Methods for Securing the Integrity of FPGA Configurations

James Braxton Webb

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Electrical Engineering

Dr. Mark T. Jones, Chair
Dr. Peter M. Athanas
Dr. Cameron D. Patterson

August 29, 2006

Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: FPGA, security, configuration, integrity, authentication, fault-injection, configurable, reconfigurable, partial, dynamic, embedded system

Copyright 2006 ©, James Braxton Webb

Methods for Securing the Integrity of FPGA Configurations

James Braxton Webb

(ABSTRACT)

As Field Programmable Gate Arrays (FPGAs) continue to become integral parts of embedded systems, it is imperative to consider their security. While much of the research in this field is oriented toward the protection of the intellectual property contained in the FPGA's configuration, the protection of the design's integrity from malicious attack against the configuration is critical to the operation of the system. Methods for attacking the configuration are semi-invasive attacks, such as fault injection, and data tampering of incoming partial bitstreams.

This thesis introduces methods for securing the integrity of an FPGA's configuration. The design and implementation is discussed for a system that consists of three parts. The first subsystem monitors the running configuration. The second subsystem authenticates partial bistreams that may be used for repairing the configuration from malicious alterations during run-time. The third subsystem indicates if the system itself succumbs to a malicious attack. The system is implemented on-chip, allowing the FPGA to effectively secure itself from attack.

Acknowledgements

The work presented in this thesis would not have been possible if it were not for the help, support, guidance, and friendship of a number of individuals. For those who are not specifically mentioned, I also thank you.

First, I would like to thank my advisor, Dr. Mark Jones, for his support and advice in the completion of this thesis and my graduate education. I would also like to thank Dr. Peter Athanas for his constant interest in my education as well as his guidance and encouragement on my work in the Configurable Computing Laboratory (CCM Lab) and on this thesis. My thanks also to Dr. Cameron Patterson for the many hours he spent sharing his vast wealth of knowledge with me and helping me complete this work.

I would like to thank the Harris Corporation for funding the project that supported this work.

My gratitude also to my friends in the CCM Lab who helped me in the completion of the work for this thesis: Stephen Craven, Joshua Edmison, and Matthew Benz. I cannot go without acknowledging my other great friends in the CCM Lab with whom I shared classes throughout my enrollment at Virginia Tech: Matthew Blanton, Eric Lorden, Tingting Meng, Jorge Surís Pietri, Yousef Iskander, and Alexander Marschner.

I would like to thank my parents, LeAnn Nease Brown and Charles Edward Webb, and my stepfather Charles Gordon Brown. Without their wisdom and support throughout my life, I would not have been able to accomplish all that I have. It is only through their sharing

of knowledge and experience that I have been able to succeed. I would also like to thank my younger brother, Edward Austin Webb; without him as a brother and friend, my life would not be complete.

I cannot thank enough my friend Claire Andross enough for her love and support through my graduate studies and completion of this thesis. If it were not for her help and encouragement, I might not have finished this work.

Last, but not least, I would like to thank all of my friends and brothers in the Fraternity of Phi Gamma Delta. Without the experiences I had with you during my time at Virginia Tech, I would not have become the person I am today. I love all my brothers, and I am proud to be a Fiji.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Contribution . . . 2

1.3 Thesis Organization . . . 2

2 Literature Review **4**

2.1 FPGAs . . . 4

2.2 Dynamic Partial Reconfiguration . . . 5

2.2.1 Run-time reconfiguration Advantages . . . 5

2.2.2 Partial Reconfiguration . . . 6

2.2.3 Self-Reconfiguration . . . 8

2.3 Bitstream Manipulation . . . 8

2.4 FPGA Security . . . 10

2.4.1 Security Classifications . . . 10

2.4.2 Attacks and Protection . . . 12

Non-Invasive Attacks . . . 12

Blackbox Attack 12
Bitstream Interception or Readback Attack 12
Side-Channel Attacks 14
Invasive Attacks 14
Semi-Invasive Attacks 15
Active Photon Probing 15
Fault Injection 15

3 Design Considerations **17**

3.1 Security Assumptions and Types of Attacks 17
3.2 Attacker Classification 19
3.3 Proposed Design 22
3.3.1 Configuration Integrity Checker 23
3.3.2 Partial Authenticator 25
3.3.3 Challenge-Response Protocol 27
3.3.4 Partial Authenticator and Challenge-Response Protocol Keys 29

4 Implementation **31**

4.1 Platform 31
4.2 On-chip Configuration Integrity Checker 32
4.2.1 Xilinx SEU 32
4.2.2 Overall Checker Structure 32

4.2.3 ICAP Readback Controller 33
4.2.4 Dynamic Data Mask Controller – Ignoring Dynamic Data 35
4.2.5 MD5 Hash Function Controller 43
Choice of Hash Function – MD5 43
Choice of Input Window and Checking Granularity 44
Controller Operation 44
4.2.6 Hash Comparator 46
4.3 Partial Authentication and Reconfiguration 46
4.3.1 Xilinx Encryption 46
4.3.2 Authentication Method 46
4.3.3 Software Preparation of Partial Bitstream 47
4.3.4 Authentication Hardware 47
4.3.5 Partial Reconfiguration 48
4.4 Challenge-Response Protocol 49
4.4.1 Challenge from Verifier 49
4.4.2 Claimant Response 50

5 Results and Analysis 51

5.1 Operation Verification 51
5.2 Device Utilization 54
5.3 Timing Analysis 55
5.3.1 Clock Frequency 55

5.3.2 Execution Time 56
Configuration Integrity Checker 56
Partial Authenticator 59
Challenge-Response Protocol 60
5.4 Security Analysis 60
5.4.1 MD5 Collision Considerations — Brute Force Attack 60
5.4.2 Design-Based Attacks 62
5.4.3 Defense Strength 63
5.5 Run-time Repair and Checking Granularity Considerations 64

6 Conclusion **67**
6.1 Summary 67
6.2 Future Work 69

Bibliography **71**

List of Figures

3.1 This figure illustrates the configuration integrity checker scanning the CLB section of the configuration. The hash function has an input window of length n. The output of the hash function is dependent upon the previous input and begins with an initial value (IV). 24

3.2 This figure illustrates a malicious alteration of the configuration changing the output of the hash function as calculated in Figure 3.1 24

3.3 Data authentication using a hash function and an encryption algorithm . . . 26

3.4 Data authentication using only a hash function 27

3.5 Challenge-response protocol using a hash function and a secret key. 28

4.1 Block diagram of Configuration Integrity Checker 33

4.2 Flow Chart of ICAP Readback Controller State Machine 34

4.3 This figure illustrates a frame that contains flip-flops. It spans 16 CLBs in height and contains 128 flip-flops. 39

4.4 Flow Chart of MD5 Hash Function Controller State Machine 45

4.5 Illustration of partial bitstream preparation software 48

4.6 Illustration of partial authentication hardware 49

4.7 Illustration of challenge-response hardware 50

5.1 The design protected by the security system is an implementation of the Lucas-Lehmer test for Mersenne Primes. 53

5.2 The first verification experiment swapped the routes to the two LEDs. 53

5.3 The second verification experiment inverted the outputs of the Lucas-Lehmer test. 54

5.4 Resource utilization percentage of each subsystem in the security system . . 56

5.5 Illustration of operations that must execute in the critical timing path of the MD5 core 57

5.6 This figure presents a graph of the amount of time it takes for the configuration integrity checker to scan the entire CLB section and for the partial authenticator to reconfigure the entire CLB section for each part available for in the Virtex-4 LX platform. 58

5.7 This figure presents a graph of storage space versus the time to reconfigure. The point at (401μs, 21.504kbits) indicates the checking granularity chosen. . 66

6.1 Block Diagram of Security System 68

List of Tables

3.1 An attack matrix indicating the observation and attack capability of an attacker. Light gray is the least effective attack and dark gray is the most effective attack. 20

4.1 Resources available on the Virtex-4 LX 25 31

4.2 Frame address of all frames containing flip-flop bits 37

4.3 Offset of flip-flop bits within each frame from Table 4.2 38

4.4 Frame Address Register Description 40

4.5 Numeric representation of frames containing flip-flop bits 41

4.6 Statistics of frames used and ignored by the configuration integrity checker . 42

5.1 Overview of the resources utilized by the system as a whole 54

5.2 Overview of the resources utilized by the system on larger devices available in the Virtex-4 LX platform 55

Chapter 1

Introduction

1.1 Motivation

Since the introduction of the first Field Programmable Gate Arrays (FPGAs) in the 1980s, the use of FPGAs has expanded from implementing only glue logic to the implementation of entire System-on-Chip (SoC) designs. As a result, FPGAs have become integral parts of embedded systems for both research and commercial products. Consequently, the security of the FPGA is critical to the systems that use them.

The majority of research in this field has focused on the issues of mapping various well known algorithms to reconfigurable hardware [1,2,3], device technology [4,5], reconfigurable advantages [6,7,8,9], hardware-software co-design [10,11,12], compilation [13,14], and simulation and debugging [15,16]. Research regarding FPGA security, however, has only recently been of interest. It has primarily focused on the use of FPGAs for software security [17] or protecting the intellectual property contained in a device's configuration, rather than the security and integrity of the system itself [18,19,20,21,22,23]. Furthermore, vendors assume that by keeping the architectural details and the format of the configuration data of their devices proprietary, the design contained in the configuration data is secure [24]. Research

has shown that properties of FPGAs can be exploited and that neither the implemented system nor intellectual property is protected by proprietary-based obscurity [20, 25, 26].

1.2 Contribution

Presented in this thesis are methods for securing the integrity of the running configuration of FPGAs. To accomplish this, a system consisting of three parts is designed and implemented. Each subsystem is created to run on-chip in user logic.

The first subsystem monitors the FPGA's configuration. This subsystem detects if the configuration is altered by an attacker. Also presented are attacks that threaten the configuration as well as techniques an attacker may use. The second subsystem is a partial bitstream authenticator that verifies that an incoming partial is from a trusted source and has not been altered. The partial bitstreams may be reconfiguring the FPGA for another functionality or repairing the configuration from malicious alterations detected by the configuration integrity checker. The third subsystem is a challenge-response protocol that allows an entity external to the FPGA to verify that the security system has not been compromised. All of the subsystems achieve their aspect of protection using schemes that involve a hash function.

The security system is tested using simulations of the attacks that it aims to defend against. Also presented is an analysis of how these attacks may compromise the system. Additionally, the feasibility of the implemented system based on its size and speed of operation is provided.

1.3 Thesis Organization

This thesis is organized into six major chapters. Chapter 1 presents the motivation and contributions of this work. Chapter 2 discusses background information and related published

work focusing on the areas of dynamic partial reconfiguration, bitstream manipulation, and FPGA security. Design considerations that were made before the implementation of the proposed systems are covered in Chapter 3. Chapter 4 describes the implementation details of the configuration integrity checker, partial authenticator, and challenge-response protocol. Results, such as device utilization, timing analysis, operation verification, and post implementation security analysis are given in Chapter 5. Finally, Chapter 6 summarizes the research performed and discusses future work.

Chapter 2

Literature Review

This chapter covers background topics and related work that pertain to this thesis. It begins with a general discussion of FPGAs, and follows with dynamic partial reconfiguration, bitstream manipulation, and related security issues.

2.1 FPGAs

FPGAs are semiconductor devices containing both programmable logic and programmable interconnects. Unlike ASIC (Application Specific Integrated Circuit) devices, an FPGA's function is not known at the time of manufacturing. Furthermore, the function defined by the FPGA's configuration can be changed as many times as desired. Though generally slower than ASICs because of their reconfigurability, FPGAs have several advantages including shorter time to market, reusability, rapid prototyping and debugging on the target hardware, and in-field updates [27]. Moreover, FPGAs offer the advantages of dynamic and partial reconfiguration [28].

2.2 Dynamic Partial Reconfiguration

Reconfiguration can be divided into two sub-categories, static and dynamic [29]. Static reconfiguration implies that a device is configured once at the outset, after which it does not change during the execution of the task at hand. Dynamic reconfiguration allows the device's configuration to change at any moment, enabling the device to adapt within changing environments. Furthermore, dynamic partial reconfiguration allows for only a portion of the configuration to change. Recent applications such as [30] demonstrate that dynamic partial reconfiguration offers the advantage of reduced silicon utilization by allowing the device to only be configured with the appropriate functionality for the current conditions. Two other areas of interest made possible by dynamic partial reconfiguration are evolvable hardware [31, 32] and fault-tolerance [33].

The following sections develop the concept of dynamic partial reconfiguration in stages. First discussed are the advantages run-time reconfiguration provides. The second section addresses partial reconfiguration and its difficulties. The final section introduces the concept of self-reconfigurability.

2.2.1 Run-time reconfiguration Advantages

Run-time reconfiguration (RTR) is the process of reconfiguring the FPGA while it is operating. The main advantage RTR offers is the ability to reduce both software and hardware complexity. An example is given in [8] in which a hardware-software interface is optimized using RTR. Software drivers interfacing with register-level hardware are replaced with only static registers in the hardware whose values are altered by reconfiguring the device. Also discussed is the possibility of folding the register values into the design and optimizing the hardware for the given value of the registers. In [8], it is demonstrated that run-time reconfiguration can remove the software-hardware interface as well as reduce the amount of silicon needed to implement the design. This methodology can be further expanded to large

SoC designs, in which entire systems are replaced during operation using run-time reconfiguration.

2.2.2 Partial Reconfiguration

Partial reconfiguration further promotes optimization by offering the ability to modify only a portion of the design. By altering distinct parts of the design, a system implemented on an FPGA can have sections still operating while others are being replaced; however, achieving this functionality has proven to be complex, as partial reconfiguration is a difficult task [34].

Since the 1990s, partial reconfiguration has been of interest to researchers who have demonstrated its usefulness [6, 7, 9]. Early systems such as the JERC [35] utilize the reconfigurable array of logic cells available on the Xilinx 6200 series devices [4]. These devices allow each logic cell to be individually accessed much like normal computer memory. Unfortunately, the overhead associated with this type of device does not allow for the FPGAs to have large configuration densities. The Xilinx Virtex series of FPGAs supports partial reconfiguration and provides sufficient capacity for implementing or prototyping complete configurable systems [36]. Unlike the 6200 series, the smallest addressable unit on the Virtex FPGAs is a frame. Each frame may contain configurable logic blocks (CLBs), input-output blocks (IOBs), block RAMs (BRAMs), clock resources, programmable routing, and configuration circuitry [37]. The FPGAs are configured using a bitstream that contains the data for each frame. Partial reconfiguration requires the creation of partial bitstreams that work with the frame-based architecture.

Several contributions have been made toward refining the procedure of partial bitstream creation. In the application of JRTR [38], a host computer controls the reconfiguration process. The configuration is altered in a Java application, which ultimately uses JBits [39] to manipulate the original configuration bitstream and create a partial bitstream. Due to the low-level at which JBits operates, other solutions have been developed that provide abstraction for the designer. In the implementation of Dynamic Hardware Plugins (DHP) [40], the

PARBIT tool [34] is used to alter the configuration. Though still operating at the bitstream level, the PARBIT tool extracts the configuration bits related to an area defined by the user in order to create the partial bitstream. This allows the user to define the configuration changes at a higher level of abstraction.

Further experimentation has been conducted so that the designer may create partial bitstreams at the modular level. Designs specified behaviorally–by functionality as opposed to a specific hardware structure–are often broken up into modules. Each module can describe various functioning segments of the design. Partial creation at the modular level often allows for modules to be specified as static or dynamic. The compilation tools in [13] evaluate a design and distinguish reconfigurable regions. The regions are identified when two or more modules drive the same output. The tools not only provide partial bitstream development at the modular level, but also work toward making it an automated process.

Xilinx has also increased its software tool support for partial reconfiguration. Xilinx offers two partial reconfiguration flows as part of their build sequence [41]. The module-based flow is more applicable when large blocks are to be reconfigured. It allows for modules to be constrained to specific areas of an FPGA. The specific area can be partially reconfigured, effectively changing the functionality of the constrained module. This also allows a designer to know what functionality is being manipulated when reading back or modifying a particular section of the design. Mastering module-based flows is an area of current research, as recent works [42, 43] have sought new ways of simplifying the process.

The second flow, the difference based flow, allows a designer to make small changes to the design, usually in FPGA Editor [44], and generate a partial bitstream based on the differences between the two designs [45]. In this thesis, this method is used to create the partial bitstreams that test the system by simulating a fault injection attack. This method is preferred in this thesis because the small changes resemble the bit manipulations an attacker might make to the currently running functionality rather than completely loading a new functionality as is usually done in module-based partial reconfiguration.

2.2.3 Self-Reconfiguration

The manner in which an FPGA is configured with a partial bitstream is another important area of research. Oftentimes, configuration is done off-chip using a host processor. As dynamic partial reconfiguration becomes more feasible for real world applications, area and cost factors make the dependency on a host computer an unattractive consideration. Self-reconfiguration allows configuration control protocols to be implemented in user logic. This enables a module that determines what configuration should be implemented as well as the configuration controller to be within an FPGA; it allows the device to be in control of its own functionality.

An application of partial self-reconfiguration is developed in [46]. A Xilinx Virtex-II FPGA is reconfigured through the Internal Configuration Access Port (ICAP) using bitstreams stored in the FPGA's Block RAM (BRAM). The implementation demonstrates many benefits of self-reconfiguration, including flexibility in both configuration methods and media sources as well as security considerations. Partial self-reconfiguration could be used to help an FPGA defend itself from malicious attacks. The system presented in this thesis will be running on the FPGA that is susceptible to attack. Should the FPGA configuration become corrupted, the system will detect the alteration and the FPGA itself will know that it has been compromised. As will be discussed in Section 5.5, the system could also request partial bitstreams to reconfigure and repair the altered area. Partial run-time self-reconfiguration can allow the system to defend itself in the event of an attack.

2.3 Bitstream Manipulation

As described earlier, bitstream manipulation can be used for partial bistream generation. Due to its tedious nature, the usefulness of bitstream manipulation compared to other methods of partial bitstream creation may be questioned; however, its low-level access can be appealing because the regular tool chain may be bypassed entirely, allowing for faster gen-

eration of partial bitstreams.

Mentioned previously, JBits [39] is one of the better known bitstream manipulation tools. It is a set of Java classes that provide an Application Programming Interface (API) to manipulate Xilinx configuration bitstreams. JBits can be used to quickly alter details as minute as look-up table (LUT) contents. Its features can also be used to check the state of elements using configurations readback from the FPGA. Additionally, other applications such as REPLICA [47] have found different and sometimes quicker methods. REPLICA is an extension of PARBIT [34]. Acting like a filter, it reduces the overhead of PARBIT by allocating the bits for a dynamic area while programming the target FPGA.

Knowledge of the configuration at the bit-level is also important for security considerations. As will be discussed in the Section 2.4, certain attacks and thus their corresponding protection strategies can utilize bit manipulation of the configuration. Unfortunately, designing a system that will use bitstream manipulation is difficult. The challenges arise from a lack of information about the structure of the bitstream. In order to protect intellectual property and to hypothetically enhance configuration security, FPGA manufacturers are reluctant to release information pertaining to the architecture of the bitstream.

The necessity of this information is shown by the Alternate Wire Database (ADB) [48] tool in which Xilinx proprietary files, as well as information from JBits, are required. ADB is an application that can interface with JBits to provide routing, unrouting, and tracing services. Its wiring services are helpful when exhaustive wiring support is needed [48]. Though some details needed to do such a task can be derived from Xilinx's documentation on the Virtex Series Configuration Architecture [37], this information is primarily intended for reading and manipulating BRAM and LUT contents and is therefore insufficient for general logic and routing configuration [27]. Furthermore, information pertaining to or applications using bitstream manipulation have not targeted newer families such as the Virtex-4 and Virtex-5.

2.4 FPGA Security

It is imperative to consider FPGA security as FPGAs continue to become integral parts of embedded systems [25]. To ensure security, system components and their interaction with the FPGA must be examined [25]. This thesis, however, focuses on the security aspects of FPGA devices themselves. The following sections cover types of attacks on FPGAs and corresponding protection mechanisms. First introduced are some classifications that can be used to evaluate FPGA security systems.

2.4.1 Security Classifications

Before developing methods to protect a system, the objective of an attacker must be known. When attempting to gain access to the design implemented on an FPGA the objective of an attacker is most likely one of the following.

- The attacker is attempting to clone the design. Cloning allows the attacker to make an exact copy of the design.
- The attacker is attempting to reverse-engineer the design. An attacker reverse-engineers a design by reconstructing a "schematic" representation. In this process, the attacker understands how the design works and how to improve it or modify it with malicious intent.
- The attacker is attempting to alter the functionality of the design. The attacker does not aim to steal the design and implement it on another FPGA. Instead, the attacker wants to alter the design so that it has a different functionlity on the FPGA that it is currently running.

Furthermore, the level of an attacker's expertise should also be considered. Researchers at IBM [49] categorize attackers into three levels, depending on their expected skill and attack strength.

Clever outsider – These attackers are often very intelligent, but may have limited knowledge of the system. They may have access to only moderately sophisticated equipment. They often try to take advantage of an existing weakness in the system, rather than try to create one.

Knowledgeable insiders – These attackers have substantial specialized technical education and experience. They have varying degrees of understanding for different parts of the system, but potential access to most of it. They often have access to highly sophisticated tools and instruments for analysis.

Funded organizations – These attackers are able to assemble teams of specialists with related and complementary skills, backed by great funding resources. They are capable of performing an in-depth analysis of the system, designing sophisticated attacks, and using the most sophisticated analysis tools.

Security levels are also described. Divided into six stages, the security definitions rank the strength of an attack required to overcome a given security implementation.

ZERO – No special security features are added to the system.

LOW – Some security features are in place. They are relatively easily defeated with common laboratory or shop tools such as pliers, soldering iron, or small microscope.

MODL – More expensive tools are required, as well as some special skills and knowledge.

MOD – Special tools and equipment are required, as well as some special skill and knowledge. The tools and equipment can be much more expensive than for MODL. The attack may become time-consuming, but will eventually be successful.

MODH – Equipment is required, but is expensive to buy and operate. Equipment will more than likely be much more expensive than that required for MOD. Special skills and knowledge are required to utilize the equipment. More than one operation may be

required so that several adversaries with complementary skills would have to work on the attack sequence. The attack could be unsuccessful.

HIGH – All known attacks have been unsuccessful. Some research by a team of specialists is necessary. Highly specialized equipment is necessary, some of which might have to be designed and built. Total costs of the attack could be in the millions of dollars. The success of the attack is uncertain.

Using the above information, the strength of a security system created to protect an FPGA's design can be classified; however, the true level of security can only be determined through testing.

2.4.2 Attacks and Protection

FPGA attacks can be split into three categories: non-invasive, invasive, and semi-invasive [26]. Each category is discussed in this section. Also provided are some example attacks, protection methods against such attacks, and published works relating to the attacks and preventions.

Non-Invasive Attacks

Non-invasive attacks do not physically harm the FPGA. Using external means, the design can be cloned or reverse-engineered by the attacker.

Blackbox Attack

Using the blackbox attack, the attacker inputs all possible combinations and records the corresponding output. The attacker is then able to re-create the FPGA design using the gathered data. The growing complexity of FPGA designs renders this attack infeasible in modern applications [25].

Bitstream Interception or Readback Attack

In this attack, the configuration is either stolen during its transmission to the FPGA or during an external readback of the configuration. Then, the attacker has the ability to program his or her own FPGA or create a schematic of the design using the bitstream. To protect against the configuration being stolen through readback, many FPGAs allow for readback to be disabled. External configuration on the otherhand is often a necessity. A common solution is encryption of the configuration bitstream.

Xilinx offers bitstream encryption based on a triple DES scheme for its Virtex-II and Virtex-II Pro [50,51] devices, and an AES scheme for its Virtex-4 and Virtex-5 [52,53] devices. The Xilinx bitstream encryption scheme is effective because without the correct encryption key it is not possible to configure other FPGAs with the encrypted bitstreams; however, when using Xilinx's bitstream encryption, neither partial reconfiguration nor readback is permitted on the Virtex-II, Virtex-II Pro, or Virtex-4 LX/SX/FX12 devices. Moreover, these operations can only be performed internally, via ICAP, on the Virtex-4 FX20-FX140 or Virtex-5 devices. Yet another limitation is that the key is stored in internal, dedicated RAM, which requires a supply battery.

To overcome these limitations, other methods have been proposed and implemented. One method uses a key that is laser cut into the die [21]. During the first configuration, bitstreams would pass through the FPGA where they would be encrypted and then loaded into external memory. Then, the FPGA would simply decrypt the bitstream during normal programing operations. The use of the laser cut key would obviate the need for a battery backup. Moreover, this methodology removes the encryption process from the tool flow.

The method proposed in [21] still requires encryption and decryption circuits in the hardware, which increases the required FPGA silicon area. To eliminate these hardware requirements, a solution that uses dynamic configuration is proposed in [22]. First, the FPGA is configured with the necessary encryption circuitry and the bitstream is encrypted using the FPGA. Then the encrypted bitstream is loaded into external memory and the encryption circuitry is removed. Also loaded into external memory is the necessary decryption circuitry.

When the FPGA needs to be reconfigured, it is first loaded with the decryption circuitry, then decrypts the encrypted bitstream and reconfigures itself accordingly. This method still relies on a key inside the FPGA, and thus a laser cut key or a key stored in battery powered RAM would have to be used. Due to the encryption and decryption circuitry not being static, this method also allows for different encryption methods for different parts of the configuration. The method in [22] was implemented in [23] for partial reconfiguration. In this implementation, an on-chip processor connected to the ICAP was used to load the correct decryption circuitry, decrypt a partial bitstream, and partially reconfigure the housing FPGA.

In the scope of this thesis, the integrity of a partial bitstream during the configuration of the FPGA is of interest. A man-in-the-middle attack in which the partial bitstream is not stolen, but altered on the way to configure the device, is possible. The encryption methods described in this section protect configuration confidentiality, but do not necessarily protect the configuration's integrity as malicious alteration could occur during the transmission.

Side-Channel Attacks

The physical nature of FPGAs might provide a side-channel that leaks information. Examples of side-channels include power consumption, timing behavior, and electromagnetic radiation [25]. Power analysis has proven to be a practical threat [54]. Proposed counter measures include hardware alterations, such as noise addition, and software alterations, such as design obfuscation. In this thesis, side-channel attacks could allow an attacker to gain knowledge of the protected design and security system. Such knowledge would allow the attacker to develop a more effective attack.

Invasive Attacks

Invasive attacks physically damage the FPGA. They require opening the device to have access to the underlying hardware. Normally, invasive attacks are used as an initial step to

understand the chip functionality and then develop cheaper and faster non-invasive attacks. This attack is often expensive and difficult because it takes advanced measures due to FPGA complexity [25, 26]. As these attacks leave the FPGA inoperative, they are not discussed in detail in this thesis.

Semi-Invasive Attacks

A newer group of attacks called semi-invasive attacks have recently been classified [26]. Like invasive attacks, semi-invasive attacks require access to the FPGA's surface; however, the FPGA will remain functional after the attack. Furthermore, unlike invasive attacks, semi-invasive attacks do not require expensive equipment or a lot of expertise to perform. Two prominent methods of attack in this category are active photon probing and fault injection.

Active Photon Probing

Semiconductor transistors are sensitive to ionizing radition. In active photon probing, a scanned photon beam interacts with an integrated ciurcuit (IC). Using instruments such as laser scanning microsopes, the state of transitors in an IC can be read. In light-induced voltage alteration (LIVA), a laser is scanned across an FPGA's surface while the voltage changes of the power supply are observed [26]. From these observations images can be created that illustrate the structure of the chip. Furthermore, by aiming the laser beam at a specific transitor, it is possible to distinguish between two different memory states. In [26], it is demonstrated that the state of SRAM cells can be observed. If the top portion contains more voltage, then the cell is in a '1' state. If the bottom portion contains more voltage, then the cell is in a '0' state. Active photon probing can allow an attacker to gain knowledge of the design implemented on the FPGA.

Fault Injection

Also demonstrated in [26], SRAM cell alteration is possible. Fault injection can be used to modify the contents of SRAM and change the state of any transistor inside the chip. In [26],

fault injection is accomplished using both an inexpensive photoflash lamp and a laser pointer. The attack could control whether or not an SRAM cell was a '0' or a '1'. Fault injection can allow an attacker to maliciously alter an FPGA's configuration at the bit-level.

Research has been conducted to protect against faults caused by naturally occurring radiation. Such bit upsets, referred to as single-event upsets (SEUs), are often a concern in space applications due to the radiation present in low-earth orbit [55]. In [56] the capabilities of dynamic reconfiguration are used to compensate for SEUs. In the application, readback was used to constantly compute a cyclic redundancy check (CRC) for each frame in the FPGA. If an SEU occurred, the CRC for the frame in which the upset corresponds would have a different CRC result. The system then partially reconfigured the device to correct the altered frame. Xilinx offers its own SEU detection and correction device [57]. Operating similarly to the implementation in [56], the device will detect and notify user logic if an SEU occurs. If operating in correction mode, the device can correct SEUs; however, if a multiple-event upset occurs, the device cannot correct the configuration.

Nevertheless, these methodologies are not meant to protect against fault injection attacks that aim to maliciously alter an FPGA's configuration. As described in Section 2.3, such attacks seems difficult because they require knowledge of bitstream composition as well as SRAM cell layout; however, it is possible to deduce the architectural details necessary for constructing a malicious configuration without knowledge of any proprietary information [20]. A discussion in [20] explains that by exploiting the ability to alter the SRAM cells of an FPGA and observe the power consumption changes, an attacker can alter the logic enough to cause unpredictable behavior in the implemented design.

Chapter 3

Design Considerations

As described in the motivation of this thesis, the focus of this work lies in ensuring the integrity of the FPGA's running configuration and achieving this protection on-chip. First discussed in this chapter are assumptions of the security system and possible attacks in this problem space. Also provided is a classification of attackers given the types of attacks possible. Finally, a design of the security system is outlined.

3.1 Security Assumptions and Types of Attacks

Before a design of the security system can be proposed, it is important to consider assumptions of the system and the types of potential attacks the system aims to protect against. There are two assumptions that further clarify the security claims of the system described in this thesis. These assumptions hold true throughout the design and implementation of the system. The first assumption is that design privacy is not of concern. The security system presented is only intended to protect configuration integrity and not configuration privacy. If privacy needs to be assured, then other methods could be used in conjunction with this application. The second assumption is that only static configuration data will be protected.

Dynamic data, such as the state of flip-flops or BRAM, will not be protected by this system. The static portion pertains to aspects of the configuration that are set at build time and do not change during operation.

The objective of the security system described in this thesis is to protect a running FPGA configuration as well as reconfigured versions of it. Many of the attacks discussed in Section 2.4.2 intend to clone or reverse-engineer the design implemented on an FPGA, but do not attempt to alter it. The primary attacks that this security system aims to protect against are the following:

1. Fault injection: The attacker is able to physically set or clear the state of a bit or bits of the configuration using a device. As outlined in Section 2.4.2, an attacker could use a photoflash lamp or a laser pointer to alter the SRAM cells of the FPGA.

2. Partial tampering: The attacker can send a partial bitstream or alter an incoming partial bitstream to change the configuration. An attacker can achieve this attack by first observing on which pins of the FPGA partial bitstreams are transferred and then connecting stimuli to these pins to enter a partial bitstream.

Despite the fact that other attacks might not alter the configuration, they still could aid an attacker in developing an effective attack. If the attacker cannot observe the state of the bits of the configuration, then the attacker has no knowledge of what bits to alter. From Section 2.4.2, some examples of attacks that could aid an attacker in gaining knowledge of the design are the following:

1. Side-channel attacks: A side-channel attack, such as power analysis, could allow an attacker to determine what parts of the FPGA are functioning.

2. Active Photon Probing: Semi-invasive observation, such as LIVA, could allow an attacker to view the state of bits on the FPGA. If used in conjunction with a fault injection attack, the attacker could observe the state of any bit on the FPGA before and after an alteration.

3. External pin monitoring: By connecting a logic analyzer to the pins that transfer a partial bitstream to the FPGA, an attacker could build a database of configuration information and piece together the design. If the system were to utilize run-time repair, as will be discussed in Section 5.5, and the attacker is using fault injection, the attacker could capture every bit of configuration data by storing the partial bitstreams sent to repair the altered area. The attacker would then have complete knowledge of the design at the bit-level.

3.2 Attacker Classification

The following attacker classification is focused on the first attack, fault injection. The second type of attack, partial bitstream tampering, would be an attempt to bypass basic cryptographic authentication schemes and will be discussed in Section 3.3.

The capability of an attacker attempting fault injection can be divided into categories based upon two criteria. Each criterion can be further broken down into ability levels.

1. Observation Capability

   None The attacker does not have enough observation ability to gain knowledge of functioning areas or the design at the bit-level.

   Unfocused The attacker has enough observation ability to gain knowledge of areas that contain functional aspects of the design, but not the design at the bit-level. Such observation ability could come from a power analysis attack.

   Specific The attacker has enough observation ability to gain knowledge of the design at the bit-level. Examples attacks are active photon probing or monitoring incoming partial bitstreams.

2. Attack Capability

Weak The attacker can only alter bits of configuration data at random. The attacker might be using a photoflash lamp to achieve fault injection, but cannot focus the lamp at specific bits or even a specific area of the FPGA.

Medium The attacker can alter bits in a specific area, but not specific bits. The attacker might be using a photoflash lamp with only enough focus to target a specific area or a laser with a control mechanism that cannot accurately pinpoint locations as small as an SRAM cell.

Strong The attacker can alter any desired bit. The attacker might be using a focused photoflash lamp or a laser pointer, both of which would have a precision control mechanism.

Based on these two criteria, an attacker can be classified in a 3x3 matrix that is shown in Table 3.1.

		Attack Capability		
		Weak	Medium	Strong
Observation Capability	None	Case 1	Case 4	Case 7
	Unfocused	Case 2	Case 5	Case 8
	Specific	Case 3	Case 6	Case 9

Legend: Class 1 (Cases 1, 2, 3, 4, 7); Class 2 (Cases 5, 6, 8); Class 3 (Case 9)

Table 3.1: An attack matrix indicating the observation and attack capability of an attacker. Light gray is the least effective attack and dark gray is the most effective attack.

The matrix entries, referred to as cases, fall into three classes that are defined as follows.

Class 1 The light gray entries indicate attacks that are the least threatening. For all cases in the first column, the attacker's observation capability does not matter, because the attacker cannot control the placement of the attack. Thus, all cases in the first column

have an identical chance of success. Attacks from the first column might be using a photoflash lamp that cannot be directed to a specific bit. With strong observation capability the attacker has gained knowledge of the entire design by collecting partial bitstreams that have been sent to repair the alterations; however, because the attacker cannot direct the attacks, the knowledge cannot be effectively used.

Similarly, all the attacks in the first row have an identical chance of success. In these cases, the attacker has no ability to observe changes to the design or gain knowledge of the design, so the attack capability is inconsequential. An attacker from the first row might have a laser mounted on a base that can accurately pinpoint any SRAM cell. Though the attacker can alter any bit, the lack of observation capability prevents gaining any knowledge about the design and thus, the attacker does not know what SRAM cells to target. All attacks that fall in Class 1 would manifest as random errors in the design, similar to SEUs.

Class 2 The medium-gray entries indicate mid-strength attacks. In Case 5, the attacker only has enough observation and attack capability to direct an attack to a functioning area and not specific bits. This attack would appear analogous to an SEU only occurring in a specific region of the FPGA. The attacker might be using a photoflash lamp that cannot focus to a specific bit, but is accurate enough to focus at a specific area of SRAM cells. The attacker could also be using power analysis to gain knowledge of which areas of the FPGA are functioning; however, the attacker is not able to determine what these areas do.

In Case 6, the attacker has enough observation ability to eventually know the design at the bit-level, but does not have the means to target specific bits. This attacker could have the same photoflash lamp as Case 5, but has also gathered information from partial bitstreams sent to repair the alterations and has reverse-engineered the design. In Case 8, the attacker has the capability to target specific bits, but does not have enough observation capability to learn which specific bits to target. The attacker

could be using power analysis comparable to Case 5, but also has an accurate laser for altering SRAM cells. Both Case 6 and Case 8 would only be as effective as Case 5.

Class 3 Class 3 is the most effective set of attacks. The only entry falling in this class is Case 9. In this case, the attacker can change any bit desired and has enough observation capability to determine which bits should be targeted for an effective attack. The attacker will most likely be using a precision laser on an accurate mechanical control base, which allows the attacker to alter any desired SRAM cell on the FPGA. The attacker has determined on which pins a partial bitstream is transfered and has connected a logic analyzer to these pins. By attacking various SRAM cells on the FPGA, the attacker has collected all partial bitstreams sent to repair the device. The attacker has also learned which SRAM cells correspond to which frames of configuration data. The design maybe reverse-engineered from the partial bitstreams, yielding knowledge of both the protected design and the security system.

3.3 Proposed Design

A three part design is presented in this section. First presented is a configuration integrity checker intended to protect the security of the running configuration. Discussed second is a partial authenticator, which aims to ensure that incoming partial bitstreams have not been altered. Third, a challenge-response protocol is developed that will signal whether the system has been compromised. All three solutions are intended to be implemented on-chip in user logic. This section concludes with a discussion of the keys needed for the partial authenticator and challenge-response protocol.

3.3.1 Configuration Integrity Checker

Configuration integrity is often checked after an FPGA has been fully configured. Readback is performed external to the chip in order to verify that the configuration on the chip is what was intended in the bitstream. To achieve this end, a comparison file and a corresponding mask file are needed from BitGen [52]. The mask file determines whether a bit should be ignored or not. Then, the masked bit is compared to the comparison file to check integrity.

Using this same method on chip would require storing these files, which would occupy significant space. Furthermore, if the configuration were changed using a partial bitstream, then a new mask file and comparison file would have to be provided. Nevertheless, readback of the configuration is a necessity. The checker must monitor the configuration constantly and to retrieve the configuration data, readback must be used. Because the files provided by BitGen will not be used to observe changes in the configuration, it is proposed that the design use a hash function. Hash functions output a digital "fingerprint" of their input, which can be data of any size. The output, known as the hash value, should be different if even a single bit is changed [58]. Thus, it is proposed that the configuration be read back and sent through a hash function as shown in Figure 3.1. The input to the hash function will be a window of length n that slides through the configuration at increments of n. The length of n will have to be defined in the implementation as the chosen hash function may be a determining factor.

If a malicious agent were to alter the configuration, as shown in Figure 3.2, the output of the hash function would change after the window passes over the modified section. The modification of even a bit of configuration data would change the hash function's output[1].

The checker will only monitor the CLB component of the configuration, which corresponds to the static portion. As will be discussed in the implementation section, there is still some

[1]There is some chance that a collision would occur and the hash output actually does not change though the configuration is altered. A collision occurs when two different messages have the same hash output. The probability of a collision occurring depends on the hash function used. An analysis of collisions are discussed in Chapter 5 for the hash function that is chosen in the implementation of this system.

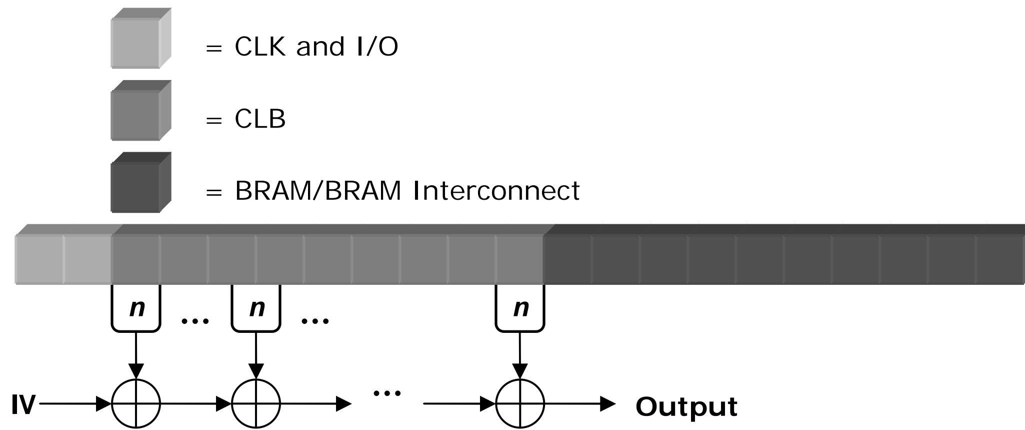

Figure 3.1: This figure illustrates the configuration integrity checker scanning the CLB section of the configuration. The hash function has an input window of length n. The output of the hash function is dependent upon the previous input and begins with an initial value (IV).

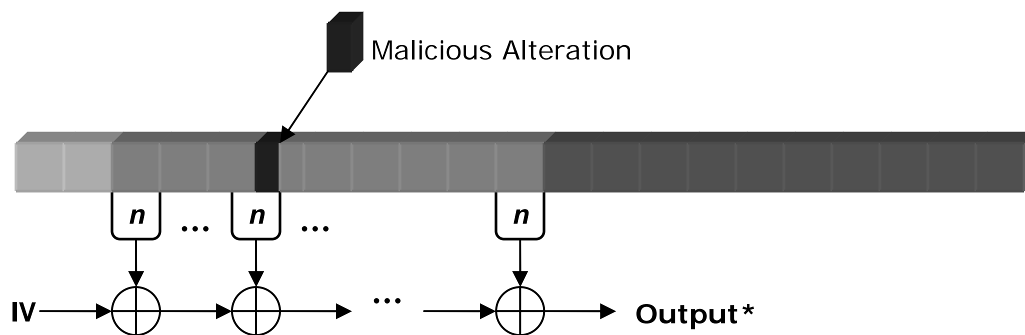

Figure 3.2: This figure illustrates a malicious alteration of the configuration changing the output of the hash function as calculated in Figure 3.1

dynamic configuration data spread throughout the CLB section, and this data will also have to be ignored.

The checker could be configured to create an output for the entire CLB section as shown in Figure 3.1. Another option is to have multiple hash values for different divisions of the CLB section. These divisions should be based on frames because they are the smallest addressable unit. The number of outputs and the division level, or checking granularity, can be chosen in the implementation phase and is further discussed in Sections 4.2.5 and 5.5. The only design dependencies resulting from this decision are that the number of outputs remain constant and that after every computation, each output is compared with the corresponding original value. Also, the checker does not have to monitor the entire CLB section. As long as the chosen protected area remains constant, the checker can monitor any subsection of the CLB data. If the checker does not protect the entire CLB section, the checker must be included in whatever region is chosen to be protected. If the checker is outside the protected area, an attacker can render it inoperable.

3.3.2 Partial Authenticator

The security system presented in this thesis is intended to protect the running configuration of a design that may require partial reconfiguration in order to operate. As will be discussed further in Section 5.5, partial reconfiguration could be used to repair the system should the integrity checker encounter an error in the configuration. If partial reconfiguration occurs, the running configuration will be altered. To ensure that these alterations are non-malicious, a partial bitstream must be authenticated to have originated from a trusted source. Origin authentication by definition includes data integrity. As stated in [58], data that has been altered effectively has a new source. If no source can be determined, then the question of alteration cannot be settled without a reference to the source. Thus, origin authentication mechanisms implicitly provide data integrity, and vice versa.

Two common ways of ensuring data integrity are illustrated in Figures 3.3 and 3.4. The

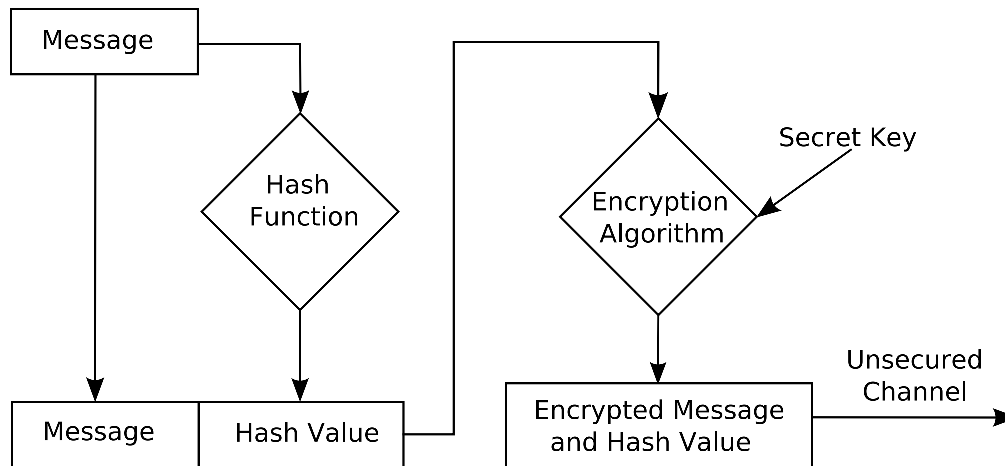

Figure 3.3: Data authentication using a hash function and an encryption algorithm

first method, Figure 3.3, involves the use of a hash function and an encryption scheme. Many methods discussed in Chapter 2 utilize encryption of bitstreams, but encryption alone does not guarantee data integrity [58]. Thus, to implement this scheme, both a decryption unit and a hash function unit would be needed in hardware. The receiver of the message must decrypt both the message and the attached hash value and then recompute the hash value of the message to ensure that the message has not been altered. The encryption proves the source of the message is trusted because knowledge of a secret key is required. The hash function validates message integrity. Without the decryption key the hash value is unknown because it is included in the encryption.

The second method, shown in Figure 3.4, only requires a hash function in implementation. In this method, the message that must be authenticated is either prepended or appended with a secret key. The only agents that would know the secret key are the trusted source and receiver of the message. After being sent through the hash function, the secret key is part of the resulting hash value. Even if a malicious third party knows the hash function being used for authentication, without the secret key, the agent cannot claim to be a trusted source of a message. In this case, the hash function validates the source of the message and the message integrity.

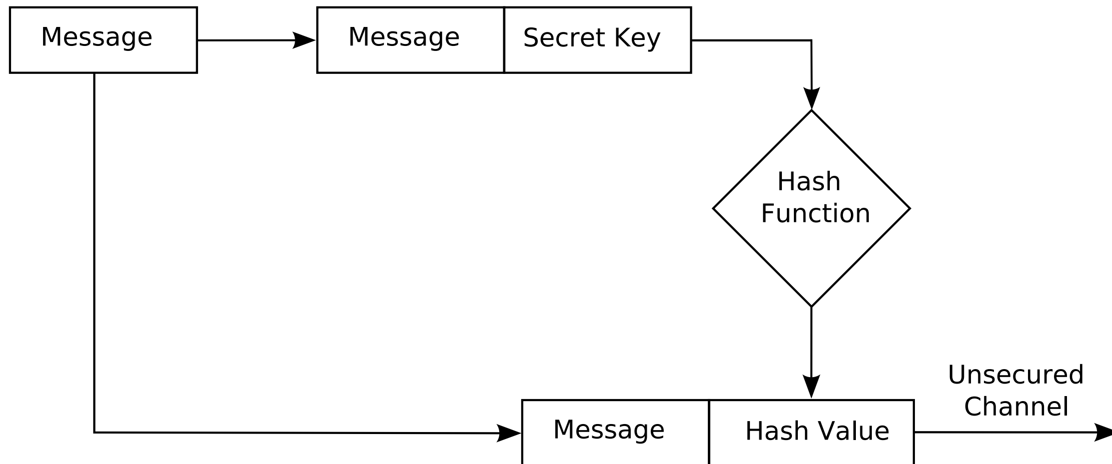

Figure 3.4: Data authentication using only a hash function

Due to privacy not being a concern in this system, as stated in the assumptions in Section 3.1, the second method is more appealing because only a hash function will be needed in hardware. Furthermore, a hash function will be used to implement the configuration integrity checker, as discussed in Section 3.3.1. If the two designs share a common hash function unit, resource utilization will be reduced during implementation. The choice of the exact hash function will be elaborated upon in Chapter 4. The checker could use error detecting hash functions such as CRC if it only needed to defend against Class 1 or possibly Class 2 attacks. Partial bitstreams in transit to the FPGA, however, are easily observable and alterable and the use of an error detecting hash function would compromise the security of the partial authenticator. Therefore, the use of a cryptographic hash function is proposed.

3.3.3 Challenge-Response Protocol

Should the security system become compromised in any way, there must be a method of notifying external parties, i.e. entities outside of the FPGA. The system is compromised if the integrity checker or partial authenticator ceases to operate correctly. Challenge-response protocols require one entity, the claimant, to prove its identity to another entity, the ver-

ifier [58]. This is done by having the claimant demonstrate knowledge of a secret that is known only to itself and the verifier. Furthermore, the secret should not be revealed during the protocol.

There are many ways of achieving challenge-response [58], but for this proposed system, using the same hash function from the previous two sections would be optimal. It is proposed that the system utilize the protocol illustrated in Figure 3.5. The protocol is based on keyed

(a) Transmission of random message to claimant (b) Transmission of hash value back to verifier

Figure 3.5: Challenge-response protocol using a hash function and a secret key.

hash functions, similar to the authentication device. The verifier transmits a message to the claimant, which is the security system, and the system must reply with a hash of the message appended with a secret key. In many challenge-response protocols, both entities participating want to validate the identity of the other; however, in this system, the purpose of the challenge-response is to simply indicate if the system is not working and thus, the identity of the verifier is not of concern.

The maximum number of times the system should be queried is once for every run of the configuration integrity checker. Querying more frequently than that would not not be helpful as the system would not know if it had been compromised until the checker attempts to monitor the configuration again. The minimum number of times would depend on the desired confidence level of the system's state of operation. It is proposed that the system be queried the maximum number of times possible, which is once after every scan of the configuration integrity checker.

The message for the challenge-response query should be random. This is so an attacker cannot use a replay attack by storing the output for a set response and posing as the system. Furthermore, to ensure that messages are not repeated, a time stamp should be included

in the message. Lastly, because the challenge-response protocol will be queried no more than once per scan of the checker, it should not be created to respond more often than this. Thus, if an attacker does query the system, the verifier will know because the system will not respond to the verifier's query.

3.3.4 Partial Authenticator and Challenge-Response Protocol Keys

Both the partial authenticator and the challenge-response protocol will require keys for operation. The key used for the challenge-response should not just be an alpha-numeric sequence stored in the hardware. Instead, the key should be based on a dynamic parameter that would change if the system is compromised. Furthermore, the key should be recomputed each time it is needed so that it does not have to be stored. Thus, it is proposed that the key be a hash of all hash values computed by the integrity checker after each scan of the configuration. This way, even if an attacker is able to render the checker inoperable, alter the configuration, and still force the challenge-response system to operate, the key would not be correct. This is due to the altered configuration changing the key. Thus, the verification entity would know that the system has been compromised because the hash value of the reply would be incorrect due to the different key. The verification entity would be using a hash of all hash values from the known trusted configuration. If the system is secure, then the hash value of the challenge-response protocol will be the same between the two entities.

It is also proposed that the key for the partial bitstreams be comprised of the configuration data; however, this key should not change after every scan of the hardware. Should the configuration be altered, and partial reconfiguration be used to repair the system, then the authentication key must agree with the entity which will transmit the partial bitstreams to the FPGA. This entity will be using the hash of the hash values from the trusted configuration, and thus, so should the partial authenticator.

The hash values of the trusted configuration are an initial condition of the security system. Therefore, the hash of these hash values which produces the key for the challenge-response

protocol and partial authenticator is the same. How this initial condition is met is implementation dependent and is discussed in Chapter 4.

Chapter 4

Implementation

This chapter discusses the implementation of the design proposed in Chapter 3. First, the platform used is introduced. Then, the implementations of the configuration integrity checker, partial authenticator, and challenge-response protocol are examined.

4.1 Platform

The security system is implemented on a Memec Virtex-4 MB Development Kit [59]. The platform consists of a Xilinx Virtex-4 LX 25 FPGA, 64MB of DDR SRAM, 4MB flash, 16-bit LVDS transceivers, USB-RS232 bridge, 10/100 Ethernet PHY, 100MHz clock source, and an RS-232 port. The LX 25 is the second smallest part offered for the Virtex-4 LX platform. A summary of its resources from [60] is given in Table 4.1.

Configurable Logic Blocks (CLBs)				**Block RAM**	
Array Row x Col	Logic Cells	Slices	Max Distributed RAM (Kb)	18 Kb Blocks	Max Block RAM (Kb)
96 x 28	24,192	10,752	168	72	1,296

Table 4.1: Resources available on the Virtex-4 LX 25

4.2 On-chip Configuration Integrity Checker

4.2.1 Xilinx SEU

Before discussing the implementation of the checker, the reasons for not using the Virtex-4 SEU controller mentioned in Section 2.4.2 are stated. The SEU controller contains functionality similar to that desired by the checker because it monitors the FPGA's configuration on-chip. The controller, however, is designed mainly for the detection of single-bit soft errors that could occur due to volatile environments such as those found in outer space. The SEU controller module can correct single bit upsets caused by such environments, but lacks sufficient functionality to correct multiple bit upsets. The SEU uses the Frame Error Correction Code (ECC) logic available on the Virtex-4 [52]. The Frame ECC uses a Hamming code parity rather than a cryptographic hash function, making the system more susceptible to malicious attacks. Furthermore, the SEU controller module has a predetermined checking granularity of a single frame. The SEU must store the frame it is checking in BRAM. While the frame is being stored, it is vulnerable to alterations by the attacker. The alterations could revert an attacker's previous changes so that the frame appears as though it has not been modified.

4.2.2 Overall Checker Structure

The on-chip configuration integrity checker is implemented as a finite state machine (FSM) that pulls in data from a readback controller, masks the data appropriately, and computes the hash function of the data. After each output of the hash function, the computed value is compared against the corresponding trusted hash value. If the values do not match, the checker raises a failure flag and outputs a number corresponding to which section of the configuration has failed. A block diagram of the checker is given in Figure 4.1. Each block of the checker is discussed in the following subsections.

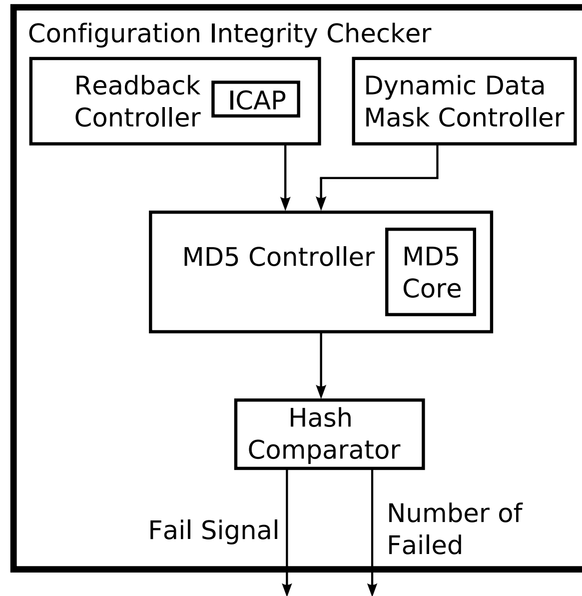

Figure 4.1: Block diagram of Configuration Integrity Checker

4.2.3 ICAP Readback Controller

The configuration data required by the configuration integrity checker must be produced from the readback process. Readback allows the configuration memory to be read from the JTAG, SelectMAP, or ICAP interface. Due to the configuration integrity checker operating on-chip inside user logic, the only available on-chip interface is the ICAP [52].

The ICAP interface allows for both programming and readback of the FPGA's configuration. In this design, a readback controller FSM is created to control readback. A high-level flow chart of the state machine is provided in Figure 4.2. The FSM first sends all necessary commands to the ICAP, preparing the ICAP to readback the data. These commands include the starting frame address and the number of frames to be read. After initializing the ICAP for readback, the controller transfers the ICAP into read mode. The state machine then clocks out the configuration bytes and sends them back to the checker state machine.

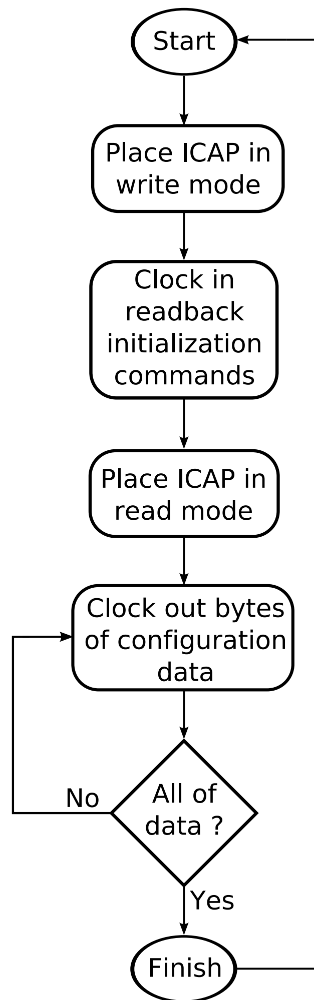

Figure 4.2: Flow Chart of ICAP Readback Controller State Machine

4.2.4 Dynamic Data Mask Controller – Ignoring Dynamic Data

Readback returns the state of all configuration data, both static and dynamic. Static data pertains to resources such as wires that are set at build time and do not change during the operation of the device. Dynamic data, such as flip-flops, changes while the device is operating. A single bit change in the readback configuration would change the computed hash value; thus, the dynamic data must be ignored. The readback controller is to read only the CLB section and consequently, the I/O and BRAM resources are ignored. Therefore, in normal operation, the only dynamic data that needs to be ignored is the flip-flop data. There are other modes that a CLB can operate in such as RAM; however, for this implementation, it is assumed that the only dynamic data that is necessary to filter out is the state of the flip-flops. If it was desired to handle CLBs that are configured as RAM, the configuration bit(s) that indicate that the CLB is being used in an alternate mode would have to be monitored. If the checker were to find a CLB in an alternate mode, the frames that consist of dynamic data while in this mode would have to be ignored.

The mask file available from BitGen used when verifying the configuration off-chip could serve the purpose of masking out dynamic data. Again, due to the space that this mask file would consume, this option is not desired on-chip. Furthermore, even a look-up table of all flip-flop locations in the configuration is not desired, because there are 21,504 flip-flops on this particular FPGA (10,752 slices * 2 flip-flop per slice). Consequently, a pattern of all flip-flop locations must be discerned to minimize the amount of resources needed to mask out the flip-flop bits.

To determine the location of all of the flip-flops, the Logic Allocation File (extension ".ll"), a file generated from BitGen, is used. The logic allocation file gives the frame address, bit offset within a frame, and bit offset within the bitstream of resources used in the design. As stated in Chapter 2, a frame is the smallest addressable unit of the configuration. A separate design is made that utilizes all flip-flops in one column of the FPGA. The logic allocation file for this design contains the location information for all flip-flops in that column. The

frame address and bit offset within the frame are used to determine the pattern of flip-flop bits. Using this information, both Table 4.2 and Table 4.3 can be generated.

Each entry in Table 4.2 corresponds to frames that contain flip-flops. Unlike the Virtex-II and Virtex-II Pro, a frame in the Virtex-4 only spans 16 CLBs in height [52]. As illustrated in Figure 4.3, each CLB contains four slices and each slice contains two flip-flops. Thus, each frame contains 128 flip-flops. Given that there are a total of 10,752 slices on the V4LX25, as given in Table 4.1, then there are a total of 2,688 CLBs and 168 blocks of 16 CLBs. These 168 blocks are broken into six segments using the frame address as shown in Table 4.2.

The frame address in Table 4.2 is decomposed based on the frame address register description from [52]. A summary of the register description is given in Table 4.4. The "Unknown" column in Table 4.2 comes from bits 6 through 8, which are not represented in Table 4.4. It is unknown if the documentation is incorrect or if Xilinx omitted these bits for proprietary reasons, but no explanation of these bits was found in Virtex-4 documentation. Notice that although there are 28 sets of CLBs for each of the six segments, the actual column address only increments after the first seven CLBs, the second seven CLBs, and then the following six CLBs. Thus, all of the frame addresses for these groups would be the same except for the bits in the Unknown column, which change throughout a group. Therefore, bits 6 through 8 are important in the frame address for determining exactly which block of 16 CLBs that frame pertains to.

Table 4.3 shows the offset of each of the 128 flip-flop bits that are located in each frame listed in Table 4.2. Table 4.3 is the same for each frame from Table 4.2, so the pattern for each flip-flop bit only has to be deduced once and then replicated for each frame that contains flip-flops. In Table 4.3, each bit from a column alternates between spanning 22 and 12 bits apart except for at the division in the middle of the pattern where the flip-flop bits are 44 bits apart. This pattern can be reduced by considering the CLB as a whole and noting that every two flip-flop bits are a distance of 27 or 11 bits apart, except for in the middle where the pair of bits are 43 bits apart.

1

	Frame Address	Top/Bottom Bit	Block Type	Row Address	Column Address	Unknown	Minor Address
CLB1	0x00000054	0	0	0	0	1	20
CLB2	0x00000094	0	0	0	0	2	20
CLB3	0x000000d4	0	0	0	0	3	20
CLB4	0x00000114	0	0	0	0	4	20
CLB5	0x00000154	0	0	0	0	5	20
CLB6	0x00000194	0	0	0	0	6	20
CLB7	0x000001d4	0	0	0	0	7	20
CLB8	0x00000214	0	0	0	1	0	20
CLB9	0x00000294	0	0	0	1	2	20
CLB10	0x000002d4	0	0	0	1	3	20
CLB11	0x00000314	0	0	0	1	4	20
CLB12	0x00000354	0	0	0	1	5	20
CLB13	0x00000394	0	0	0	1	6	20
CLB14	0x000003d4	0	0	0	1	7	20
CLB15	0x00000494	0	0	0	2	2	20
CLB16	0x000004d4	0	0	0	2	3	20
CLB17	0x00000514	0	0	0	2	4	20
CLB18	0x00000554	0	0	0	2	5	20
CLB19	0x00000594	0	0	0	2	6	20
CLB20	0x000005d4	0	0	0	2	7	20
CLB21	0x00000614	0	0	0	3	0	20
CLB22	0x00000654	0	0	0	3	1	20
CLB23	0x00000694	0	0	0	3	2	20
CLB24	0x000006d4	0	0	0	3	3	20
CLB25	0x00000714	0	0	0	3	4	20
CLB26	0x00000754	0	0	0	3	5	20
CLB27	0x00000794	0	0	0	3	6	20
CLB28	0x000007d4	0	0	0	3	7	20

2

	Frame Address	Top/Bottom Bit	Block Type	Row Address	Column Address	Unknown	Minor Address
CLB1	0x00004054	0	0	1	0	1	20
CLB2	0x00004094	0	0	1	0	2	20
CLB3	0x000040d4	0	0	1	0	3	20
CLB4	0x00004114	0	0	1	0	4	20
CLB5	0x00004154	0	0	1	0	5	20
CLB6	0x00004194	0	0	1	0	6	20
CLB7	0x000041d4	0	0	1	0	7	20
CLB8	0x00004214	0	0	1	1	0	20
CLB9	0x00004294	0	0	1	1	2	20
CLB10	0x000042d4	0	0	1	1	3	20
CLB11	0x00004314	0	0	1	1	4	20
CLB12	0x00004354	0	0	1	1	5	20
CLB13	0x00004394	0	0	1	1	6	20
CLB14	0x000043d4	0	0	1	1	7	20
CLB15	0x00004494	0	0	1	2	2	20
CLB16	0x000044d4	0	0	1	2	3	20
CLB17	0x00004514	0	0	1	2	4	20
CLB18	0x00004554	0	0	1	2	5	20
CLB19	0x00004594	0	0	1	2	6	20
CLB20	0x000045d4	0	0	1	2	7	20
CLB21	0x00004614	0	0	1	3	0	20
CLB22	0x00004654	0	0	1	3	1	20
CLB23	0x00004694	0	0	1	3	2	20
CLB24	0x000046d4	0	0	1	3	3	20
CLB25	0x00004714	0	0	1	3	4	20
CLB26	0x00004754	0	0	1	3	5	20
CLB27	0x00004794	0	0	1	3	6	20
CLB28	0x000047d4	0	0	1	3	7	20

3

	Frame Address	Top/Bottom Bit	Block Type	Row Address	Column Address	Unknown	Minor Address
CLB1	0x00008054	0	0	2	0	1	20
CLB2	0x00008094	0	0	2	0	2	20
CLB3	0x000080d4	0	0	2	0	3	20
CLB4	0x00008114	0	0	2	0	4	20
CLB5	0x00008154	0	0	2	0	5	20
CLB6	0x00008194	0	0	2	0	6	20
CLB7	0x000081d4	0	0	2	0	7	20
CLB8	0x00008214	0	0	2	1	0	20
CLB9	0x00008294	0	0	2	1	2	20
CLB10	0x000082d4	0	0	2	1	3	20
CLB11	0x00008314	0	0	2	1	4	20
CLB12	0x00008354	0	0	2	1	5	20
CLB13	0x00008394	0	0	2	1	6	20
CLB14	0x000083d4	0	0	2	1	7	20
CLB15	0x00008494	0	0	2	2	2	20
CLB16	0x000084d4	0	0	2	2	3	20
CLB17	0x00008514	0	0	2	2	4	20
CLB18	0x00008554	0	0	2	2	5	20
CLB19	0x00008594	0	0	2	2	6	20
CLB20	0x000085d4	0	0	2	2	7	20
CLB21	0x00008614	0	0	2	3	0	20
CLB22	0x00008654	0	0	2	3	1	20
CLB23	0x00008694	0	0	2	3	2	20
CLB24	0x000086d4	0	0	2	3	3	20
CLB25	0x00008714	0	0	2	3	4	20
CLB26	0x00008754	0	0	2	3	5	20
CLB27	0x00008794	0	0	2	3	6	20
CLB28	0x000087d4	0	0	2	3	7	20

4

	Frame Address	Top/Bottom Bit	Block Type	Row Address	Column Address	Unknown	Minor Address
CLB1	0x00400054	1	0	0	0	1	20
CLB2	0x00400094	1	0	0	0	2	20
CLB3	0x004000d4	1	0	0	0	3	20
CLB4	0x00400114	1	0	0	0	4	20
CLB5	0x00400154	1	0	0	0	5	20
CLB6	0x00400194	1	0	0	0	6	20
CLB7	0x004001d4	1	0	0	0	7	20
CLB8	0x00400214	1	0	0	1	0	20
CLB9	0x00400294	1	0	0	1	2	20
CLB10	0x004002d4	1	0	0	1	3	20
CLB11	0x00400314	1	0	0	1	4	20
CLB12	0x00400354	1	0	0	1	5	20
CLB13	0x00400394	1	0	0	1	6	20
CLB14	0x004003d4	1	0	0	1	7	20
CLB15	0x00400494	1	0	0	2	2	20
CLB16	0x004004d4	1	0	0	2	3	20
CLB17	0x00400514	1	0	0	2	4	20
CLB18	0x00400554	1	0	0	2	5	20
CLB19	0x00400594	1	0	0	2	6	20
CLB20	0x004005d4	1	0	0	2	7	20
CLB21	0x00400614	1	0	0	3	0	20
CLB22	0x00400654	1	0	0	3	1	20
CLB23	0x00400694	1	0	0	3	2	20
CLB24	0x004006d4	1	0	0	3	3	20
CLB25	0x00400714	1	0	0	3	4	20
CLB26	0x00400754	1	0	0	3	5	20
CLB27	0x00400794	1	0	0	3	6	20
CLB28	0x004007d4	1	0	0	3	7	20

5

	Frame Address	Top/Bottom Bit	Block Type	Row Address	Column Address	Unknown	Minor Address
CLB1	0x00404054	1	0	1	0	1	20
CLB2	0x00404094	1	0	1	0	2	20
CLB3	0x004040d4	1	0	1	0	3	20
CLB4	0x00404114	1	0	1	0	4	20
CLB5	0x00404154	1	0	1	0	5	20
CLB6	0x00404194	1	0	1	0	6	20
CLB7	0x004041d4	1	0	1	0	7	20
CLB8	0x00404214	1	0	1	1	0	20
CLB9	0x00404294	1	0	1	1	2	20
CLB10	0x004042d4	1	0	1	1	3	20
CLB11	0x00404314	1	0	1	1	4	20
CLB12	0x00404354	1	0	1	1	5	20
CLB13	0x00404394	1	0	1	1	6	20
CLB14	0x004043d4	1	0	1	1	7	20
CLB15	0x00404494	1	0	1	2	2	20
CLB16	0x004044d4	1	0	1	2	3	20
CLB17	0x00404514	1	0	1	2	4	20
CLB18	0x00404554	1	0	1	2	5	20
CLB19	0x00404594	1	0	1	2	6	20
CLB20	0x004045d4	1	0	1	2	7	20
CLB21	0x00404614	1	0	1	3	0	20
CLB22	0x00404654	1	0	1	3	1	20
CLB23	0x00404694	1	0	1	3	2	20
CLB24	0x004046d4	1	0	1	3	3	20
CLB25	0x00404714	1	0	1	3	4	20
CLB26	0x00404754	1	0	1	3	5	20
CLB27	0x00404794	1	0	1	3	6	20
CLB28	0x004047d4	1	0	1	3	7	20

6

	Frame Address	Top/Bottom Bit	Block Type	Row Address	Column Address	Unknown	Minor Address
CLB1	0x00408054	1	0	2	0	1	20
CLB2	0x00408094	1	0	2	0	2	20
CLB3	0x004080d4	1	0	2	0	3	20
CLB4	0x00408114	1	0	2	0	4	20
CLB5	0x00408154	1	0	2	0	5	20
CLB6	0x00408194	1	0	2	0	6	20
CLB7	0x004081d4	1	0	2	0	7	20
CLB8	0x00408214	1	0	2	1	0	20
CLB9	0x00408294	1	0	2	1	2	20
CLB10	0x004082d4	1	0	2	1	3	20
CLB11	0x00408314	1	0	2	1	4	20
CLB12	0x00408354	1	0	2	1	5	20
CLB13	0x00408394	1	0	2	1	6	20
CLB14	0x004083d4	1	0	2	1	7	20
CLB15	0x00408494	1	0	2	2	2	20
CLB16	0x004084d4	1	0	2	2	3	20
CLB17	0x00408514	1	0	2	2	4	20
CLB18	0x00408554	1	0	2	2	5	20
CLB19	0x00408594	1	0	2	2	6	20
CLB20	0x004085d4	1	0	2	2	7	20
CLB21	0x00408614	1	0	2	3	0	20
CLB22	0x00408654	1	0	2	3	1	20
CLB23	0x00408694	1	0	2	3	2	20
CLB24	0x004086d4	1	0	2	3	3	20
CLB25	0x00408714	1	0	2	3	4	20
CLB26	0x00408754	1	0	2	3	5	20
CLB27	0x00408794	1	0	2	3	6	20
CLB28	0x004087d4	1	0	2	3	7	20

Table 4.2: Frame address of all frames containing flip-flop bits

Flip-Flop Bit Locations		Column	Two Column
5	6	Difference	Difference
33	34	28	27
45	46	12	11
73	74	28	27
85	86	12	11
113	114	28	27
125	126	12	11
153	154	28	27
165	166	12	11
193	194	28	27
205	206	12	11
233	234	28	27
245	246	12	11
273	274	28	27
285	286	12	11
313	314	28	27
325	326	12	11
353	354	28	27
365	366	12	11
393	394	28	27
405	406	12	11
433	434	28	27
445	446	12	11
473	474	28	27
485	486	12	11
513	514	28	27
525	526	12	11
553	554	28	27
565	566	12	11
593	594	28	27
605	606	12	11
633	634	28	27
677	678	44	43
705	706	28	27
717	718	12	11
745	746	28	27
757	758	12	11
785	786	28	27
797	798	12	11
825	826	28	27
837	838	12	11
865	866	28	27
893	894	28	27
905	906	12	11
933	934	28	27
945	946	12	11
973	974	28	27
985	986	12	11
1013	1014	28	27
1025	1026	12	11
1053	1054	28	27
1065	1066	12	11
1093	1094	28	27
1105	1106	12	11
1133	1134	28	27
1145	1146	12	11
1173	1174	28	27
1185	1186	12	11
1213	1214	28	27
1225	1226	12	11
1253	1254	28	27
1265	1266	12	11
1293	1294	28	27
1305	1306	12	11

Table 4.3: Offset of flip-flop bits within each frame from Table 4.2

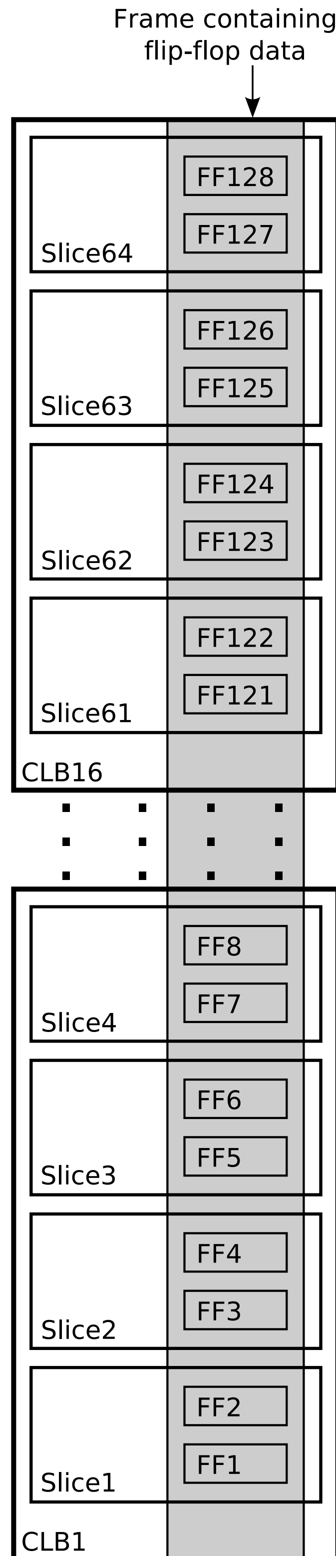

Figure 4.3: This figure illustrates a frame that contains flip-flops. It spans 16 CLBs in height and contains 128 flip-flops.

Address Type	Bit Index	Description
Top/Bottom Bit	22	Select Between top-half rows (0) and bottom-half rows (1).
Block Type	21:19	Block types are. CLB/IO/CLK (000), block RAM Interconnect (001), block RAM content (010).
Row Address	18:14	Selects a row of frames. Rows are 16 CLBs in height, with an HCLK in the middle. The row addresses increase away from the middle (in both top and bottom).
Column Address	13:9	Selects a column of CLBs. Column addresses start at 0 on the left and increase to the right.
Minor Address	5:0	Selects a memory-cell address line within a column.

Table 4.4: Frame Address Register Description

Next, a pattern is needed for the frames in Table 4.2. This information cannot easily be deduced using the frame addresses, because they do not necessarily increment by one when going from one block of 16 CLBs to the next. Instead, the bit offset within the bitstream from the logic allocation file is used. Using the offset from the last flip-flop bit in a frame to the first flip-flop bit in the next frame, the number of bits residing between the two flip-flop can be determined. From [52] it is known that the number of bytes in a frame is 164. Thus, the number of frames per 16 CLBs can be determined as given in Equation 4.1.

$$\text{number of frames per 16 CLB block} = \frac{\text{CLB}_n\text{, flip-flop}_1 \text{ bit offset} - \text{CLB}_{n-1}\text{, flip-flop}_{128} \text{ bit offset}}{164 * 8} \tag{4.1}$$

By assigning a simple numeric count to each frame, Table 4.5 is created. It can be deduced from Table 4.5 that for the V4LX25 there are 22 frames in-between each frame containing flip-flops, except for two locations in which there are 43 and 55 frames respectively[1]. Therefore, it can be concluded that there are 22 frames per block of 16 CLBs.

The only element left in determining where the flip-flop bits reside is an initial condition. This condition is where the CLB configuration data begins. Because each frame address that contains flip-flop bits has a minor address of 20, and because it can be deduced that each

[1] The pattern of frames was also determined for the V4LX40 and V4LX60 devices. In these patterns, the number of exceptions (two) for the jumps in the frame addresses were no different than for the V4LX25. If more exceptions occur in larger devices, the size of the hardware needed to implement the dynamic data mask controller would increase.

	Section						
Column	1	2	3	4	5	6	Column Difference
X0/X1	50	782	1514	2246	2978	3710	
X2/X3	72	804	1536	2268	3000	3732	22
X4/X5	94	826	1558	2290	3022	3754	22
X6/X7	116	848	1580	2312	3044	3776	22
X8/X9	138	870	1602	2334	3066	3798	22
X10/X11	160	892	1624	2356	3088	3820	22
X12/X13	182	914	1646	2378	3110	3842	22
X14/X15	204	936	1668	2400	3132	3864	22
X16/X17	247	979	1711	2443	3175	3907	43
X18/X19	269	1001	1733	2465	3197	3929	22
X20/X21	291	1023	1755	2487	3219	3951	22
X22/X23	313	1045	1777	2509	3241	3973	22
X24/X25	335	1067	1799	2531	3263	3995	22
X26/X27	357	1089	1821	2553	3285	4017	22
X28/X29	412	1144	1876	2608	3340	4072	55
X30/X31	434	1166	1898	2630	3362	4094	22
X32/X33	456	1188	1920	2652	3384	4116	22
X34/X35	478	1210	1942	2674	3406	4138	22
X36/X37	500	1232	1964	2696	3428	4160	22
X38/X39	522	1254	1986	2718	3450	4182	22
X40/X41	544	1276	2008	2740	3472	4204	22
X42/X43	566	1298	2030	2762	3494	4226	22
X44/X45	588	1320	2052	2784	3516	4248	22
X46/X47	610	1342	2074	2806	3538	4270	22
X48/X49	632	1364	2096	2828	3560	4292	22
X50/X51	654	1386	2118	2850	3582	4314	22
X52/X53	676	1408	2140	2872	3604	4336	22
X54/X55	698	1430	2162	2894	3626	4358	22
Section Difference		83	83	83	83	83	

Table 4.5: Numeric representation of frames containing flip-flop bits

A	6022	Total number of configuration frames
B	3360	Total number of frames used in hash function (28 column pairs * 6 rows * 20 frames/ column-pair-rows)
B/A	55.80%	Percentage of frames used in hash function
C	336	Total number of frames ignored within the CLB section (28 column pairs * 6 rows * 2 frames/ column-pair-rows)
C/A	5.58%	Percentage of frames ignored within the CLB section
D=B+C	3696	Total number of frames in CLB section
D/A	61.37%	Percentage of frames in CLB section
E=A-(D)=A-(B+C)	2326	Total number of frames ignored outside CLB section
E/A	38.63%	Percentage of frames ignored outside of CLB section
F=C+E	2662	Total number of frames ignored
F/A	44.20%	Percentage of frames ignored

Table 4.6: Statistics of frames used and ignored by the configuration integrity checker

block of 16 CLBs contains 22 frames, it can be determined that the frame that contains flip-flops is the twenty-first frame in each block of 16 CLBs. The first frame address containing flip-flops is 0x54. Subtracting 20, or 0x14, from this value to get the first frame of CLB data, results in a frame address of 0x40. Therein, it is assumed that all frames between frame addresses 0x0 and 0x40 are not CLB data and must be ignored. Thus, the readback controller starts readback at frame address 0x40. Determining where CLB data ends is not as difficult, considering that the total number of 16 CLB blocks are known, as shown in Table 4.5.

Despite ignoring the flip-flops, there is still some dynamic data in the CLB section. This was determined by testing readback on the device with the flip-flop bits masked out and noting that the data is not static. To solve this problem, the number of frames that are readback per 16 CLB block are reduced from 22 to 20 by ignoring the first and last frame. In the end, 3,360 frames are used in the hash computation out of 6,022 total configuration frames. Table 4.6 summarizes this information.

4.2.5 MD5 Hash Function Controller

Choice of Hash Function – MD5

As stated in Chapter 3, because the partial authentication and challenge-response protocol both require a cryptographic hash function, the configuration integrity checker should use the same hash function to reduce resource utilization.

If the configuration integrity checker only needed to defend against Class 1 or Class 2 attacks, an error detection hash function such as a CRC calculation would probably suffice. These techniques provide protection against non-malicious error [58] which, in essence, is how Class 1 and Class 2 attacks manifest themselves.

For Class 3 attacks, more security is desired. A CRC or checksum calculation is not intended for cryptographic use because an attacker may be able to determine certain alterations that produce a desired output value [58]. Cryptographic hash functions are specifically designed to prevent undetectable intentional modification [58]. Because a Class 3 attacker could alter any desired bit and has significant knowledge of the design, the attacker may be able to produce a desired output.

The cryptographic hash chosen is MD5. MD5 is a commonly used hash function [58] that produces a 128-bit hash value. It is designed to be suitable for high-speed software applications by being based on a simple set of bit manipulations operating on 32-bit operands [61]. MD5 processes the input in 512-bit blocks. For a detailed description of the MD5 algorithm refer to [58] and [61].

To implement the MD5 algorithm, a core from Opencores.org is used [62]. The core receives the 512-bit inputs in four 128-bit blocks. Whenever the core must be reset so that the initial chaining variables of the MD5 algorithm are used, a `newtext` signal is asserted for one clock cycle. Until this signal is asserted, the MD5 core uses the previous hash results for the current computation. The core does not begin computation until an entire 512-bit block is given. Furthermore, it cannot receive a new block until it has finished the computation

for the current block.

Choice of Input Window and Checking Granularity

The input window of length n, shown in Figure 3.1, must be 512 bits because the MD5 algorithm takes inputs in 512-bit blocks. If the input data is smaller than 512 bits, padding can be used; however, this does not imply that a hash value should be computed for sections less than 512 bits. Calculating a hash value for every byte or word of configuration data would produce an impractical number of output values to manage. The smallest practical checking granularity would most likely be a frame because it is the smallest addressable unit in the configuration. The largest granularity would be the entire configuration. For this implementation, the checking granularity is chosen to be the 20 frames that are static in a block of 16 CLBs. This granularity is chosen because it produces a reasonable number of hash values to manage while still allowing flexibility for run-time repair considerations. This reasoning is further reinforced after examination in Section 5.5. Thus, the MD5 function will process 3,280 bytes per output. This corresponds to 52 512-bit blocks after padding is included in the 52nd block. Because there are 168 blocks of 16 CLBs, this granularity results in 168 128-bit hash values that must be stored and compared.

Controller Operation

A flow chart of the MD5 controller state machine is given in Figure 4.4. The controller buffers 16 bytes of data for the 128-bit input to the MD5 core. The data is a byte of configuration data from the readback controller masked with a byte from the dynamic data mask controller. When the MD5 core is ready and is not performing any computations, the 128-bit input is loaded into the MD5 core. After the 20 frames in a block of 16 CLBs that are going to be computed are sent to the MD5 core, the controller waits for the final hash output and then sends this result to the hash comparator.

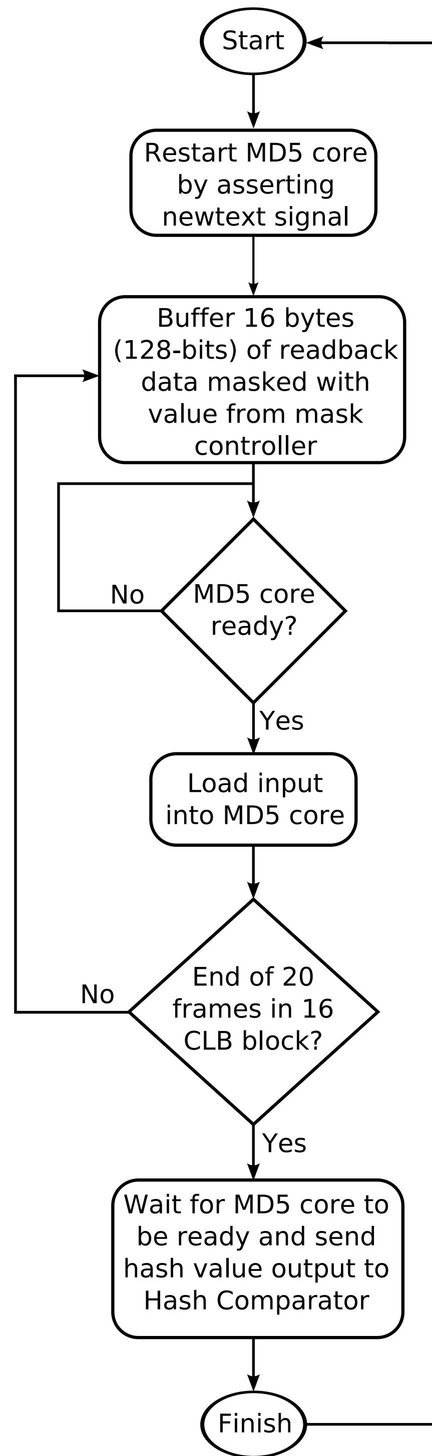

Figure 4.4: Flow Chart of MD5 Hash Function Controller State Machine

4.2.6 Hash Comparator

The hash comparator component checks to see if the hash value computed has changed from the trusted value. To obtain the initial values necessary for comparison, another assumption is made for this implementation. It is assumed that the first full configuration of the FPGA and the first scan of the configuration is safe and the checker can build a database of trusted hash values. Other methods of obtaining the initial hash values are discussed in Section 6.2. The checker also calculates the key for the partial authenticator and the verifier of the challenge-response protocol.

If the calculated hash value is different from the stored hash value, the checker raises a failure flag and outputs the number of the failed section. The numbering scheme starts at zero for the first block of 16 CLBs and increments by one for a total of 168 blocks.

4.3 Partial Authentication and Reconfiguration

4.3.1 Xilinx Encryption

Before discussing the implementation of the partial authenticator, the reasons for not using the Virtex-4 encryption capabilities should be stated. The Virtex-4 offers AES encryption of the bitstream, as was mentioned in Section 2.4.2. The encryption is applied during the bitstream generation process using BitGen. As was described in Chapter 2, when using Xilinx's encryption method, readback and partial reconfiguration are not allowed on Virtex-4 LX FPGAs. These requirements are necessary to the system.

4.3.2 Authentication Method

As was stated in Section 3.3.2, the method of choice for partial authentication is a hash signature with a secret key prepended to the input message. For a high level diagram of this protocol see Figure 3.4. Because MD5 is used in the configuration integrity checker, it is also

used in the partial authenticator to reduce the amount of hardware consumed. The secret key is the MD5 hash value of all other hash values computed by the configuration integrity checker. The secret key is assumed to be an initial condition that can be calculated and sent off-chip to the entity that will be delivering the partial bitstreams to the FPGA. In this implementation, this process is done by hand; however, other methods of achieving this key distribution are discussed in Section 6.2.

4.3.3 Software Preparation of Partial Bitstream

Before sending a partial bitstream to the board, it must be prepared for the authentication process. A software program has been created to execute the preparation procedure and is illustrated in Figure 4.5. The program first reads in the partial bitstream and prepends it with the secret key. The program then computes the MD5 hash value of the bitstream and key. Next, the partial bitstream is prepended with the computed hash value. The program then appends the necessary padding to the end of the partial bitstream as required by the MD5 algorithm. Finally, the bitstream is prepended with its size in bytes. The size includes the length of the padding, but not the prepended hash value. The bitstream is then ready to be sent over an unsecured channel to the partial authenticator on the FPGA. The padding is applied in software to reduce the resources the hardware authenticator would consume if it were to perform the padding.

4.3.4 Authentication Hardware

Before reconfiguring the FPGA with the partial bitstream, the bitstream must be authenticated by verifying its source and its integrity. An illustration of the authentication hardware is shown in Figure 4.6. The authentication hardware first reads in the hash value that has been prepended to the partial bitstream and stores it for comparison. It also reads in the size of the partial bitstream, including the padding, so that it knows how many bytes must be input into the hash function. It then reads in the partial bitstream and inputs it into the

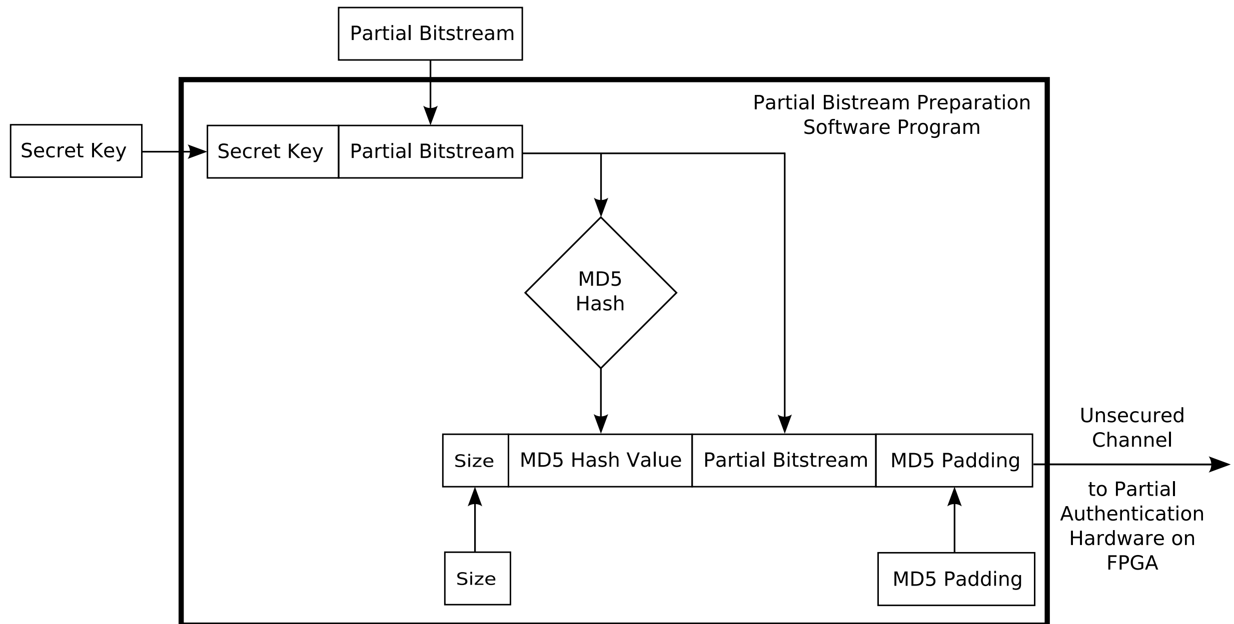

Figure 4.5: Illustration of partial bitstream preparation software

MD5 core in 16 byte segments. The padding does not need to be calculated in hardware, because it was precomputed in software. After the entire bitstream has been hashed, the value is taken and compared to the original hash value sent with the bitstream. If the hash values are the same, the authentication state machine proceeds to reconfigure the device.

4.3.5 Partial Reconfiguration

After authenticating the partial bitstream, the state machine reconfigures the FPGA. As with readback, reconfiguration is accomplished using the ICAP. The partial bitstream contains all commands needed for reconfiguration. Thus, the state machine does not need to first send the ICAP a stored set of initial instructions. Instead, it just clocks each value of the partial bitstream into the ICAP.

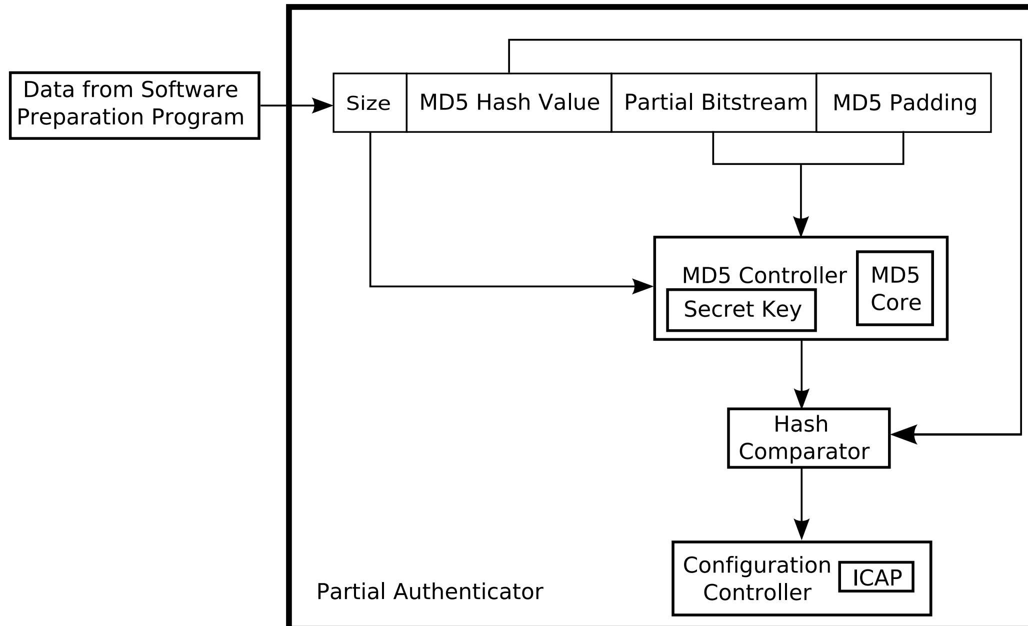

Figure 4.6: Illustration of partial authentication hardware

4.4 Challenge-Response Protocol

4.4.1 Challenge from Verifier

The challenge-response protocol, illustrated in Figure 3.5, is implemented as proposed in Section 3.3.3. The verifier sends the security system a random message and the size of the random message, as shown in Figure 4.7. Also included in the message is any padding necessary for the MD5 algorithm. This method is chosen to help the message be resistant to repetition because the message size does not have to be repeated. The padding is included to reduce hardware utilization in the claimant implementation.

As stated in Section 3.3.3, the verifier should be an entity external to the FPGA. Though this entity is not implemented, it is assumed that it would be impervious to the attacks outlined in Chapter 3. The entity could be an ASIC because the hardware layout is static and not susceptible to fault injection. The entity could also be separate from the system and challenges could be issued remotely so that the attacker does not have physical access

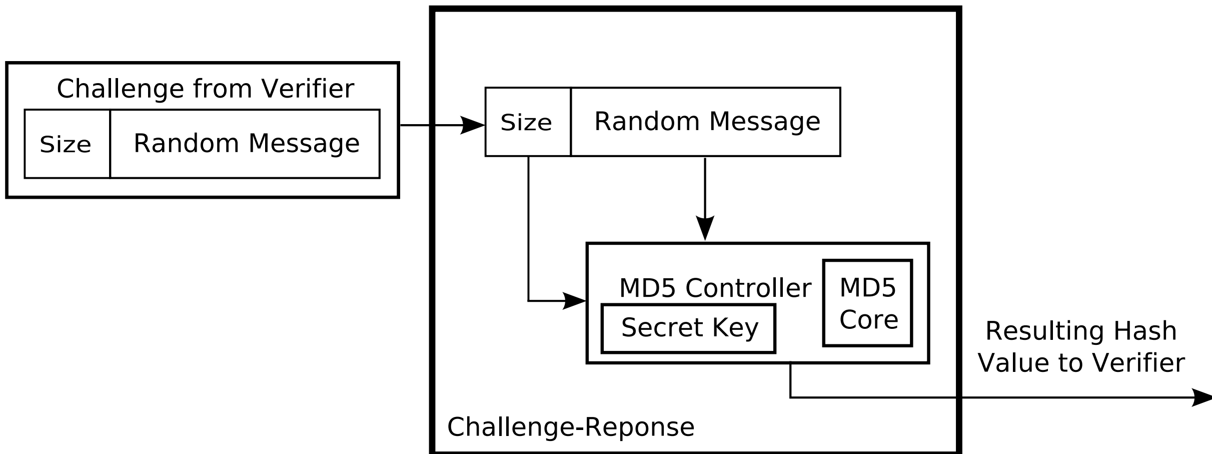

Figure 4.7: Illustration of challenge-response hardware

to the entity issuing the queries.

4.4.2 Claimant Response

The security system acts as the claimant in the challenge-response protocol. As illustrated in Figure 4.7, the hardware first reads the length of the message. It then computes the MD5 hash of the message, padding, and secret key. The calculated hash value is then sent back to the verifier. The secret key is comprised of the hash of all hash values computed by the checker after each scan of the configuration. If the configuration ever changes, the key will also change. Thus, the only time the claimant can be truly verified is when the configuration has not been maliciously altered. Otherwise, the verifier will get an incorrect hash value as it is using the key computed from the initial conditions of the hash values. If the design being protected requires partial reconfiguration in order to operate, then a new key will be computed from new trusted hash values. Similar to the initial condition already stated, it is assumed that the verifier would be able to generate the new key as well. Should the verifier get the incorrect hash value or not get a hash value at all, then it is assumed that the security system has been compromised.

Chapter 5

Results and Analysis

This chapter presents the results and analysis of the system implemented in Chapter 4. First, a description of how the operation of the system was verified is provided. Then, issues relating to device utilization are discussed. Next, a timing analysis and security analysis of the system are given. Lastly, a projected performance and analysis of how run-time repair could be included with the system is discussed.

5.1 Operation Verification

To test how the system responds to malicious alterations of the configuration, partial bitstreams were created to simulate a fault injection attack. The partial bitstreams were created using the difference-based flow provided by Xilinx, as discussed in Section 2.2.2. The partial bitstreams made alterations to the configuration that would affect the operation of a protected design. The goal of the experiment was to ensure that the configuration integrity checker detects changes in the configuration.

The protected design was an implementation of the Lucas-Lehmer test for Mersenne numbers. The Lucas-Lehmer test is an efficient deterministic primality test for determining if a Mersenne number M_n is prime [63]. A Mersenne number has the form $M_n = 2^n - 1$. The test uses the recurrence equation $S_n = \left(S_{n-1}^2 - 2\right) \bmod M_n$ with $S_0 = 4$. M_n is prime if

and only if $S_{n-2} = 0$. The value S_{n-2} is called the residue for n. For example, the sequence obtained for $n = 7$ is given by 4, 14, 67, 42, 111, 0, so $M_7 = 127$ is a prime. The protected design tested for $n = 3, 4, 5, 6,$ and 7, which was defined by a 3-bit input on the board's dip switches. The design calculated all S_0 through S_{n-2} and determined if M_n was a Mersenne prime. As illustrated in Figure 5.1, if M_n was a Mersenne prime, a `yes` signal was set, which was routed through a flip-flop to an LED. If M_n was not a Mersenne prime, a `no` signal was set, which was routed to another LED. The protected design resided in the upper left corner of the FPGA and used approximately 7% of the FPGA's resources by consuming 744 slices.

The first experiment swapped the routes from the `yes` and `no` signals to their corresponding LEDs, as illustrated in Figure 5.2. If the flags were being sent to an external module that made decisions based on these values, then an attacker would have switched the correspondence between inputs and outputs. The change only affected a total of 24 frames in four blocks of 16 CLBs. After reconfiguring the device with the partial bitstream, the checker successfully detected the changes in all four blocks. The second experiment involved the inversion of the `yes` and `no` signals, as illustrated in Figure 5.3. The effect on the LEDs is the same as for the previous experiment. An attacker would switch the input-output correspondence. The alterations only affect two frames, one frame in two blocks of 16 CLBs. To perform the inversion, only six bits of configuration data need to be changed for each inverter. The checker successfully detected the changes in both blocks of data. The third experiment involved changing the contents of a LUT in one of the slices for the computation of S_n. The change affected only one frame of configuration data. The checker detected the change for the block of 16 CLBs that contained the frame.

The three experiments validated the operation of the checker. Successful authentication by the partial authenticator was also verified during these experiments, because the partial bitstream had to be prepared and authenticated before reconfiguring the device. To prove that an altered partial bitstream would not be authenticated, one of the partial bitstreams was modified after the software preparation process. When attempting to reconfigure the FPGA, the authenticator determined that the hash value of the bitstream differed from the

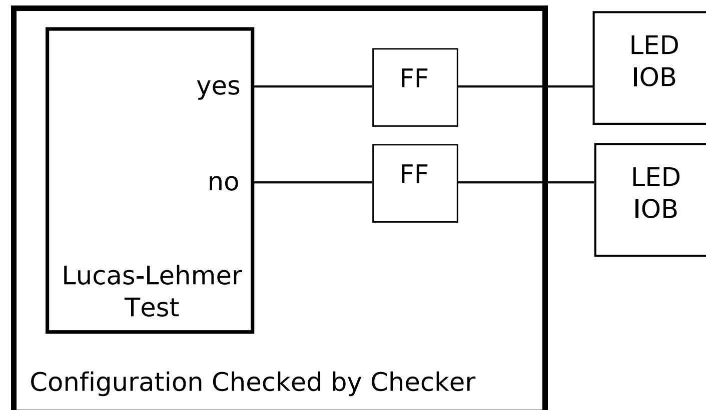

Figure 5.1: The design protected by the security system is an implementation of the Lucas-Lehmer test for Mersenne Primes.

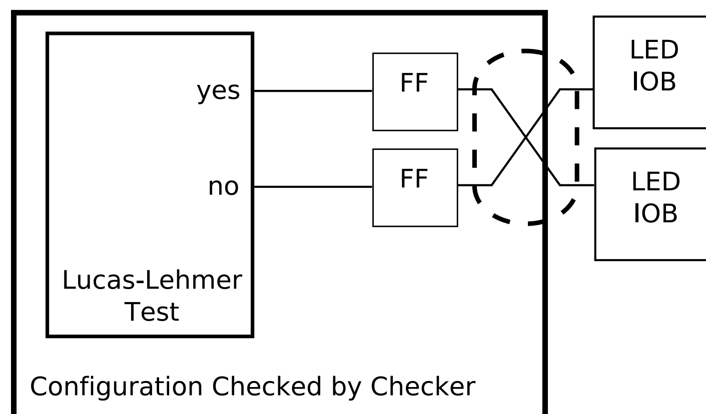

Figure 5.2: The first verification experiment swapped the routes to the two LEDs.

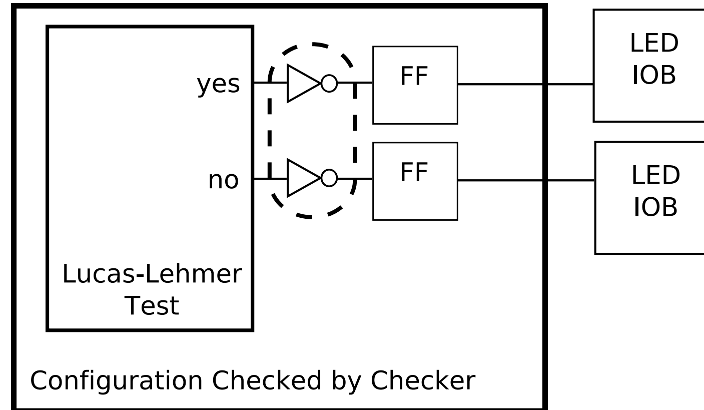

Figure 5.3: The second verification experiment inverted the outputs of the Lucas-Lehmer test.

hash value sent with it and did not reconfigure the FPGA. To verify the challenge-response protocol, challenge questions were submitted to the system. Each question was submitted continuously, and thus the system responded once for every run of the configuration integrity checker. After observing the response for each question, the hash value was correct because the configuration was not altered. When the configuration was altered by one of the partial bitstreams, the hash value was incorrect.

5.2 Device Utilization

A general overview of the resources used by the system as a whole are given in Table 5.1. As can be seen from Table 5.1, 69% of the hardware logic of the V4LX25 was used. Thus, the system would probably not be acceptable on a chip of this size, because only about 30% of the logic could be used for the actual design that is to be protected. The V4LX25 is the

Resource	Amount Used	Percentage
ICAPs	1 of 2	50%
RAMB16s	4 of 72	6%
Slices	7509 of 10752	69%

Table 5.1: Overview of the resources utilized by the system as a whole

Resource	Amt. Used	Percentage
ICAP	1 of 2	50%
RAMB16s	8 of 160	5%
Slices	7509 of 26624	28%

(a) V4LX60

Resource	Amt. Used	Percentage
ICAP	1 of 2	50%
RAMB16s	16 of 336	5%
Slices	7509 of 89088	8%

(b) V4LX200

Table 5.2: Overview of the resources utilized by the system on larger devices available in the Virtex-4 LX platform

second smallest part available in the Virtex-4 LX platform. If implemented on a mid-sized LX chip, such as the V4LX60, the resulting utilization would be more acceptable as shown in Table 5.2(a). This allows for over 70% of the hardware logic to be used for the protected design. On the largest chip, the V4LX200, there would be over 90% of the hardware logic resources available for the protected design as shown in Table 5.2(b).

By averaging the sequential logic and combinational logic usage percentages from the synthesis area usage report, the resources consumed by each individual system can be approximated. The result is the statistics shown in Figure 5.4. Due to the MD5 core being shared by each system, it has been separated out. If a different hash function were chosen, then this segment's percentage would change, and the size of the system as a whole would change accordingly. The largest segment is the checker system. Its size is due to the fact that not only is it a large FSM itself, but it also contains a large FSM for the readback controller and another FSM for the dynamic data mask controller.

5.3 Timing Analysis

5.3.1 Clock Frequency

The system was run at a clock frequency of 50MHz. This frequency was chosen to accommodate the operation of the MD5 core. The MD5 core contains a critical path at the point where it computes the output of a round in a single clock cycle. The diagram in Figure 5.5

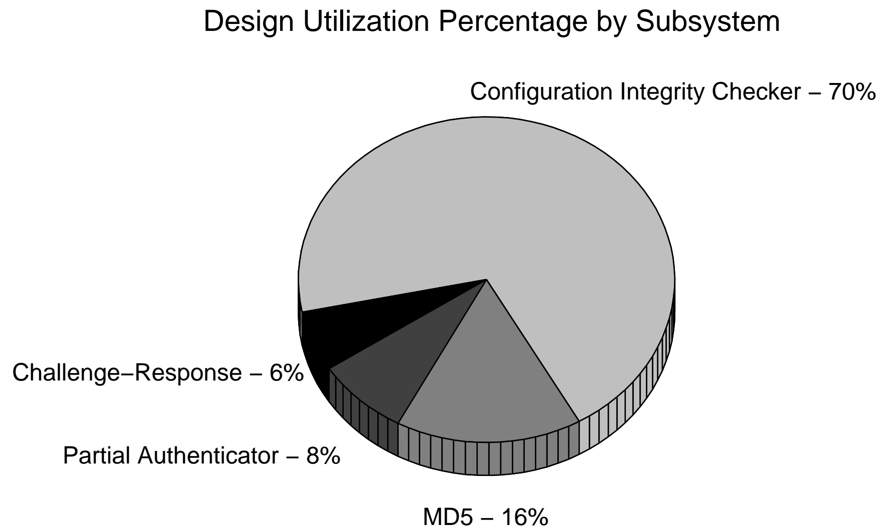

Figure 5.4: Resource utilization percentage of each subsystem in the security system

illustrates the operations that must execute in combinational logic.

To correct this problem, the MD5 core could be pipelined such that each of these functions would have its own clock cycle for computation. Though this would require more clock cycles for the MD5 core to finish generating a hash value, it would allow for the entire system to operate at a faster frequency and thus improve the overall timing.

5.3.2 Execution Time

The number of clock cycles necessary for each subsystem to fully complete its task was computed. Each subsystem is individually analyzed in this section.

Configuration Integrity Checker

For the checker, the number of clock cycles necessary to compute the hash value of a single block of data was calculated. Also, the number of cycles required to skip a single frame was determined. These values were used to extrapolate the number of clock cycles required

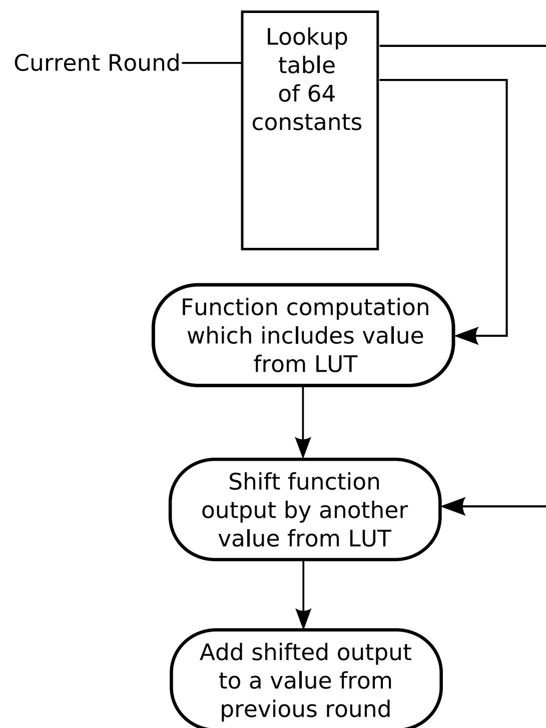

Figure 5.5: Illustration of operations that must execute in the critical timing path of the MD5 core

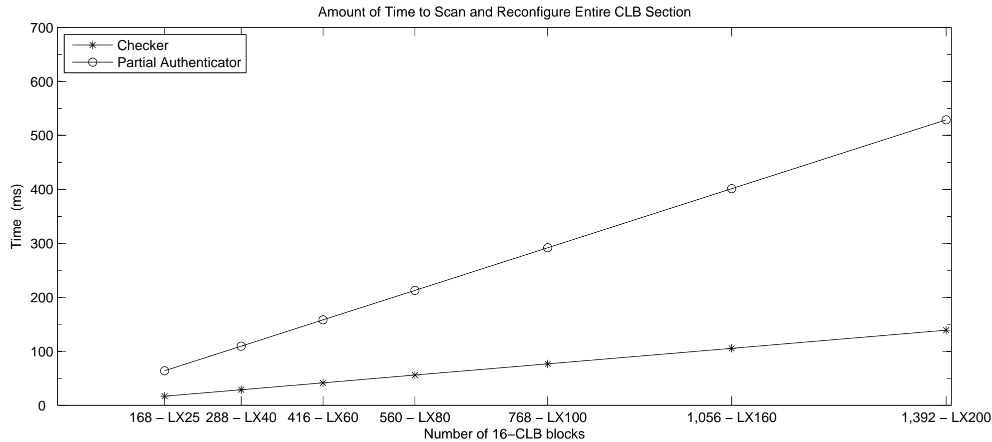

Figure 5.6: This figure presents a graph of the amount of time it takes for the configuration integrity checker to scan the entire CLB section and for the partial authenticator to reconfigure the entire CLB section for each part available for in the Virtex-4 LX platform.

to hash all of the blocks and the total number of frames skipped, respectively. Each block took 4,006 clock cycles to compute. Thus, all 168 blocks took a total of 673,008 clock cycles to finish. At 50MHz, this corresponds to 80.12μs per block and 13.46ms for all 168 blocks. Each frame that was skipped between blocks took 492 clock cycles or 9.84μs to complete. Recalling from Table 4.6 that 336 frames were skipped, the total number of clock cycles required to skip all frames was 165,312, which at 50MHz is 3.306ms. Therefore, the time it took to scan the entire CLB section was 838,320 clock cycles or 16.77ms at 50MHz. The time it takes to scan the entire CLB section for each part available in the Virtex-4 LX platform is given in Figure 5.6.

As a bench mark for the speed of operation, if the MD5 hash function is replaced with only an XOR operation, the system could run at 122MHz. At this frequency, the checker would only take 6.87ms to scan the entire configuration. If the MD5 core could be made to operate at 122MHz, the system would be close to this bench mark.

Partial Authenticator

The partial authenticator's execution time is dependent upon the size of the partial bitstream. The number of clock cycles required for the authentication process as a function of the number of bytes in a partial bitstream is given in Equation 5.1.

$$\text{number of clock cycles for authentication } = 1070 + 209 \times \left(\frac{numbytes}{64}\right) \tag{5.1}$$

The constants correspond to the number of clock cycles required for the authenticator's FSM to operate. The 64 in the divisor converts the number bytes in the partial bitstream, shown as *numbytes*, to the number of 512-bit blocks in the partial bitstream. If the partial bitstream is authenticated, the number of clock cycles required to feed the partial bitstream to the ICAP is shown in Equation 5.2.

$$\text{number of clock cycles to reconfigure } = 2 \times numbytes \tag{5.2}$$

After determining the number of bytes in the partial bitstream, multiplying the computed number of clock cycles by 1/50MHz, or 20ns, would give the number of seconds required for execution as shown in Equations 5.3 and 5.4.

$$\text{execution time if authenticated } = \left[1070 + 337 \times \left(\frac{numbytes}{64}\right)\right] \times 20ns \tag{5.3}$$

$$\text{execution time if not authenticated } = \left[1070 + 209 \times \left(\frac{numbytes}{64}\right)\right] \times 20ns \tag{5.4}$$

The size of the three partial bitstreams used in the verification tests were 7,474 bytes, 1,250 bytes, and 906 bytes. The execution times of the partial authenticator given these three partial bitstreams were 812μs, 157μs, and 125μs respectively. The time it would take to reconfigure the entire CLB section for each part available in the Virtex-4 platform is given in Figure 5.6. Also, using the bench mark of an XOR operation and running the system at 122MHz, the three partials would take 332μs, 63.8μs, and 50.7μs respectively.

Challenge-Response Protocol

The execution time of the challenge-response protocol is dependent upon the size of the challenge question. The total number of clock cycles required for the response process is given in Equation 5.5.

$$\text{number of clock cycles } = 943 + 209 \times \left(\frac{numbytes}{64} \right) \tag{5.5}$$

The constants correspond to the number of clock cycles required for the challenge-response's FSM to operate. The 64 in the divisor converts the number bytes in the query, shown as *numbytes*, to the number of 512-bit blocks in the query. After the number of bytes in the question is known, the number of clock cycles can be multiplied by 20ns to determine the execution time as shown in Equation 5.6.

$$\text{execution time} = \left[943 + 209 \times \left(\frac{numbytes}{64} \right) \right] \times 20ns \tag{5.6}$$

A query size of 512 bytes was used. A query of this size causes the challenge-response to take 52.3μs to execute. Using the bench mark of an XOR operation and running the system at 122MHz, a query of this size would only take the challenge-response system 20.9μs to operate.

5.4 Security Analysis

Having verified the implementation of the system, the extent to which it protects the integrity of the configuration against the three classes of attacks outlined in Section 3.2 can be determined. Also, the defense strength, as was discussed in Section 2.4.1, can be classified.

5.4.1 MD5 Collision Considerations — Brute Force Attack

The MD5 algorithm is susceptible to collisions, as are all hash functions. A collision occurs when two different messages hash to the same value [61]. Without considering the presence

of an attacker, there is a chance that two random messages hash to the same value. The Birthday Attack, resulting from the Birthday Paradox, states that for for an n-bit hash function, only $2^{\frac{n}{2}}$ hashes must be calculated before there is more than a 50% chance that a collision is found [58].

On the other hand, in this system, the attacker is given an initial message and an initial hash value. MD5 is a widely accepted hash function that is secure in its one-wayness [58]. In other words, it is difficult to reconstruct the input based upon the output. Thus, to find a second preimage of the hash value for a given message, the attacker has to resort to an exhaustive search where 2^{128} alterations are possible. The exponent value of 128 correlates to the size of the hash value which is 128-bits. The probability of an attacker finding a second preimage is given in Equation 5.7.

$$p(n) = 1 - \left(\frac{2^{128} - 1}{2^{128}}\right)^{n} \tag{5.7}$$

Solving Equation 5.7 for the number of iterations n with a probability of 50% results in 2.36×10^{38} iterations. The amount of time it would take an attacker to achieve this number iterations is given in Equation 5.8.

$$\text{time to achieve brute force attack } = \left(2.36 \times 10^{38}\right) * t \tag{5.8}$$

If an attacker is exhausting all possibilities by making alterations using fault injection and then observing the changes from the configuration integrity checker's computations, t would be equal to 16.77ms as was derived in Section 5.3.2. Consequently, the amount of time it would take the attacker to find a second preimage is 1.25×10^{29} years. This is an extremely long time and it can be assumed that the system would no longer be in use by the time the attacker would find a second preimage. Referring to the classes of attackers defined in Section 3.2, a Class 3 attacker has the highest likelihood of achieving a brute-force attack on the system. Unlike Class 1 and Class 2 attackers, the Class 3 attacker can control the attacks well enough not to replicate an alteration. Still, unless a Class 3 attacker is on an extremely low budget, it should be assumed that this method will not be used for a brute force attack.

Suppose an attacker has a budget exceeding $10 million. From [64] it is explained that such a budget could buy a machine that has found an MD5 collision using the Birthday Attack in as little as 24 days. Using this as a computational benchmark, suppose the attacker can afford a machine of this speed for a brute force attack on the security system. Further, suppose the attacker is of Class 3 and, using the observational attacks given in Section 3.1, has reproduced the entire design at the bit-level. This reproduction has allowed the attacker to reverse-engineer the design. Thus, the attacker knows how all aspects of the design operate, including the security system. With this knowledge the attacker has determined what hash function is being used and what frames are being checked. Now the attacker can make changes to the design and observe the changes in the attacker's own supercomputing system, which relieves the attacker from the time constraints of the security system. The amount of time it would take to find a second preimage is 8.41×10^{17} years.

5.4.2 Design-Based Attacks

The amount of time it would take for a brute-force attack makes it an unlikely choice for an attacker and it would behoove a Class 3 attacker to attempt a different strategy. Attacks against the security system's design could compromise the system such that it would not be aware that the configuration has been modified. Due to the fact that a Class 3 attacker has knowledge of the design at the bit-level, it may be possible for the attacker to find important registers in the design that correspond to the FSM for each system. Furthermore, the attacker could also learn the details of the state numbering scheme and what operations occur for each state. Such knowledge would allow the attacker to stop the configuration integrity checker from operating by forcing it to an unknown state. Then, alterations to the configuration could be made without changing the hash values stored. Moreover, if the attacker also controlled the partial authenticator FSM, malicious partial bitstreams could be loaded by bypassing the authentication step. The challenge-response protocol would provide some resistance because it acts in response to the checker operating successfully; however, because the attacker knows the state numbering scheme, the attacker could force

the challenge-response system to operate. The attacker would also have to stop the challenge-response system from erasing the BRAM of stored hash values used to create the key. The key would prove to be correct because the checker is not operating and cannot change the hash values. The challenge-response system would then successfully respond and the system would appear as though it has not been compromised.

Another vulnerable point in the system is the BRAMs used to store the hash values computed by the configuration integrity checker. As was stated in Section 3.1, the BRAMs are not protected because they contain dynamic data. If an attacker modifies the configuration, the attacker could attack both the failure flag and the block failure number sent from the configuration integrity checker to prevent any corrections to the configuration from occurring. The attacker could also modify the BRAMs used to store the hash values that the challenge-response key is computed from to make it the correct hash value. Also, the attacker could store partial bitstreams sent to change the configuration in BRAM and compute the correct hash values from the partial bitstreams.

The system is also exposed between scans of the configuration integrity checker. At the speed of operation determined in Section 5.3.2 of 16.77ms, any bit of configuration data is vulnerable to attack for 16.77ms after it is scanned. If the operation of the design being protected is sensitive to attacks that may occur in this time window, then the attacker could successfully affect the design's operation without having to compromise the security system.

5.4.3 Defense Strength

From this security analysis, the defense strength of the system can be classified. Class 1 and Class 2 attackers would not be able to achieve either a brute force or design-based attack of the system. As outlined in Section 3.2, a Class 1 attacker at best has strong observation capability or attack capability, but not both. These deficiencies hinder a Class 1 attacker from having an effective attack and the modifications would appear similar to SEUs as opposed to malicious alterations. Such an attack does not have a high probability

of creating a collision based on the discussion in 5.4.1. The attacker also would not be able to succeed at the design-based attack. As discussed in Section 3.2, a Class 2 attacker can target a specific area of the FPGA; however, does not have either the observation capability or attack capability to succeed at targeting specific bits. Such an attacker also does not have a high probability of succeeding at a brute for attack or a design-based attack. It can therefore be concluded that the system can defend against both Class 1 and Class 2 attackers.

A Class 3 attacker might be successful at a design-based attack; however, doing so would require some expertise. An attack on the design would require great knowledge of the bitstream composition if the attacker reconstructed the design from a collection of run-time repair partial bitstreams as discussed in Section 3.1. Furthermore, to force the FSMs to certain states, the attacker would have to make multiple alterations every clock cycle. This would require either a fault injection device that could flash multiple times every 20ns or multiple injection devices. Moreover, the devices would have to be precise enough to attack within the 20ns clock cycle window. If the frequency of operation was increased, all design-based attacks would be even more difficult.

These results can be used to classify the attacker and the defense strength of the system using IBM's model, as discussed in Section 2.4.1. A Class 3 attacker that could accomplish the design-based attack would be a "knowledgeable insider", because substantial knowledge of the system would be required as well as some specialized tools. The defense level of the system would be "MOD", again, because of the specialized knowledge and tools required to compromise the system.

5.5 Run-time Repair and Checking Granularity Considerations

As has been stated throughout this thesis, the security system discussed is not only intended to signal if the configuration has been altered, but is also intended to aid in the implemen-

tation of a system that repairs the configuration. Should the configuration integrity checker find a malicious change in the configuration, it can notify a system that obtains the correct partial bitstreams for the altered area. The size of these partial bitstreams and the time it takes to reconfigure depends upon the checking granularity chosen for the checker.

As was stated in Section 4.2.5, the smallest practical checking granularity would be a frame because it is the smallest addressable unit in the FPGA's bitstream architecture. This granularity was not chosen in the implementation because of how many output values it would produce (3,360); however, with run-time repair, such a granularity would allow for fast reconfiguration times because only the frames that were altered would be reconfigured. On the other extreme, a checking granularity of the entire configuration would allow for minimal storage space, but not for quick run-time repair times. In this scenerio, the entire configuration would have to be reprogrammed should an alteration occur. Thus, a space-time tradeoff can be inferred where an appropriate tradeoff between the two should be selected. A graph of the number of hash outputs from the checker versus the time to reconfigure is given in Figure 5.7. The graph is only plotted for a window of 300μs versus 200kbit. As can be seen on the graph, there is a knee where an intermediate value between the extremes can be found.

In this design, it was decided to calculate a hash value for every 20 frames in a block of 16 CLBs for a total of 168 hashes. This point is indicated on the graph in Figure 5.7. If any alterations occur, a minimum of 20 frames would be required to repair the corruption. This decision is reasonable because the run-time repair would have a minimum resolution of a CLB.

Reconfiguring a block of 16 CLBs takes 401μs. This time reinforces the acceptability of the granularity as it only takes 2.3% of the amount of time the checker takes to detect an alteration. The 168 hash values only consume about 21.5kbit of storage space on the chip. As can be seen from Table 5.1, this uses a small percentage of the storage space available and reinforces the chosen granularity from a space utilization perspective.

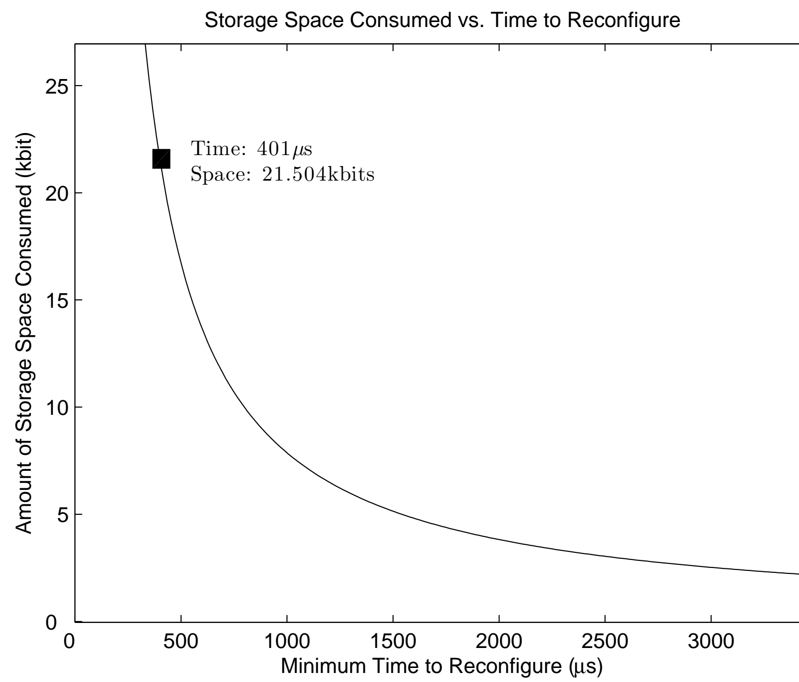

Figure 5.7: This figure presents a graph of storage space versus the time to reconfigure. The point at (401μs, 21.504kbits) indicates the checking granularity chosen.

Chapter 6

Conclusion

6.1 Summary

The primary goal of this thesis was to develop methods for securing the integrity of an executing FPGA configuration as well as reconfigured versions of it. These methods were all achieved on-chip. A system was designed and implemented that consists of three parts. An illustration of the system is provided in Figure 6.1. The first part of the system consists of a configuration integrity checker that monitors the configuration for malicious alterations that could be caused by fault injection or by altering incoming partial bitstreams. The system retrieves the configuration data via readback through the Internal Configuration Access Port (ICAP) and successfully masks out dynamic data to only monitor the static part of the configuration. An MD5 core is used to create a digital "fingerprint" for blocks of the configuration such that if a bit of configuration data is altered, the hash value is changed and the checker will know that the design has been corrupted. The second part of the system involves a partial authenticator that ensures that incoming partial bitstreams are trusted. These partial bitstreams could be reconfiguring the FPGA for another functionality or repairing the configuration from malicious alterations detected by the configuration integrity checker. The third part of the system pertains to a challenge-response protocol that would allow an entity external to the FPGA to verify that the security system has not been compro-

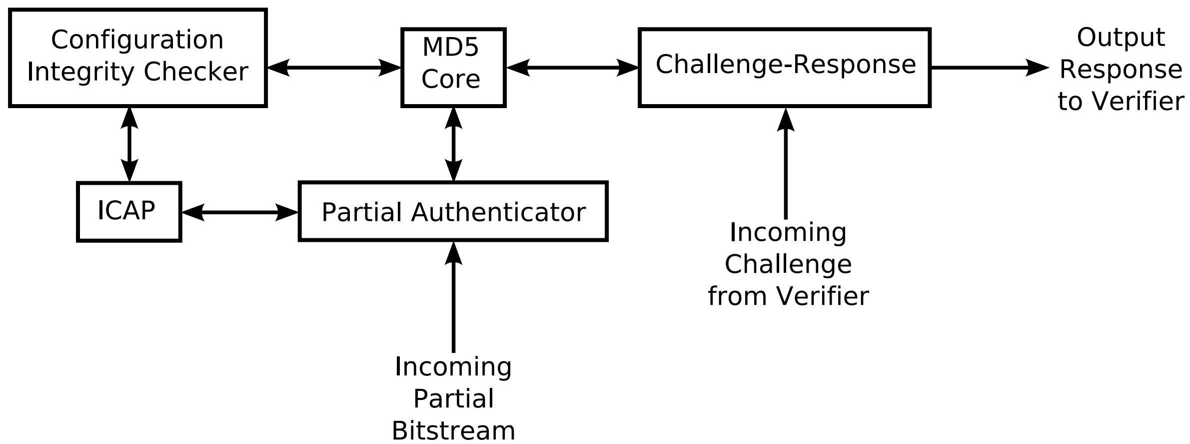

Figure 6.1: Block Diagram of Security System

mised. Both the partial authenticator and the challenge-response protocol utilize the same MD5 core used by the configuration integrity checker and use a secret key that is comprised of the hashes of the configuration.

The system was implemented on a Virtex-4 LX25 FPGA and consumed 69% of the available slices. The system would probably not be acceptable on a chip of this size, because only about 30% of the logic could be used for the actual design that is to be protected. The V4LX25 is the second smallest part available in the Virtex-4 LX platform. If implemented on a mid-sized LX chip, such as the V4LX60, the resulting utilization would be more acceptable, allowing 70% of the hardware logic to be used for the protected design. On the largest chip, the V4LX200, there would be over 90% of the hardware logic resources available for the protected design. The system was set to operate at a frequency of 50MHz, but with some modifications to the MD5 hardware, the system could run faster. The system proved to successfully detect malicious alterations through partial bitstreams that were created to simulate a fault injection attack. Furthermore, the operations of the partial authenticator and challenge-response protocol were verified using both trusted and non-trusted partial bitstreams and challenge queries, respectively.

Through a security analysis, three classes of attackers were defined. Each class increased

the attack capability and the observation capability of a malicious adversary. The observation capability affects an attacker's ability to gain knowledge of the design. By collecting partial bitstreams sent to repair the device or using an active photon probing technique, the attacker can reverse-engineer both the protected design and the security system. The more an attacker can learn about the design, the more effective the alterations the attacker can make. The attack capability depends on the attacker's ability to target specific bits in the FPGA configuration architecture. The effectiveness of an attack decreases if the attacker cannot target particular bits for modification. A Class 1 attacker was defined as having either no observation capability or weak attack strength. A Class 2 attacker was defined as being restricted to only observing a specific area of the configuration or only being able to attack a specific area. A Class 3 attacker was defined as having enough observation capability to gain knowledge of the design at the bit-level and enough attack capability to target any desired bit for alteration.

The system was shown to be secure against both Class 1 and Class 2 attackers. The fact that a Class 3 attacker could be successful at specific attacks on the security system was discussed. Such an attacker would need enough knowledge of bitstream composition to reverse-engineer the security system and would require enough attack capability to direct precise and frequent fault injections.

6.2 Future Work

There are a few areas where further work could improve the system. The first area of improvement involves the initial conditions necessary for the operation of the system. The initial trusted hash values as well as the key that is computed from these hash values for the partial authenticator and challenge-response protocol should be computed off-chip. Doing so would require masking out all flip-flop data in the full bitstream and computing the hash value. The location of the first frame of CLB data would need to be determined. The location could be ascertained by determining the number of frames that occur prior to the CLB

section. Another method involves creating bitstreams that only change the configuration in the first CLB and through comparison, finding the offset from the beginning of the bitstream. Also, any commands that occur with the configuration data would need to be taken into account. Some commands might specify that multiple frames are the same and thus, only the data for one frame is placed in the bitstream to reduce the bitstream size. Furthermore, if the system is to protect designs that will use dynamic partial run-time reconfiguration for operation, the keys necessary for the challenge-response protocol will need to be computed off-chip for the reconfigured versions of the design.

Another area of future work concerns the frequency of operation. As discussed in Section 5.3, the system is run at 50MHz due to a critical path in the MD5 core. The core could be pipelined so that more clock cycles are given for the operations in the critical path. Doing so would allow for the entire system to operate at a faster frequency and thus improve overall timing.

The system could also be made more parameterizable, allowing for it to be easily implemented on different platforms and possibly different families of Xilinx FPGA devices. This would involve finding the pattern of dynamic data in other chips by either building a database manually or creating a program that executed an algorithm of the steps described in Section 4.2.4. The algorithm can be created because the Logic Allocation File provides the numerical data necessary to determine a pattern.

Bibliography

[1] P. Graham and B. Nelson, "Genetic Algorithms In Software and In Hardware–A Performance Analysis of Workstation and Custom Computing Machine Implementations," in *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1996, pp. 216–225.

[2] J.-W. Jang, S. B. Choi, and V. K. Prasanna, "Energy- and Time-Effecient Matrix Multiplication on FPGAs," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 11, November 2005, pp. 1305–1319.

[3] M. Guo, M. O. Ahmad, M. N. S. Swamy, and C. Wang, "FPGA Design and Implementation of a Low-Power Systolic Array-Based Adaptive Viterbi Decoder," in *Proceedings of the 2003 International Symposium on Circuits and Systems*, vol. 2, May 2003, pp. II–276–II–279.

[4] S. Nisbet and S. A. Guccione, "The XC6200DS Development System," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, FPL 1997*, W. Luk and P. Y. K. Cheung, Eds. Berlin: Springer-Verlag, September 1997, pp. 61–68.

[5] P. Beckett, "A Fine-Grained Reconfigurable Logic Array Based on Double Gate Transistors," in *IEEE International Conference on Field-Programmable Technology*, December 2002, pp. 260–267.

[6] P. Lysaght and J. Dunlop, "Dyanmic Reconfiguration of FPGAs," in *More FPGAs*, W. Moore and W. Luk, Eds. Abingdon, England: Abingdon EE&CS Books, 1993, pp. 82–94.

[7] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. de Wit, "A Dynamic Reconfiguration Run-Time System," in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds. Los Alamitos, CA: IEEE Computer Society Press, April 1997, pp. 65–75.

[8] S. A. Guccione and D. Levi, "The Advantages of Run-Time Reconfiguration," in *Reconfigurable Technology: FPGAs for Computing and Applications, Proc. SPIE 3844*, J. Schewel, Ed. Bellingham, WA: SPIE–The International Society for Optical Engineering, September 1999, pp. 87–92.

[9] A. Thompson, "An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics," in *Proc. 1st Int. Conf. on Evolvable Systems (ICES'96)*, ser. LNCS, T. Higuchi, M. Iwata, and L. Weixin, Eds., vol. 1259. Springer-Verlag, 1997, pp. 390–405.

[10] G. McGregor, D. Robinson, and P. Lysaght, "A Hardware/Software Co-design Enviroment for Reconfigurable Logic Systems," in *Proceedind of FPL'98*, September 1998, pp. 258–267.

[11] T. S. Mohamed and W. Badawy, "Integrated Hardware-Software Platform for Image Processing Applications," in *IEEE International Workshop on System-on-Chip for Real-Time Applications*, July 2004, pp. 145–148.

[12] C. Ross and W. Bohm, "Using FIFOs in Hardware-Software Co-Design for FPGA Based Embedded Systems," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, April 2004, pp. 318–319.

[13] W. Luk and P. Y. Cheung, "Compilation Tools for Run-Time Reconfigurable Designs," in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and

J. Arnold, Eds. Los Alamitos, CA: IEEE Computer Society Press, April 1997, pp. 56–65.

[14] V. Kalenteridis, H. Pournara1, K. Siozios, K. Tatas, G. Koytroympezis, I. Pappas, S. Nikolaidis1, S.Siskos1, D. J. Soudris, and A. Thanailakis2, "An Integrated FPGA Design Framework: Custom Designed FPGA Platform and Application Mapping Toolset Development," in *Proceedings of the Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[15] P. Lysaght and J. Stockwood, "A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, September 1996, pp. 381–390.

[16] S. A. Guccione, "WebScope: A Circuit Debug Tool," in *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications*, R. W. Hartenstein and A. Keevallik, Eds. Berlin: Springer-Verlag, September 1998.

[17] J. N. Edmison, "Hardware Architectures for Software Security," PhD Dissertation, Virginia Polytechnic Institute and State University, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, June 2006.

[18] S. J. Harper, "A Secure Adaptive Network Processor," PhD Dissertation, Virginia Polytechnic Institute and State University, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, April 2003.

[19] J. P. Graff, "A Key Management Architecure for Securing Off-Chip Data Transfers on an FPGA," Masters Thesis, Virginia Polytechnic Institute and State University, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, June 2004.

[20] I. Hadžić, S. Udani, and J. M. Smith, "FPGA Viruses," in *FPL*, 1999, pp. 291–300.

[21] T. Kean, “Secure Configuration of a Field Programmable Gate Array,” in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01).* Washington, DC, USA: IEEE Computer Society, 2001, pp. 259–260.

[22] L. Bossuet, G. Gogniat, and W. Burleson, “Dynamically Configurable Security for SRAM FPGA Bitstreams,” in *18th International 2004 Proceedings of Parallel and Distributed Processing Symposium*, April 2004.

[23] A. S. Zeineddini and K. Gaj, “Secure Partial Reconfiguration of FPGAs,” in *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology*, December 2005, pp. 155–162.

[24] *Configuration Issues: Power-up, Volatility, Security, Battery Back-up (xapp092)*, Xilinx, November 1997, version 1.1.

[25] T. Wollinger, J. Guajardo, and C. Paar, “Security on FPGAs: State-of-the-Art Implementations and Attacks,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 3, pp. 534–574, 2004.

[26] S. P. Skorobogatov, “Semi-Invasive Attacks—A New Approach to Hardware Security Analysis,” University of Cambridge Computer Laboratory, Cambridge, United Kingdom, Technical Report, April 2005.

[27] N. Steiner, “A Standalone Wire Database for Routing and Tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs,” Masters Thesis, Virginia Polytechnic Institute and State University, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, August 2002.

[28] C. Morford, “BitMat - Bitstream Manipulation Tool for Xilinx FPGAs,” Masters Thesis, Virginia Polytechnic Institute and State University, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, December 2005.

[29] E. Sanchez, J.-O. Haenni, J.-L. Beuchat, A. Stauffer, and A. Perez-Uribe, "Static and Dynamic Configurable Systems," in *IEEE Transactions on Computers*, vol. 48, no. 6, June 1999.

[30] J. Liang, R. Tessier, and D. Goeckel, "A Dynamically-Reconfigurable, Power-Efficient Turbo Decoder," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, April 2004, pp. 91–100.

[31] E. Sanchez and A. Upegui, "Evolving Hardware by Dynamically Reconfiguring Xilinx FPGAs," in *ICES*, 2005, pp. 56–65.

[32] A. Upegui and E. Sanchez, "On-chip and On-line Self-Reconfigurable Adaptive Platform: the Non-Uniform Cellular Automata," in *2006 20th International Parallel and Distributed Processing Symposium*, April 2006.

[33] K. Paulsson, M. Hübner, M. Jung, and J. Becker, "Methods for Run-Time Failure Recognition and Recovery in Dynamic and Partial Reconfigurable Systems Based on Xilinx Virtex-II Pro FPGAs," in *Proceedings of the 2006 Emerging VLSI Technologies and Architectures (ISVLSI'06)*, March 2006.

[34] E. L. Horta and J. W. Lockwood, "PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs)," Department of Computer Science, Applied Research Lab, Washington University, Saint Louis, Technical Report, July 2001.

[35] E. Lechner and S. A. Guccione, "The Java Environment for Reconfigurable Computing," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, FPL 1997*, I. W. Luk and P. Y. K. Cheung, Eds. Berlin: Springer-Verlag, September 1997, pp. 284–293.

[36] M. Dyer, C. Plessl, and M. Platzner, "Partially Reconfigurable Cores for Xilinx Virtex," in *Reconfigurable Computing Is Going Mainstream.* 12th International Conference on Field-Programmable Logic and Applications, September 2002, pp. 292–301.

[37] *Virtex Series Configuration Architecture (xapp151)*, Xilinx, October 2004, version 1.7.

[38] S. McMillan and S. A. Guccione, "Partial Run-Time Reconfiguration Using JRTR," in *Field-Programmable Logic and Applications, Lecture Notes in Computer Science*, R. W. Hartenstein and H. Grunbacher, Eds., vol. 1896. Berlin: Springer-Verlag, August 2000, pp. 352–360.

[39] S. A. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-based Interface for Reconfigurable Computing," in *Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 2000.

[40] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour, "Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration," in *Design Automation Conference (DAC)*, New Orleans, LA, June 2002.

[41] *Two Flows for Partial Reconfiguration: Module Based or Difference Based (xapp290)*, Xilinx, September 2004, version 1.2.

[42] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular Dynamic Reconfiguration in Virtex FGPAs," in *IEEE Proceedings – Computers and Digital Techniques*, vol. 153, May 2006, pp. 157–164.

[43] F. Berthelot and F. Nouvel, "Partial and Dynamic Reconfiguration of FPGAs: A Top Down Design Methodology for an Automatic Implementation," in *2006 IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, March 2006.

[44] *FPGA Editor Guide*, Xilinx, 1999, version 2.1i.

[45] *Development System Reference Guide*, Xilinx, December 2005.

[46] R. Fong, S. Harper, and P. Athanas, "A Versatile Framework for FPGA Field Updates: An Application of Partial Self-Reconfiguration," in *2003 Proceedings of the 14th IEEE International Workshop on Rapid Systems Prototyping*, September 2003, pp. 117–123.

[47] H. Kalte, G. Lee, M. Porrmann, and U. Rückert, "REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems," in *2005 19th IEEE International Parallel and Distributed Processing Symposium*, April 2005, pp. 151–158.

[48] N. Steiner and P. Athanas, "An Alternate Wire Database for Xilinx FPGAs," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)*, pp. 336–337.

[49] D. G. Abraham, G. M. Dolan, G. P. Double, and J. V. Stevens, "Transaction Security System," *IBM Systems Journal*, vol. 30, no. 2, pp. 206–229, 1991.

[50] *Virtex-II User Guide. Xilinx User Guide 2*, Xilinx, March 2005.

[51] *Virtex-II Pro and Virtex-II Pro X User Guide. Xilinx User Guide 12*, Xilinx, March 2005.

[52] *Virtex-4 Configuration Guide. Xilinx User Guide 71*, Xilinx, January 2006.

[53] *Virtex-5 Configuration User Guide. Xilinx User Guide 191*, Xilinx, May 2006.

[54] F.-X. Standaert, E. Peeters, G. Rouvroy, and J.-J. Quisquater, "An Overview of Power Analysis Attacks Against Field Programmable Gate Arrays," in *Proceedings of the IEEE*, vol. 94, February 2006, pp. 383–394.

[55] M. Wirthlin, E. Johnson, and N. Rollins, "The Reliability of FPGA Circuit Design in the Presence of Radiation Induced Configuration Upsets," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)*, April 2003, pp. 133–142.

[56] M. Gokhale, P. Graham, E. Johnson, N. Rollins, and M. Wirthlin, "Dynamic Reconfiguration for Management of Radiation-Induced Faults in FPGAs," in *18th International 2004 Proceedings of Parallel and Distributed Processing Symposium*, April 2004.

[57] *Single Event Upset (SEU) Detection and Correction Using Virtex-4 Devices (xapp714)*, Xilinx, June 2006.

[58] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[59] *Virtex-4 MB Development Board User's Guide*, Memec, December 2005, version 3.0.

[60] *Virtex-4 Family Overview*, Xilinx, February 2006, version 1.5.

[61] B. Schneier, *Applied Cryptography*, 2nd ed. John Wiley and Sons, 1996.

[62] J. C. Villar, "SystemC/Verilog MD5." Opencores.org, September 2005, www.opencores.org.

[63] F. Lemmermeyer, *Reciprocity Laws*. Springer, January 2000.

[64] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*. New York, NY, USA: John Wiley and Sons, Inc., 2001.

Vita

James Braxton Webb was born in Durham, North Carolina in 1983 and grew up in Chapel Hill. After graduating from Chapel Hill High in 2001, he enrolled at Virginia Polytechnic Institute and State University (Virginia Tech). During his undergraduate career he assisted Dr. Robert Hendricks in the development of Lab-in-a-Box—a home experiment lab manual for undergraduate circuit analysis classes. The manual was published in August of 2006. In May of 2005, James graduated *magna cum laude* with a B.S. in Electrical Engineering and a minor in Music. James enrolled in Virginia Tech's B.S./M.S. program offered to select students and began his Masters education in conjunction with the completion of his undergraduate studies. In June of 2005, James began work with the Virginia Tech Configurable Computing Laboratory (CCM Lab) as a Graduate Research Assistant. In August of 2006, James completed his graduate education and received his M.S in Electrical Engineering. James is currently employed by JDS Uniphase Corporation, Test and Measurement Telecom Division, Germantown, Maryland as a hardware development and research engineer. His work still includes systems that utilize FPGAs.