture, we plan to investigate the application of lattice scheduling [Hu 1992], in which a core can "donate" its unused time to a core with a higher security label in a round-robin fashion.

Some of our security techniques are applicable to other kinds of devices besides FPGAs, such as Chip Multi-Processors (CMPs). For example, we would like to extend our drawbridge model to support Networks-on-Chip (NoCs) to provide efficient and secure communication among many cores. There are several open research questions in the design of NoCs, and techniques for incorporating security into the design of NoCs by enhancing the network interface are starting to emerge [Diguet et al. 2007; Florin et al. 2007a; 2007b].

ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers for their comments.

REFERENCES

ADEE, S. 2008. The hunt for the kill switch. *IEEE Spectrum 45*, 5.

BAKER, Z. AND PRASANNA, V. 2004. Efficient architectures for intrusion detection. In *Proceedings of the 12th Annual International Conference on Field-Programmable Logic and its Applications (FPL'04)*.

BAKER, Z. AND PRASANNA, V. 2005. Computationally-Efficient engine for flexible intrusion detection. *IEEE Trans. VLSI*.

BELL, D. AND LAPADULA, L. 1973. *Secure Computer Systems: Mathematical Foundations and Model*. The MITRE Corporation, Bedford, MA.

BETZ, V., ROSE, J. S., AND MARQARDT, A. 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic, Boston, MA.

BONDALAPATI, K. AND PRASANNA, V. 2002. Reconfigurable computing systems. *Proc. IEEE 90*, 7, 1201–1217.

BOSSUET, L., GOGNIAT, G., AND BURLESON, W. 2004. Dynamically configurable security for SRAM FPGA bitstreams. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*.

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

COBURN, J., RAVI, S., RAGHUNATHAN, A., AND CHAKRADHAR, S. 2005. SECA: Security-Enhanced communication architecture. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'05)*.

COMPTON, K. AND HAUCK, S. 2002. Reconfigurable computing: A survey of systems and software. *ACM Comput. Surv. 34*, 2, 171–210.

DEHON, A. AND WAWRZYNEK, J. 1999. Reconfigurable computing: What, why, and implications for design automation. In *Proceedings of the Design Automation Conference*. 610–15.

DIGUET, J., EVAIN, S., VASLIN, R., GOGNIAT, G., AND JUIN, E. 2007. NOC-Centric security of reconfigurable SoC. In *Proceedings of the 1st International Symposium on Networks-on-Chip (NOCS'07)*.

FIORIN, L., PALERMO, G., LUKOVIC, S., AND SILVANO, C. 2007a. A data protection unit for NoC-based architectures. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07)*.

FIORIN, L., SILVANO, C., AND SAMI, M. 2007b. Security aspects in networks-on-chips: Overview and proposals for secure implementations. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods, and Tools (DSD'07)*.

GLAS, B., KLIMM, A., SANDER, O., MULLER-GLASER, K., AND BECKER, J. 2008. A system architecture for reconfigurable trusted platforms. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE)*.

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

GUCCIONE, S., LEVI, D., AND SUNDARARAJAN, P. 1999. JBits: Java-Based interface for reconfigurable computing. In *Proceedings of the 2nd Annual Conference on Military and Aerospace Applications of Programmable Logic Devices and Technologies (MAPLD)*.

HADZIC, I., UDANI, S., AND SMITH, J. 1999. FPGA viruses. In *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications (FPL'99)*.

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

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

HILL, T. 2006. AccelDSP synthesis tool floating-point to fixed-point conversion of MATLAB algorithms targeting FPGAs. http://www.digchip.com/application-notes/9/43439.php.

HU, W. 1992. Lattice scheduling and covert channels. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*.

HU, W.-M. 1991. Reducing timing channels with fuzzy time. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*.

HUFFMIRE, T., BROTHERTON, B., CALLEGARI, N., VALAMEHR, J., WHITE, J., KASTNER, R., AND SHERWOOD, T. 2008. Designing secure systems on reconfigurable hardware. *ACM Trans. Des. Autom. Electron. Syst. 13*, 3.

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

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

JAIN, A., KOPPEL, D., KALIGIAN, K., AND WANG, Y.-F. 2006. Using stationary-dynamic camera assemblies for wide-area video surveillance and selective attention. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

KASTNER, R., KAPLAN, A., AND SARRAFZADEH, M. 2004. *Synthesis Techniques and Optimizations for Reconfigurable Systems*. Kluwer Academic, Boston, MA.

KEAN, T. 2001. Secure configuration of field programmable gate arrays. In *Proceedings of the 11th International Conference on Field Programmable Logic and Applications (FPL'01)*.

KEAN, T. 2002. Cryptographic rights management of FPGA intellectual property cores. In *Proceedings of the 10th ACM International Symposium on Field-Programmable Gate Arrays (FPGA'02)*.

KEMMERER, R. 1983. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Trans. Comput. Syst. 1*, 3, 256–277.

KOCHER, P., LEE, R., MCGRAW, G., RAGHUNATHAN, A., AND RAVI, S. 2004. Security as a new dimension in embedded system design. In *Proceedings of the 41st Design Automation Conference (DAC'04)*.

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

LACH, J., MANGIONE-SMITH, W., AND POTKONJAK, M. 1999b. Robust FPGA intellectual property protection through multiple small watermarks. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation (DAC'99)*.

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

LEWIS, J. AND MARTIN, B. 2003. Cryptol: High assurance, retargetable crypto development and validation. In *Proceedings of the Military Communications Conference (MILCOM)*.

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

LISANKE, B. 1991. Logic synthesis and optimization benchmarks. Tech. rep., Microelectronics Center of North Carolina.

LYSAGHT, P. AND LEVI, D. 2004. Of gates and wires. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*.

LYSAGHT, P. AND STOCKWOOD, J. 1996. A simulation tool for dynamically reconfigurable field programmable gate arrays. *IEEE Trans. VLSI Syst. 4*, 3.

MANGIONE-SMITH, W., HUTCHINGS, B., ANDREWS, D., DEHON, A., EBELING, C., HARTENSTEIN, R., MENCER, O., MORRIS, J., PALEM, K., PRASANNA, V., AND SPAANENBURG, H. 1997. Seeking solutions in configurable computing. *Comput. 30*, 12, 38–43.

MCLEAN, M. AND MOORE, J. 2007. Securing FPGAs for red/black systems: FPGA-Based single chip cryptographic solution. *Military Embedded Systems Mag.*

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

MIRZAEI, S., KASTNER, R., IRTURK, A., WEALS, B., AND CAGLEY, R. 2008. Design space exploration of a cooperative MIMO receiver for reconfigurable architectures. In *Proceedings of the IEEE International Conference Application-Specific Systems, Architectures and Processors (ASAP)*.

NIU, W., LONG, J., HAN, D., AND WANG, Y.-F. 2004. Human activity detection and recognition for video surveillance. In *Proceedings of the IEEE Multimedia and Expo Conference*.

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

RUSHBY, J. 1981. Design and verification of secure systems. *ACM SIGOPS Oper. Syst. Rev. 15*, 5.

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

RUSHBY, J. 1999. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. NASA contractor rep. CR-1999-209347, NASA Langley Research Center.

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

SAPUTRA, H., VIJAYKRISHNAN, N., KANDEMIR, M., IRWIN, M., BROOKS, R., KIM, S., AND ZHANG, W. 2003. Masking the energy behavior of DES encryption. In *Proceedings of the IEEE Design Automation and Test in Europe (DATE'03)*.

SCHAUMONT, P., VERBAUWHEDE, I., KEUTZER, K., AND SARRAFZADEH, M. 2001. A quick safari through the reconfiguration jungle. In *Proceedings of the Design Automation Conference*. 172–177.

SENIOR, A., PANKANTI, S., HAMPAPUR, A., BROWN, L., TIAN, Y.-L., AND EKIN, A. 2003. Blinkering surveillance: Enabling video privacy through computer vision. Tech. rep. RC22886, IBM.

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

THE MATH WORKS INC. 2006. MATLAB User's Guide.

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

TRIMBERGER, S. 2007. Trusted design in FPGAs. In *Proceedings of the 44th Design Automation Conference*.

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

WILTON, S. 1997. Architectures and algorithms for field-programmable gate arrays with embedded memory. Ph.D. thesis, University of Toronto.

XILINX INC. 2005. Virtex-II Platform FPGAs: Complete Data Sheet. Xilinx Inc., San Jose, CA.

XILINX INC. 2006. Getting started with the embedded development kit (EDK). http://www.xilink.com/support/documentation/sw_manuals/edk821_getstarted.pdf.

YU, P. AND SCHAUMONT, P. 2007. Secure FPGA circuits using controlled placement and routing. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07)*.

Received April 2008; revised October 2008; accepted March 2009