A Cross Platform Method for FPGA Integrity Checking

Matthew Aaron Benz

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

In

Computer Engineering

Dr. Mark T. Jones, Chair

Dr. Peter M. Athanas

Dr. Cameron D. Patterson

August 30, 2007

Bradley Department of Electrical and Computer Engineering

Blacksburg, Virginia

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

Copyright 2007 ©, Matthew Aaron Benz

A Cross Platform Method for FPGA Integrity Checking

Matthew Aaron Benz

(ABSTRACT)

As embedded systems continue to evolve and the number of applications they support continues to increase, so does the diversity of the hardware they employ. As a result, the Field Programmable Gate Arrays (FPGAs), which have become fundamental elements in their design, have advanced in size and complexity as well. Because of this, it is now impossible to ignore the security implications that accompany such a progression. It is then not only important to prevent malicious attacks targeted at FPGAs from extracting the intellectual property contained in their configuration, but to now extend the research in this field by providing a cross-platform solution capable of securing the integrity of FPGA configurations at run-time. Today, there exist myriad attack strategies employed against FPGAs, the majority of which are seen in the form of semi-invasive attacks. These attacks manipulate the configuration of an FPGA and typically modify the state of the transistors that make up said configuration.

This thesis introduces a multi-platform method for checking the integrity of an FPGA's configuration. The details of the system's design and implementation are discussed in addition to the analysis of the design trade-offs met when employing the system across multiple FPGA families. The system is implemented entirely in hardware and resides on-chip, providing an FPGA the ability to act as private entity capable of successfully detecting when it has been maliciously attacked.

Acknowledgements

The contributions presented in this thesis would not have been possible if it were not for the support, guidance and motivation given to me from a very long list of friends, family and Virginia Tech faculty. For those who are not mentioned explicitly, the support you have given me has not gone unnoticed, and I would like to take this time to thank you.

To begin, I would like to thank my advisor Dr. Mark Jones for his guidance throughout my graduate academic career. Under his direction, I have learned the value of patience and thoroughness, and I am surely better off personally and professionally for it. In addition, I would like to thank Dr. Peter Athanas for his patience, knowledge and advice during my time as both a graduate and undergraduate student at Virginia Tech. Also, I would like to thank Dr. Cameron Patterson for serving on my graduate committee.

Next, I would like to acknowledge each member of the Virginia Tech faculty who played a role in furthering my education over the past 6 years. Without your guidance and wisdom this work would not have been possible.

I could not go without thanking Brac Webb for the role he played as both a friend and co-member on the Harris Design team at the Virginia Tech CCM lab. The contributions presented in this thesis could not have been possible if not for his work.

I am forever grateful to my parents, Linda and Larry Benz, for not only providing me the opportunity to obtain a graduate education, but for their love, patience and encouragement throughout my collegiate career and life. If it were not for you, there is no way I would be where I am today. In addition, I would like to thank my grandparents Ann and Conrad Benz for their continued interest, encouragement and support in both my academic career and growth as a

person. I would also like to thank Wendy Cruzan for enlightening me with her grammatical prowess. This work would have never been completed without your assistance.

Lastly, I would like to thank my friends and brothers of the Pi Kappa Alpha Fraternity. It was through the relationships I formed and the experiences I had while being a PIKE at Virginia Tech that helped make me the person I am today. I would like to specifically thank Eugene Shim for his continued social encouragement, as well as Michael Perry for his advice and academic timeline (a great source of inspiration and motivation). Last, but not least, I would like to thank Byron Baptist for being the best friend anyone could ask for. If it was not for your advice, trust and willingness to continually motivate, I defiantly could not have accomplished all that I have. In the bonds, φφκα.

Contents

List of Figures

[1] Figures and images used, which are not my own, are of fair use

List of Tables

Chapter 1

Introduction

1.1 Motivation

Since their introduction in the 1980s, Field Programmable Gate Arrays (FPGAs) have evolved from small configurable chips with limited application into run-time reconfigurable (RTR) multi-million gate hardware devices. Consequently, a significant share of the market, which has been historically dominated by Application Specific Integrated Circuits (ASICs), Digital Signal Processors (DSPs) and microprocessor based systems, is now occupied by FPGAs [55,71,73,74]. Modern FPGAs are capable of housing entire System-on-Chip (SoC) designs and are now used in the medical, industrial, networking, digital computing, telecommunications, wireless and defense fields [72]. FPGAs have now shifted from being an intermediate step in the design process to frequently appearing in the final product, making it imperative to consider the security implications they present.

Historically, the majority of the research in the field of configurable computing has focused on areas outside of the topic of FPGA security. However, as FPGAs begin to become the

backbone of many real world applications, the need for research on the topic of FPGA security has been recognized. A number of research efforts have begun to investigate the security of, not only the intellectual property contained in FPGA configurations [40,41,42,43,44], but auxiliary topics such as network traffic filtering [71], partial bitstreams [45] and using FPGAs to develop secure software platforms [39]. While these areas represent significant progress in this field, there have been few efforts that have succeeded in securing the integrity of FPGA configurations.

A system that has succeeded in securing aspects of the integrity of FPGA configurations on the Xilinx Virtex-4 platform is presented in [2]. This system is comprised of three parts. The first subsystem actively monitors the configuration of a FPGA, checking for malicious alterations made by an attacker. Next, a subsystem for securing partial bitstreams, which are used to reconfigure portions of the device's configuration, is presented. Finally, an implementation of a classic challenge-response protocol is outlined. This subsystem can be used by an external entity to ensure that the system is operating correctly.

1.2 Contributions

While systems such as the one presented in [2] have successfully secured the integrity of FPGA configurations, they have only done so on one particular platform for one specific device. This provides a solution that lacks portability and parameterizability.

Presented in this thesis is a cross platform method by which dynamic data can be actively masked from FPGA configuration bitstreams. In addition, an approach to extending the configuration readback and hash generation/comparison methodologies outlined in [2] across multiple platforms is presented. Also, a flow designed to automatically generate the relative readback bitstream locations of all dynamic flip-flop data present on any device in any Xilinx FPGA family is described. As a final contribution, a cross platform implementation of the challenge-response subsystem presented in [2] is outlined. When combined, these contributions

demonstrate that a FPGA configuration integrity checking solution can be developed that supports multiple devices spanning multiple platforms.

The cross-platform configuration integrity checking method presented supports two configurations, each of which is optimized to consume opposing sets of resources. This serves to provide system designers a security solution that best fits the resources they have available. Platform specific configurations of the dynamic data masking subsystem are also provided. These configurations are provided to increase portability and relieve the system designer of the burden of performing complex parameterizations when porting system to a new platform.

The cross-platform integrity checking solution presented was developed and tested under a series of simulated fault injection attacks generated though the use of partial reconfiguration. An analysis of the results of these experiments is described. In addition, a complete analysis of the systems resource consumption, timing characteristics and platform portability is presented. Finally, potential future work that could be used to improve the system is presented.

1.3 Thesis Organization

The remaining five chapters of this thesis are arranged as follows. Chapter 2 provides an overview of FPGAs and FPGA security. Also presented is an outline of the previous work performed in the area of securing the integrity of FPGA configurations. Chapter 3 describes the design and implementation of the platform dependent configuration integrity checker presented in [2]. In Chapter 4, a flow designed to automate the process of identifying the relative locations of all dynamic data in an FPGA readback configuration is described. Also, the method by which the configuration readback, hash computation and challenge-response components presented in [2] were extended across multiple platforms is described. To conclude this chapter, the approach to combining these contributions to form a platform independent solution to securing the integrity of FPGA configurations is presented. Chapter 5 provides a comprehensive multi-platform analysis of the resource requirements, timing characteristics and security strengths and

weaknesses of the system. Finally, in Chapter 6, a summarization of the research performed, as well as a discussion of potential future work is detailed.

Chapter 2

Background and Previous Work

The content of this chapter serves to provide a background of current and previous work on the topics covered in this thesis. These topics include FPGAs, security profiles and classifications, attack strategies commonly employed against FPGAs, extending tools across multiple FPGA devices/platforms and previous work in the field of FPGA configuration integrity checking.

2.1 FPGAs

Field Programmable Gate Arrays (FPGAs) are devices that contain programmable logic blocks and interconnect which can be configured to realize designs ranging from basic logic gates to complex embedded systems. As the name implies, FPGAs are “field programmable”, meaning that a system designer can re-program the device to model different sets of hardware as many times as needed. In general, designs instantiated on FPGAs draw more power and exhibit higher latencies than those implemented using application specific integrated circuits (ASICs). The tradeoffs for this lack in performance include shorter time to market due to rapid prototyping, cost efficiency, throughput and resource efficiency [47]. Another benefit FPGAs provide to

offset their lack in performance is dynamic partial reconfigurability [28]. Dynamic partial reconfiguration refers to the process of reconfiguring a portion of the design running on the FPGA in parallel with the execution of the rest of the design. The ability to reconfigure a portion of a hardware design at run-time provides limitless possibilities, and many research efforts have targeted this field [57,58,59,60,61]. Applications typically supported by FPGAs include digital signal processing (DSPs) [55,63,73,74], digital imaging and computer vision [33,69], cryptography [65,80], bioinformatics [64,76,77], software-defined radios [62,78,79] and software algorithm modeling [1,19,24,25].

2.2 Security Profiles and Classification

As the methods used to secure embedded systems and the intellectual property they contain continue to advance in robustness and complexity, so have the attacks that have been developed to compromise their security. In order to classify the ability of a security system to resist these attacks, IBM's security classification strategy, detailed in [2] and defined in [66], is presented.

In this classification strategy, IBM groups attackers into three classes, based on the attacker's ability successfully compromise the security of a design. These classes are outlined as follows:

Class I – "Clever Outsiders":

Class I attackers are typically well trained, but lack knowledge specific to the system being attacked. These attackers are limited to moderately sophisticated equipment and typically attempt to take advantage of the pre-existing weaknesses the system exhibits. Class I attackers do not have sufficient knowledge or resources to create a weakness in the system.

Class II – "Knowledgeable Insiders":

Class II attackers have extensive technical training and experience specific to the system being attacked. These attackers have access to highly sophisticated equipment used to analyze the system. While these attackers may not fully understand the design of each of the systems components, they do have access to the majority of these components for analysis.

Class III – "Funded Organizations":

Class III attackers are commonly seen as a diverse team of specialists, with each member of this team possessing related skills. When combined, these individual members form a team with the ability to perform in-depth analysis of each of the system's components. These attackers are capable of designing advanced attack strategies which produce weaknesses in the system using sophisticated analysis tools. Attackers in this class are also typically extremely well-funded, and are often times government backed.

Also provided in this security classification is a metric of how resistant a design is to potential attacks. In this metric, security ratings are based on the level of care put into the design of the system being evaluated, as well as the level of resources required to compromise the security of the system. This rating system is outlined as follows:

Level – LOW

Security features are in place, but can be compromised with the use of easily obtainable equipment costing no more than $2,768.

Level – MODERATE-LOW

Most inexpensive attacks are withstood. Successful attacks typically require an attacker with some special knowledge and moderately expensive equipment not to exceed $8,459.

Level – MODERATE

Only Level II and III attackers have a chance to successfully compromise the system. Special tools and skills are required, and the total equipment required to produce the attack costs up to, but not exceeding $84,590.

Level – MODERATE – HIGH

Detailed analysis of the system is needed to compromise its security. The equipment required to produce the attack costs up to $422,950. The attack may also require several Level II attackers with a wide range of skills to launch.

Level – HIGH

No known attacks that have the potential to compromise the system exist, and a new attack strategy must be developed. The total cost of supporting the attack may be over 2.77 million dollars[2] and the success of the attack is not guaranteed. Only large heavily funded organizations can support such an attack and are typically associated with government.

Using the security classification strategy outlined in this section it is possible to classify the level of security a system designed to protect an FPGAs configuration provides. Subsequently, the profile of attackers with the potential to succeed in an attack against the system can also be determined.

2.3 FPGA Attack and Protection Strategies

As FPGAs begin to appear as the backbone of many computing systems, the set of applications containing secure information that they support is growing as well. Because most FPGAs are SRAM-based, these attacks, which were previously designed to target embedded systems, can now be used to target not only the memories contained in FPGAs, but the configuration data of

[2] The monetary values presented were adjusted from the original values presented in [66] based on an inflation rate of 50.38%. This rate was based on the 16 year period starting at January 1st, 1991 and ending on January 1st, 2007

the device as well. This configuration data is a physical representation of the functionality achieved by the design contained in the FPGA. If an alteration is made to a portion of the configuration data that corresponds to a portion of the design instantiated on the device, the functionality of this design will also be modified. Not only does this provide attackers the ability to modify the contents of an FPGA's on-chip memories, but the operation of the design running on the device as well. The scope of the attacks used against SRAM-based devices is extremely wide, and contains a number of invasive, non-invasive and semi-invasive strategies [17]. The objectives these attacks aim to accomplish include obtaining secure information from the design [70], reverse engineering the design contained in the FPGA's configuration [47], cloning the configuration of the FPGA [44] and simply rendering the executing design inoperable [17]. In Sections 2.3.1 and 2.3.2, attack methodologies used to maliciously extract and modify FPGA configuration data are presented. In addition, neutralization strategies used to prevent these attacks are outlined.

2.3.1 Extracting Secure Information from a FPGA Configuration

A significant number of attacks that attempt to clone or reverse engineer the design contained in the configuration of an FPGA, using semi-invasive and non-invasive methods, exist. The most significant of these attacks include side channel, data extraction and bitstream interception attacks.

Side-Channel Attacks

Side-channel attacks are typically considered non-invasive or passive attacks [17]. These attacks typically attempt to obtain information from a cryptosystem by taking advantage of its physical attributes instead of the theoretical weaknesses in the algorithm it employs. The two most prominent physical properties typically targeted by side-channel attacks are timing characteristics and power consumption [67].

Side-channel attacks that target a system's timing characteristics are referred to as timing analysis attacks. These attacks typically take advantage of the fact that many cryptographic algorithms, while secure in the algorithm they employ, do not exhibit consistency throughout their operation. Breaks in timing consistency can stem from conditional branches in the cryptographic algorithm, optimization techniques and caching systems [67]. Timing analysis attacks have been used against cryptographic algorithms such as RSA and DES [68].

A power analysis attack is a form of side-channel attack that typically targets variations in gate-level power consumption. Among the devices which have become a target of these attacks, digital integrated circuits using CMOS technology are among the most prominent [75]. The power analysis attacks, which are typically employed against such devices, include simple (SPA) and differential power analysis (DPA) attacks [67].

To prevent side-channel attacks there are a number of security strategies that can be employed. Increasing measurement noise [81], power signal filtering [82,85] and asynchronous circuits [83,84] are all viable options for protecting a device.

Data Extraction Attacks

From [70], it can be seen that there are a number of semi-invasive attacks that have been proven successful at actively reading the contents of not only SRAM cells, but also Flash and even registers. As a result, these attacks are directly applicable to most SRAM-based FPGAs. One of the easier of these attacks to perform involves the use of a red laser and a relatively inexpensive microscope. This attack first removes the casing of the chip, and then focuses the laser on the chip's surface using the microscope. Once a map is made of the SRAM locations on the chip, the laser is swept across the mapped portions of the chip's surface and the states of the SRAM cells can then be observed. When observing a particular cell, if the luminosity of the top and bottom portions of the cell are compared, it is possible to determine the state of the cell. When the top portion of the cell is brighter, the state is a '1', and when the bottom portion is brighter, the state

is a'0' [70]. Because most commercial FPGAs are SRAM-based, if an attacker can actively readback the state of a SRAM cell, with enough time, large portions of the chip's configuration could also be determined. This would then provide an attacker a method by which they can clone the design contained on the device.

Bitstream Interception Attacks

One of the most straightforward attacks used against FPGAs involves intercepting the bitstream used to program the device between the root ROM and the FPGA at power on [44]. Bitstreams used to configure FPGAs contain the physical description of the design being programmed on the device. If an attacker obtains this bitstream, then the entire design contained within the bitstream is compromised unless additional security measures are taken. The most common method of preventing such an attack involves encrypting the bitstream prior to programming the device. The bitstream is encrypted using a secret key which is only known by the encrypting device and the corresponding decryption device used on the FPGA. FPGA vendors typically provide some form of bitstream encryption to help combat such attacks. Xilinx FPGAs use triple DES for the Virtex-II and Virtex-II Pro platforms and AES for the Virtex-4 and Virtex-5 platforms [10,11,12].

2.3.2 Altering FPGA Configurations

This thesis is primarily concerned with attacks with the potential to compromise the integrity of FPGA configurations. These attacks generally tend to be either invasive or semi-invasive. Invasive attacks, or attacks that physically modify the device being attacked, typically render the device inoperable [17].

Invasive Attacks on FPGA Configurations

Because invasive attacks suspend the execution of the design running on the device, they can be detected through the use of a challenge-response protocol. This protocol is implemented in the configuration integrity checker presented in this thesis. Preventing said attacks is much more difficult. Invasive attacks can prove successful against microprocessor-based systems because the layout of the system can been easily mapped, and the underlying hardware probed.

The configuration of a modern FPGA can contain millions of bits of configuration data. Thus, for an attacker to successfully probe this configuration data, they must have detailed knowledge of the design instantiated on the device, as well as how the components of this design are mapped to the configuration. Due to the level of complexity involved with probing the configuration of a FPGA, invasive attacks are not often used against these devices. For this reason, the prevention of invasive attacks is outside the scope of this thesis.

Semi-Invasive Attacks on FPGA Configurations

Semi-invasive attacks, which have the ability to alter the contents of SRAM-based devices (such as FPGAs), present the greatest threat to the configuration integrity checker outlined in this thesis. These attacks are relatively new to the computing industry and tend to be less expensive than classic invasive attacks, but as easily repeatable as non-invasive attacks. These attacks require the packaging of the chip to be removed, but their application does not damage the chip, as they do not require de-passivation or creating contacts on the chip [17].

Fault injection attacks are a form of semi-invasive attacks. Fault injection attacks can be defined as a method to systematically produce changes in the state of the transistors of which typically make up a larger component in a system. From [17,69] it can be seen that there exist inexpensive and effective methods to induce changes in transistor state so precise that the state of individual SRAM memory cells can be altered. As a result, an attacker employing such a strategy

has the ability to maliciously modify the configuration of an SRAM-based FPGA on a bit-by-bit basis. This attack could potentially compromise the configuration integrity checker presented in this thesis if the right approach is taken. If the goal of an attacker was simply to induce unintended behavior in the system, this attack could find great success. However, if the aim of the attacker were to manipulate bits of the FPGAs configuration to determine information about the system, they would most likely not have success. In order for such an attack to prove successful, the attacker would need to have extensive knowledge of, not only the composition of FPGA bitstreams, but also of the SRAM cells used in the FPGAs design. Such an attacker would require a detailed layout of how the attacked design was mapped and routed for the device being used as well.

2.4 Software Portability

As the number of systems that utilize FPGAs continues to increase, so does the number of software platforms that must be supported by the applications used in their development. With modern FPGAs beginning to become third, fourth and even fifth generation devices, the range of FPGA devices and families that must be supported by this software is rapidly increasing as well. Today, software designers must then produce solutions that are not only correct in functionality, but that support a wide array of software platforms and FPGA devices used in today's modern designs. In Sections 2.4.1 and 2.4.2, previous work in promoting software portability across various software platforms and hardware devices is described.

2.4.1 Software Platform Portability

In today's configurable computing industry, system designers require new software to support platforms such as Microsoft Windows, Mac OS X, common flavors of Linux such as Red Hat and Debian, as well as other UNIX based systems. Software used in FPGA development such as JBits [52], the Xilinx XST [56] and Altera Quartus toolkits [90], ADB [48] and PARBIT [49] have all begun to provide software solutions capable of running on these platforms. While the

approach taken to providing this portability is not uniform across the FPGA software design industry, there are a few common methods used when developing said software.

Java Based GUIs and APIs

Applications such as JBits [52] and ADB [48] were designed as Java APIs to take advantage of the native portability Java provides. This approach defines a set of platform independent function calls that can be used by a software designer to perform specific tasks. This is the easiest approach to ensuring a high level of portability, as back end applications written in Java are compatible on almost every platform. GUI applications written in Java, such as some of the accessory application front ends provided in the Xilinx XST [56] and Altera Quartus [90] tool kits, are not as portable by nature. While still relatively portable across multiple platforms, applications designers face the choice of using either the 1.1 or 1.2 versions of the Java Development Kit. The 1.1 version is rather poorly designed but boasts great portability, while the 1.2 version contains a superior toolkit and greater capabilities, but at the cost of some portability loss.

C / C++ Based Applications

A common software approach to ensuring portability utilizes the native C and C++ programming languages, as well as additional standard C/C++ libraries, to develop powerful backend applications. Applications such as PARBIT [49] and the back ends of the Xilinx XST [56] and Altera Quartus [90] tool kits utilize this approach. The core C programming language and the standard C libraries themselves are extremely portable. Typically, it is up to the programmer, however, to manage the function calls used to ensure that the application will be supported by compilers across multiple platforms. The portability limitations of this approach begin to become apparent when inter-process communication and multi-threading are done. C++ features such as templates, standard I/O libraries and exception handling are also typically not well supported across multiple platforms [91].

Scripting Languages

Scripting languages have reasonably good portability, but are not as strong as Java or C. Perl offers good portability and provides programmers a set of bindings to the Tk toolkit that supports portable GUIs across Unix, Mac OS X and Windows [92]. As a downside, some Perl scripts require add-on libraries that may not be included in standard distributions of many platforms. An example application that uses Perl scripts can be seen in the database creation functionality of the ADB [48] application.

Shell scripting, however, does not provide a high level or portability. The bash shell has become standard across many platforms, and native shell commands are widely supported. Problems arise however when shell scripts make use of other auxiliary commands which may not be standard on many platforms.

The approach used in developing the automated dynamic data mapping method presented in this thesis uses C applications controlled by standard shell scripts. The shell script implemented ensures portability through the use native bash shell commands and calls to auxiliary lightweight C applications.

2.4.2 Extending Software Tools Across Multiple FPGA Devices and Families

Many software applications used in the FPGA design process require advanced knowledge of the make up of the FPGAs for which they are used. Applications such as JBits [52] and ADB [48] rely on determining proprietary information about the bitstream makeup used to program Xilinx FPGAs. Determining this information for new devices within a family is a reasonable task, however producing this information across multiple FPGA platforms can prove to be very challenging. Typically, applications such as these provide native support for the most current FPGAs on the market at the time of their deployment, and leave future platforms generally unsupported. It is for this reason that it becomes very challenging to utilize the advanced features of modern FPGAs in conjunction with popular third party FPGA design tools that only support

previous generations of the hardware. Because the interworkings of most FPGAs are kept proprietary, and updates to third-party FPGA related software are infrequent at best, system designers are commonly left to rely on software provided by FPGA vendors to support designs implemented on next generation devices. This not only discourages future third-party FPGA related software development, but allows vendors to control the power of new FPGA designs by selecting which subsets of configuration information they wish to make public.

2.5 FPGA Configuration Integrity Checking

While there has not been an enormous amount of research done in the area of FPGA configuration security, there has been work done in single-event upset (SEU) detection and recovery, as well as configuration integrity checking on the Xilinx Virtex 4 platform.

2.5.1 SEU Discovery and Repair

When used in space and high altitude applications, SRAM based devices, such as FPGAs, are susceptible to small faults that may alter their configuration. These faults are commonly referred to as single-event upsets, or SEUs. SEUs are typically radiation-induced and are seen in both low earth orbit and in the presence of solar flares [89].

In order to detect and recover from an SEU, an approach was developed in [87] that actively monitors the configuration of an FPGA through the use of a cyclic redundancy check (CRC) computation. This computation was performed on frame-by-frame basis for the entire configuration of the device that was read back. If the resulting CRC value of a frame was different, it was known that an SEU had occurred and effected this portion of the devices configuration. After the detection of such an event, the system utilized RTR to correct the alteration. Most modern FPGAs can withstand exposure to large amounts of radiation without being subjected to an SEU [89]. However, for high-reliability applications, Xilinx has provided its own SEU discovery and repair module [88]. This module operates in a similar fashion to the

system described in [87], and has the capability to correct SEUs. Altera's SEU detection and repair strategy is present on all Stratix and Cyclone FPGAs. This approach provides built-in dedicated circuitry to check the FPGAs configuration for SEUs. While these strategies can successfully detect SEUs on SRAM based FPGAs, they cannot, however, detect and recover from malicious semi-invasive fault injection attacks [2]. This shortcoming forms the basis for which the configuration integrity checking solution presented in this thesis was founded upon.

2.5.2 Configuration Integrity Checking on the Virtex 4 Platform

In [2], a system capable of securing FPGA configurations against such semi-invasive attacks on the Xilinx Virtex 4 platform is presented. The core of this system is defined to include the readback controller, hash generator, hash comparator and challenge-response components of the system. For the remainder of this thesis, this core will be referred to as the Platform Dependent Configuration Integrity Checker, or PDCIC. A significant portion of the work presented in this thesis serves to extend the components of the PDCIC core across multiple platforms. These extended components combined with a platform independent dynamic data masking controller form the multi-platform configuration integrity checker presented in this thesis.

Dynamic Data Identification

When designing a system that can actively detect when a malicious alteration to an FPGA configuration has occurred, the configuration data of the FPGA must be continuously monitored for malicious changes. This is typically achieved through the use of active read back via the FPGA's internal configuration access port (ICAP). In order to effectively produce a static readback configuration, the dynamic data located in the FPGA's configuration bitstream being read back must be masked out. The static readback configuration is then analyzed to determine if a malicious attack was being performed. The process by which this dynamic data was identified and removed is the basis for the automated, platform independent solution presented in this thesis.

CLB Block Hash Generation and Comparator

If readback configuration data is being accurately masked for dynamic data, it is possible to generate checksums which represent segments of the data. These checksums can then be used to determine exactly when and what specific portion of an FPGA has been maliciously attacked. The MD5 hashing algorithm, used by the PDCIC to generate these checksums, was also used in the platform independent integrity checker presented in this thesis. This hashing algorithm is typically used to produce a unique, fixed-length representation of a message of arbitrary length. The implementation of the algorithm used was obtained from [15]. The method by which the hashes produced by the MD5 algorithm were generated, stored and compared also form the foundation for the methods used in the multi-platform version.

Providing Assurance of Correct Operation

The PDCIC employs a classic challenge-response subsystem for providing an entity external to the FPGA, a method by which the health of the security system can be polled. There are many ways that this protocol can be implemented [21], however, the particular design that was used was obtained from [2]. This challenge-response system, which acts as the claimant, accepts an arbitrary length challenge from the verifier and responds by returning a hash of a secret key concatenated with the original message to the user. While the structure of this challenge-response system was significantly altered in the multi-platform version of the configuration integrity checker presented in this thesis, its general concept and framework were used as a foundation for the design of the new subsystem.

2.6 Applications That Can Benefit From Configuration Integrity Checking

There has been a significant amount of work done to secure user and system IP contained in embedded systems on FPGA platforms. The majority of the system design in this area focuses on protecting embedded systems containing a processor core as the central point of the protected design. In [40], a FPGA based network processor is presented that takes advantage of RTR to provide both user based operation and concealment of the intellectual property (IP) contained in a users design. This functionality is achieved through the use of a modular approach to the design of the processor core. When a valid user of the system is not present, the design can effectively remove user IP from the device, preventing malicious reverse engineering of the design. A system presented in [41] implements a key management system instantiated on an FPGA that can provide secure user sessions. This design can effectively prevent an outsider from discovering the implementation details of an embedded application or CPU that is being protected.

These designs have successfully secured user and system IP on several FPGA platforms. Many non-invasive attacks that were commonly used against embedded processing systems can now be neutralized by the security strategies they employ. Unfortunately, the level of security these systems proved does not protect against invasive and semi-invasive attacks against an FPGAs configuration. If a configuration integrity checking strategy was utilized as an auxiliary method for protecting the integrity of these systems, great success in providing resistance against such attacks could be seen. Due to the disparity in the FPGA platforms such security systems are implemented on, a multi-platform integrity checking solution would be needed to realize such an auxiliary security solution. The cross-platform FPGA configuration integrity checking solution presented in this thesis provides a solution that satisfies both these requirements.

Chapter 3

Method for Securing FPGA Configurations on the Virtex 4 Platform

This chapter outlines the design and implementation of the platform dependent configuration integrity checking core (PDCIC) presented in [2]. In this chapter, the configuration readback controller, hash generator/comparator and challenge-response components in this core are outlined. These components are later extended to support the design of the cross-platform integrity checker presented in this thesis. This chapter is intended to provide insight into the origins of the multi-platform configuration integrity checker's design outlined in Chapter 4.

3.1 FPGA Configuration Readback Control

The FPGA configuration data, which must be monitored for malicious modifications, is obtained using active readback from the FPGAs internal configuration access port (ICAP). The ICAP module provided by the FPGA manufacturer (in this case Xilinx) provides an interface to the

FPGA's internal status and configuration registers [10,11,14]. To begin active readback of the FPGA's configuration, several commands needed to be written to a number of different configuration registers. These commands specify the parameters of the data that is to be read back. These readback initiation commands correspond to commands 0 though 19 in Table 3.1. Once written, the ICAP will begin to clock out the internal configuration data of the FPGA starting at the address written to the frame address register (FAR) in command 17. The ICAP will then proceed to read back the number of bytes specified by the write to the FDRO register in command 19. Once all the desired configuration data has been read back, the ICAP is shutdown though the use of commands 20 through 25 outlined in Table 3.1.

	Command Index	Command Description	Command Value
Startup ICAP	command[0]	Synchronize the ICAP	0xAA_99_55_66
	command[1]	No Operation	0x20_00_00_00
	command[2]	Write 1 word to Command Register	0x30_00_80_01
	command[3]	Write RCRC Command To Command Register	0x00_00_00_07
	command[4]	No Operation	0x20_00_00_00
	command[5]	No Operation	0x20_00_00_00
	command[6]	No Operation	0x20_00_00_00
	command[7]	No Operation	0x20_00_00_00
	command[8]	No Operation	0x20_00_00_00
	command[9]	No Operation	0x20_00_00_00
	command[10]	No Operation	0x20_00_00_00
	command[11]	No Operation	0x20_00_00_00
	command[12]	No Operation	0x20_00_00_00
	command[13]	No Operation	0x20_00_00_00
	command[14]	Write 1 word to Command Register	0x30_00_80_01
	command[15]	Write RCFT Command to Command Register	0x00_00_00_04
	command[16]	Write 1 word to the Frame Address Register	0x30_00_20_01
	command[17]	Write Type 2 Frame Start Address	0x00_00_00_00
	command[18]	Read From FDRO Register	0x28_00_60_00
	command[19]	Read N Words From FDRO Register	0x48_02_B5_7A
Shutdown ICAP	command[20]	No Operation	0x20_00_00_00
	command[21]	No Operation	0x20_00_00_00
	command[22]	No Operation	0x20_00_00_00
	command[23]	Write 1 word to Command Register	0x30_00_80_01
	command[24]	Write RCRC Command To Command Register	0x00_00_00_07
	command[25]	Write 1 word to Command Register	0x30_00_80_01
	command[26]	Write Desynchronize command to Command Register	0x00_00_00_0D
	command[27]	No Operation	0x20_00_00_00
	command[28]	No Operation	0x20_00_00_00

Table 3.1: Readback commands used to startup and shutdown the Virtex 4 ICAP

A finite state machine (FSM) was designed to write the commands shown in Table 3.1, one byte at a time, to the ICAPs input data port. This FSM also is used to process the readback configuration data output by the ICAP on its output data port. The layout of the internal registers written to / read from by these commands can be found in [11]. The operation of this FSM is outlined in Figure 3.1, which was described in [2].

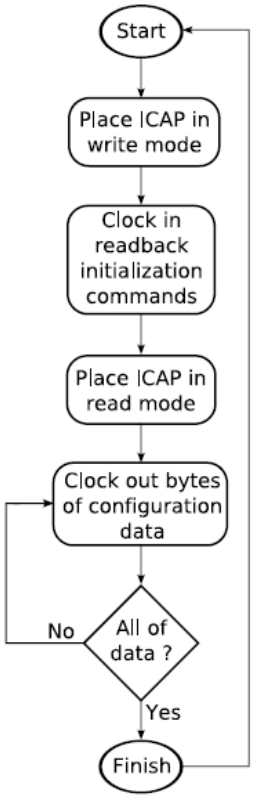


Figure 3.1: Outline of the FSM used to read to / write from the ICAP on the Virtex 4 platform [2].

3.2 Detection of Malicious Configuration Modifications

The strategy employed by the PDCIC for detecting when and where a malicious attack on the FPGA's configuration has occurred involves continuously computing hash values on blocks of configuration data. As these hash values are computed, they are compared against "trusted" hash values for the same blocks of configuration data which are known to represent the unaltered configuration of the device. Any difference between the two hash values would represent a maliciously altered portion of the FPGA's configuration data. To compute these hashes values, first a hash function needed to be selected. This hash function must not only provide sufficient latency and resource utilization characteristics, but excellent resistance against brute force and design-based attacks.

3.2.1 MD5 Hashing Algorithm

The hash algorithm selected for the PDCIC is the MD5 hash computation algorithm. Typically, the MD5 hash algorithm takes an input message of arbitrary length (in 512 bit chunks of data) and produces a fixed length (128 bits in the PDCIC) unique representation of that message. The characteristics one would look for in selecting such an algorithm are speed of computation, low resource requirements and the level of security provided by the algorithm. For hashing algorithms, there are two main security related properties which are desirable. The first is the algorithm's "one-wayness" [21]. The second, and in relation to the security of the components in the PDCIC being the most important, is the ability of the hash function to minimize collisions [20]. A collision is defined to be two distinct sets of input messages that produce the same hash value as a result. The MD5 algorithm exhibits both of these properties, and because it also demonstrates sufficient slice utilization and computation latency characteristics, it was chosen for this design [2]. The MD5 module used in the PDCIC was obtained from [15]. Its operation was controlled by a FSM, whose operation is outlined in Figure 3.2, described in [2].

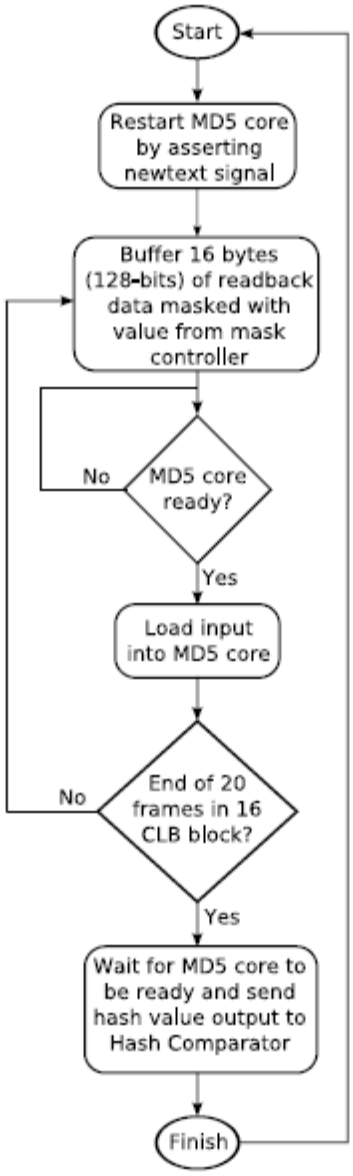


Figure 3.2: MD5 hash function finite state machine outline [2]

3.2.2 Hash Value Granularity Considerations

When considering the granularity at which hash values should be computed, there are several factors that must be considered. These factors include, but are not limited to, the precision of the locality of the attack that can be detected, the overhead of the hash computation algorithm which is incurred and amount of memory consumed by the system.

The smallest granularity that could be selected on the Virtex 4 platform would be an *n-length* partition of the readback bitstream, where *n* is the minimum size of an input message to the MD5 hashing algorithm (512 bits in this case) [15]. Choosing such a small granularity would allow one to precisely determine the area(s) of the FPGA's configuration that were modified, as well as provide fast configuration restoration times. The price a system designer would pay for choosing such small granularity size would be the huge overhead from hash computations that would be incurred. If this granularity were set at one hash per readback configuration frame, this would also result in a very large memory requirement. On the LX25 device, the readback configuration bitstream contains 6256 frames. As a result, all 6256 of the corresponding 128-bit trusted hash values for these frames would have to be stored in memory. Conversely, choosing the largest possible granularity size would result in one hash value that would represent the entire CLB section of the readback bitstream. Such a granularity would minimize the overhead of hash computations and reduce the amount of memory needed to store the resulting hash values. Unfortunately, it would also make run-time repair of the malicious alterations difficult to perform. This is because the entire FPGA would have to be reconfigured to correct malicious alterations, even if they are as small as a few bits. A granularity this large would also make it difficult to determine exactly which portion of the FPGA is being attacked. This would prevent the systems designer from making design alterations to combat attacks that are being employed. To achieve a middle ground, the the PDCIC set the granularity at a block of 20 frames for the Virtex 4 platform. This corresponds to 1 block of 16 CLBs per hash, resulting in 168 hashes being used to represent the entire CLB configuration.

3.2.3 Attack Locality Determination

When a malicious alteration of the FPGA's configuration is detected by the hash comparator of the PDCIC, the number of the hash containing altered configuration data is output. In order to identify the location relative to layout of the FPGA which was attacked, it is necessary to understand how the hash values produced correspond to physical portions of the FPGA. From Figure B.4, it can be seen that frame addresses increase from left to right and bottom to top on the FPGA. As a result, frames are readback starting at the lower left corner of the FPGA and ending in the upper right corner. It would then follow that the index of resulting hash values would increase in the same manner. The layout of the physical dimensions of the chip, in proportion to hash value indices, is outlined in Figure B.5.

3.3 Ensuring Reliable Operation

An attacker with the ability to maliciously alter the configuration data of an FPGA can potentially disrupt the operation of the PDCICs components, and thereby leave the configuration of the FPGA and any design it contains exposed. Thus, a method was put in place that allows an entity external to the FPGA to poll the health of the configuration integrity checker. The method selected was a classic challenge response subsystem. In the challenge-response component of the PDCIC, a challenge is issued from an external entity containing an input message of arbitrary length prepended with the length of the message. This message is then concatenated with a secret key to which only the entity issuing the challenge and the challenge-response subsystem have access. The message formed by this concatenation is then processed by the MD5 hash function, and its result returned in the form of a response to the challenger. In the PDCIC, the secret key used by the challenge-response subsystem was selected to be a hash of all 168 current hash values. This requires the PDCIC to store 168 128-bit words in the form of the current hash values. In addition, the 168 trusted hash values are also stored. Storing both the current and trusted hash values not only provides the challenger the ability to poll the systems health, but also allows this external entity to determine if the current FPGA configuration has been altered.

This is possible because the secret key used in forming the response is a hash of the 168 current hash values. Any divergence between these values and the trusted hash values would result in a different secret key being produced. This would then result in a different response, which is received by the challenger. An outline of this system can be seen in Figure 3.3, which was taken from [2].

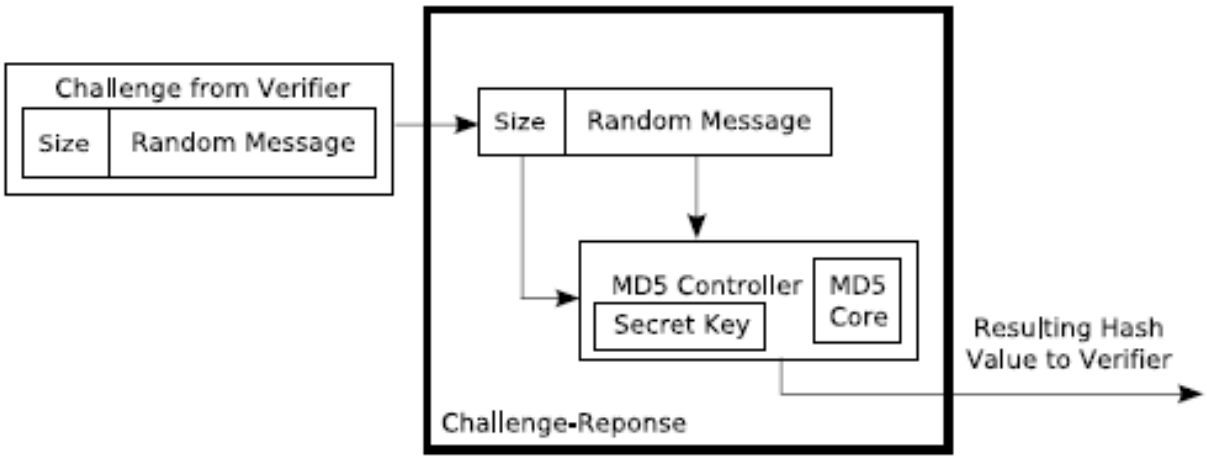


Figure 3.3: Challenge response subsystem architecture

3.4 Coverage of Future Work

In [2], several areas of potential future work to improve the design of the PDCIC were outlined. These areas include improving upon the initial conditions the system is dependent upon, increasing the frequency at which the system operates, and increasing the portability/parameterizability of the system. The work presented in this thesis focuses primarily on increasing the portability and parameterizability of the components that were inherited from the PDCIC. However, adjustments to conditional stipulations put on portions of the design, such as the challenge-response subsystem, were addressed as well.

Chapter 4

Multi-Platform Configuration Integrity Checker Design and Implementation

This chapter describes the design methodologies used in developing the components of the cross-platform configuration integrity checker presented in this thesis. First, the dynamic data identification and masking strategy is outlined. The methodology for automating this process and extending it across multiple FPGA platforms is then presented. Next, the approach used to extending the configuration readback, hash generation/comparison and challenge-response components of the design, outlined in [2], across multiple platforms is described. The chapter is concluded with a discussion of the design and implementation of the serial I/O subsystem employed by the cross-platform integrity checker.

4.1 Dynamic Data Identification and Masking

As with any hardware design, a design running on an FPGA contains two parts, a static portion and a dynamic portion. The dynamic portion of a design contains values that are constantly changing according to the operation of the static portion of the design. As a result, this dynamic

portion of the design must not be considered when checking the design's readback configuration for malicious alterations. If these dynamic portions of the configuration are considered, upon detection of a change in the design's configuration, it would be extremely difficult to determine the source of the alteration. Moreover, it would then be impossible to tell if the configuration of the design changed due to these dynamic components or due to a malicious attack. Once these dynamic portions of a design are masked from the configuration data being monitored for malicious attacks, one can reliably determine when a design's configuration data has been maliciously altered.

4.1.1 Dynamic Data Identification Strategy

To determine where dynamic data is located in the configuration bitstream of a Xilinx FPGA, an approach was developed which takes advantage of Xilinx logic allocation (.ll) files. These logic allocation files can be generated using the Xilinx Bitgen software (with the "-l" option specified) [56]. This software is typically used to take a user's design that has been synthesized, mapped, and routed, and generate a bitfile (.bit), which can be used to configure the target FPGA with the user's design [56]. Logic allocation files provide the frame address and bitstream offsets of all data that is considered to be dynamic in FPGA bitstreams. As a result, they are a critical component in the dynamic data identification strategy outlined.

In the method developed, a design was created which occupies one column of combinational logic blocks (CLBs) on the FPGA being mapped for dynamic data. This design was developed in such a way as to occupy all flip-flop resources in this CLB column. Because these resources are utilized, their relative locations, and therefore the locations of all dynamic data in this column, will be displayed in the resulting logic allocation file. This provides the information necessary to appropriately mask out all possible dynamic data locations in this column of CLBs. It should be noted that not all flip-flops in every column are utilized as dynamic data in every design, resulting in small static portions of the configuration that are not included in hash computations. When these flip-flop locations are left unutilized, it typically

means that they are not a part of the design being implemented, and therefore it is a valid assumption that leaving them unprotected by the configuration integrity checker does not present a security threat to the system.

In this strategy it is assumed that LUTs in the column be analyzed are not configured to act as RAM modules. If this assumption were not present, the bitstream locations of the dynamic data in these modules would need to be determined as well. This would be advantageous if a dynamic data masking strategy custom to a particular design was being developed. However, it is not advantageous to mask every possible location of this data in the design-independent version of the dynamic data masking strategy. Even if LUTs contained in each CLB column had not been configured as portions of a RAM module, they would still be masked using this approach. This would result in large static portions of the FPGAs configuration that are not included in the hash value computation for each block of hash data. Because this data is not considered in these computations, these static portions of the design contained in the configuration would be left unprotected from malicious attacks.

After the location of all dynamic data in this particular column was determined, the design was iteratively constrained to occupy each CLB column on the FPGA being mapped. The resulting absolute frame addresses, as well as relative (to the frame being addressed) bitstream offsets were copied from the generated logic allocation file for each column. The frame addresses, which correspond to locations of the dynamic data in each CLB column, are by default in the Xilinx specific format used to read and write to/from the FPGA's frame address register. An example of this format for the Xilinx Virtex 4 family is shown in Figure 4.1.

Address Type	Top / Bottom Bit	Block Type			Row Address					Column Address								Minor Address					
Bit Index	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 4.1: Virtex 4 frame address description

The frame address and bitstream offsets obtained from the method outlined in Section 4.1.1 were then be converted to frame addresses relative to configuration bitstream that is read back from the FPGA's ICAP. The conversion method performed is outlined in Equation 4.1. This conversion process was automated using lightweight software. A detailed description of this automation process is contained in Section 4.2.2. An example of both the Xilinx formatted frame addresses and resulting relative frame addresses for the LX25 device on the Virtex 4 platform can be seen in Figures B.1 and B.2, respectively.

$$Absolute\ Frame\ Offset = \frac{(Absolute\ Bitstream\ Offset - Frame\ Relative\ Bit\ Offset)}{(Words\ per\ frame \cdot Bits\ per\ word)} \quad (4.1)$$

Once all the relative frame addresses that contain dynamic data were obtained, all that was needed to be done was to find the bit locations inside these frames that corresponded to bits of dynamic data that must be masked out. These bits are provided in the logic allocation file, and form regular patterns inside frames that contain dynamic data. These patterns are uniform inside each platform, and only differ in length for platforms such as the Virtex II and II Pro platforms that do not have uniform frame lengths. To demonstrate the difference in bit patterns across multiple platforms, examples of these bits that must be masked for both the Virtex II XC2VP30 and Virtex 4 LX25 devices can be seen in Figures A.2 and B.3, respectively.

After all frame addresses relative to the FPGAs readback configuration bitstream were determined, a FSM was created to mask the corresponding data. This FSM systematically determined when data that was being read back belongs to a frame that must be masked. If needed, this dynamic data was then selectively masked from the configuration data that was being read back before being used in hash value computation.

4.1.2 Memory Considerations

From a configuration standpoint, the FPGA is divided into three major sections: (1) clock / IO / CLB data, (2) BRAM data, and (3) BRAM interconnect. From the Xilinx frame address

description in Figure 4.1, it can be seen that a frame is the smallest addressable segment in the device. This is true for all devices in all Xilinx FPGA families. Frames are arranged into block types, row addresses, column addresses and minor addresses. Block types correspond to type of data that is stored in the corresponding frames they address. As one would expect, block types are arranged into the three major categories previously mentioned, with the assigned addresses shown in Figure 4.2.

Block Type	Assigned Address
Clock / IO / CLB data	000
BRAM data	001
BRAM interconnect	010

Figure 4.2: Xilinx block type layout

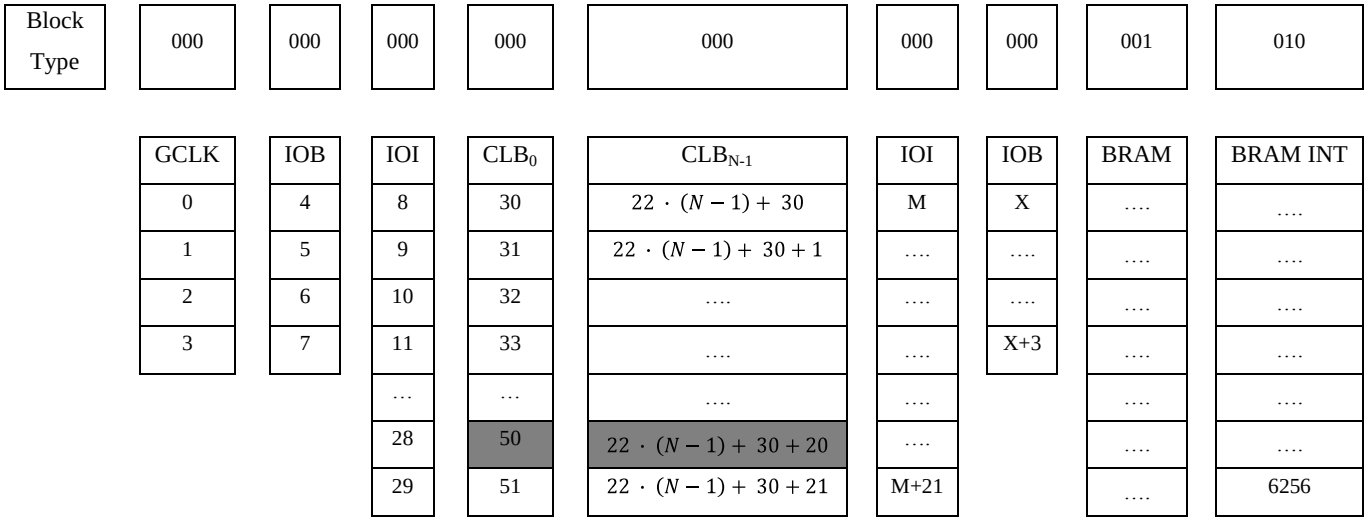
On most FPGAs, dynamic data comes in two basic forms: flip-flop state information and RAM data. Flip-flop state information is typically stored in the CLB section of the FPGA corresponding to block type 000. RAM data, however, can potentially be stored in either block type 000 or block type 001. This is a result of the fact that FPGA manufacturers provide the system designer the capability of configuring the lookup tables (LUTs) contained in CLBs into RAM modules. Depending on the parameters of a design, the synthesis tools used to synthesize the design will analyze the design and determine if the RAM data structures present would be best served to be instantiated in the dedicated block RAM (BRAM) provided on the device, or in LUT RAMs contained in user space. This is of major concern when developing a dynamic data idnetification strategy. This is primarily because only flip-flop data located in the CLB section of block type 000 is masked from readback configuration data when using the dynamic data identification strategy outlined in Section 4.1.1. If LUT RAMs are instantiated in the CLB data space, the CLB portion of the configuration data that is read back (which describes this space) will be continuously changing due to its contents containing RAM data that is not static. This makes it impossible to tell if changes in the configuration data are the result of a malicious attack or simply a result of the components of the design manipulating RAM contents. It is for this

reason that an assumption of the system's design requires all RAM modules to be instantiated either outside of the protected region of the CLB section or in the dedicated BRAMs of the FPGA. As a result, the only dynamic data located in the protected CLB data space is flop-flop data whose locations can be masked out using the technique described in Section 4.3.2. This allows the system to effectively detect when malicious changes have been made to the FPGA's configuration.

4.1.3 FPGA Configuration Bitstream Layout

In designing a FSM capable of masking out the dynamic data contained in the CLB portion of the readback data stream, it is necessary to identify where the CLB section actually begins and ends in this data stream. For the Virtex II/II Pro and Virtex 4 Xilinx FPGA families, this portion of the configuration readback bitstream begins at the 30^{th} frame which is read back (Starting from frame 0). To determine where the CLB section ends, the length of this section must be determined. On the Virtex II and Virtex II Pro platforms, the length of the CLB section is device-dependent and can be found in the Virtex II and Virtex II Pro user guides [9,10]. On the Virtex 4 platform, however, this value must be computed. The first step in determining this value is to find the number of logical frames contained in each Xilinx formatted frame address. From analyzing the logic allocation files produced by the flow outlined in Section 4.1.1, the number of logical frames located in between each frame containing dynamic flip-flop data can be found. On the Virtex 4 platform, this value is 22 frames. It can then be taken that each Xilinx formatted frame address corresponds to 22 frames in the readback configuration bitstream. Next, the index of the frame inside each block of 22 frames that contain dynamic data must be found. This index can be computed by subtracting the index of the first frame in the CLB section of the readback configuration (30) from the index of first frame containing flip-flop data (50). Therefore, inside each block of 22 frames, the 20^{th} frame contains dynamic data that must be masked out. This value can then be used to compute the last frame in the CLB section. This is done by determining the last frame to contain dynamic flip-flop data in the CLB section from the logic allocation files produced and adding 2 to it. The last frame containing dynamic data represents frame 20 in a 22-

frame block, so the last frame in the block, and therefore the CLB section of the configuration readback bitstream, would be the index of the last frame containing dynamic data plus 2. An outline of the configuration readback bitstream is shown in Figure 4.3.



**The frame highlighted grey represents the frame at offset 20 inside the blocks of 22 frames, and contains dynamic flip-flop data*

Figure 4.3: Outline of the Virtex 4 readback configuration data stream

The frames located after the CLB section (after frame 4350) in the readback bitstream correspond to I/O and BRAM data. The data contained in these frames represents either: (a) dynamic data corresponding to an I/O or memory component of a design, or (b) static data that represents an unused portion of the device. It is for these reasons that the configuration integrity checker does not protect the configuration data present in these regions.

Now that the entire layout of the readback bistream configuration is known for both the Virtex II / II Pro and Virtex 4 platforms, it is possible to create a FSM to mask this data out of the readback bitstream.

4.2 Platform Independent Dynamic Data Masking

The manual dynamic data identification process can prove to be extremely tedious (especially as devices grow in size and complexity), as it can require several days to map a single FPGA. As a result, it would be desirable to develop method to automate this process. Such automation would reduce the time required to map the dynamic data contained in the CLB section of an entire FPGA from hours or even days down to only a few minutes. It would also be advantageous to remove the complex parameterization needed in the mapping process and replace it with a single and straightforward input file. Such a method would serve to remove the device and family restrictions of the system presented in Chapters 3, and provide the opportunity for such a system to easily be ported across devices and even platforms. An approach that embodies all of these characteristics was developed, and its design is outlined in Sections 4.2.1 and 4.2.2.

4.2.1 Approach

The approach that was developed to automate the dynamic data mapping process is outlined in Figure 4.4 and includes the following steps:

1. Production of a map file used to represent the layout of the target device
2. Generating both a custom top module used for instantiation and a user constraints file
3. Producing logic allocation files for each column of the target device
4. Compiling all the produced logic allocation files into a tabular representation of the dynamic data locations relative to the FPGA's configuration.

In order to accomplish these tasks, several pieces of lightweight software were developed. The generate_ll bash script was used to control the automation process and provide the appropriate parameters to the called executables. This script uses only native bash function calls to promote portability. The gen_config application was developed in C / C++, and serves to produce a custom verilog top module and device specific user constraint file. These custom

generated files were then used to instantiate a module that fills all flip-flops contained in a specific column of CLBs. Instantiating such a module allowed the generate_ll script to run the appropriate Xilinx software to generate a logic allocation file corresponding to the FPGA configuration containing only the targeted column being filled. Once logic allocation files were generated for each column on the device, the compile_results executable, which was also written in C, was used to compile these results into tabular format. This table contains Xilinx formatted frame address, absolute bit offsets, relative bit offsets (to the current frame) and relative frame numbers (to the readback bitstream) corresponding to all dynamic data in the configuration. This compilation was then be used to configure a dynamic data masking FSM.

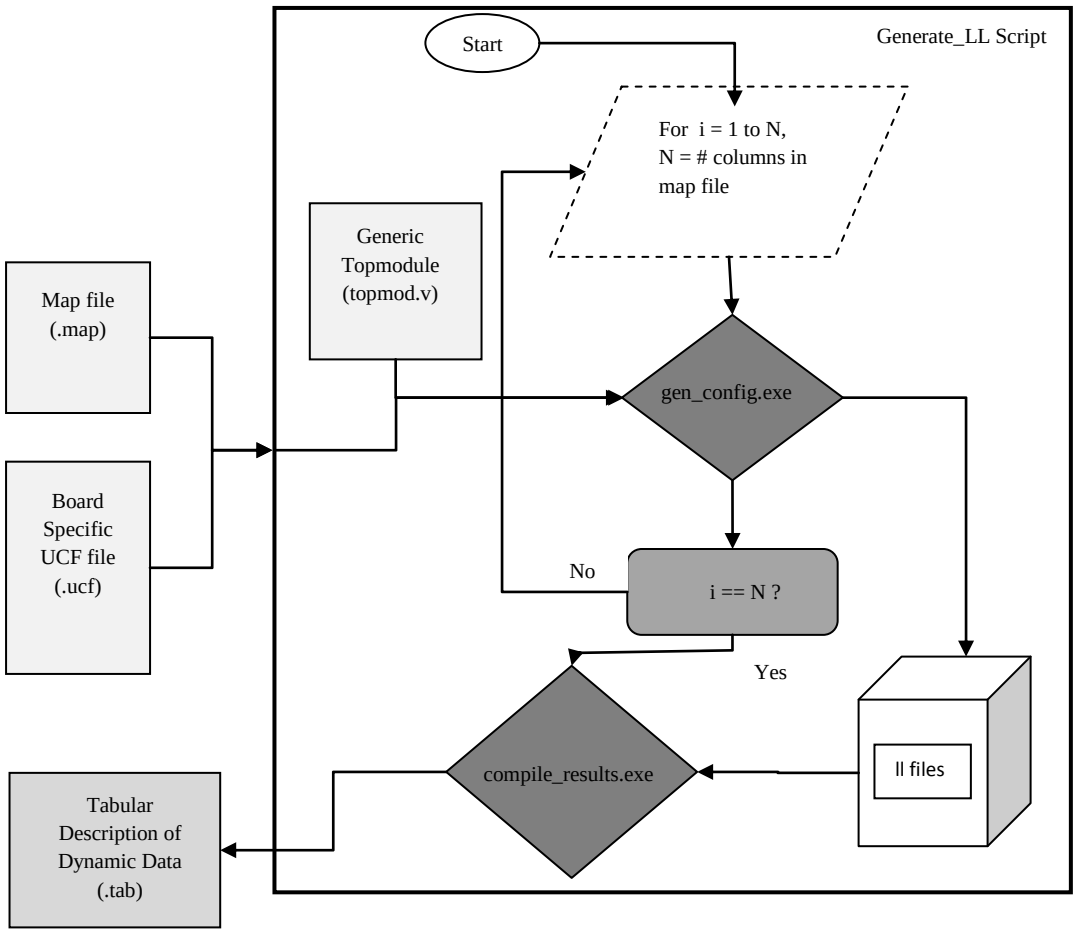


Figure 4.4: Diagram of Automated Dynamic Data Identification Process

Examples of the logic allocation file generation process, as well as a sample section from the tabular data produced from the compile_results executable, are provided in Figures 4.5 and Table A.2, respectively.

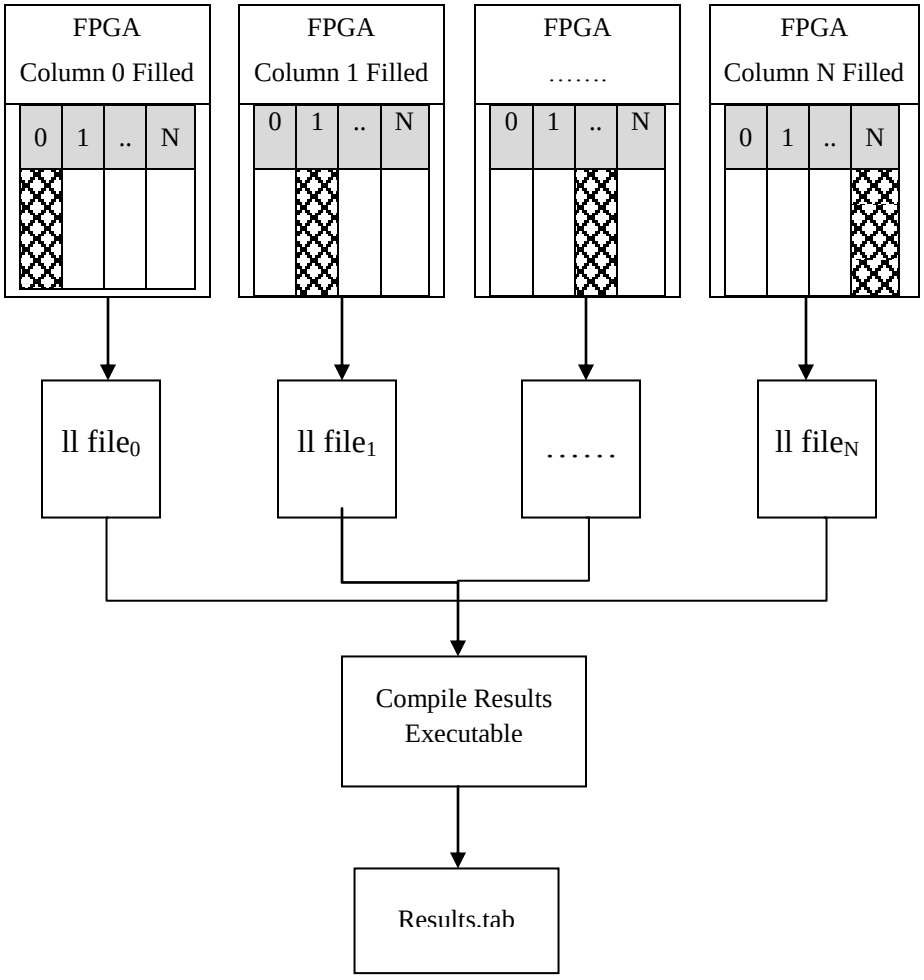


Figure 4.5: Diagram of Logic Allocation File to Tabular Results Process

4.2.2 Automating the Dynamic Data Mapping Process

Map File Generation

To promote simplicity and portability, the only significant input to the automated dynamic data mapping process is a map file that describes the layout of the chip for which the dynamic data is

being mapped. Almost every FPGA differs in the size of their respective CLB arrays. Certain devices also contain auxiliary components, such as “hard core” processors that reside inside the CLB data portion of the chip. For these reasons, it is necessary to know the layout of the chip so that the CLB columns in the device can be filled with the appropriate data to generate correct and complete logic allocation files. To provide this layout to the mapping tools, a map file is generated that contains an outline of the device being mapped. With the use of this map file, the protected areas of the device can be mapped around, and a complete dynamic data map can be generated.

This map file is created by first determining the dimensions of the CLB array for the target device. Next, any CLB regions which are considered “protected” must also be determined. This information can be easily be found using the Xilinx FPGA Editor software [6]. Once this information has been determined, a map file can be generated by describing this information on a column-by-column basis. An example map file for the Xilinx XC2VP30 device is provided in Table A.1.

Generate_Config Executable

The “generate_config” executable is a C application designed to accept design-specific parameters as command line arguments, and generate custom HDL and Xilinx user constraint files. These custom output files are used to instantiate a design capable of filling all flip-flops in a particular CLB column. The design specific input parameters include a device specific topmodule, Xilinx user constraint file and user generated map file. Also, CLB coordinates of the column to be filled by the generated design is input as a command line argument. The usage guide for this software is included in Appendix C.

Logic Allocation File Generation Script

In order to provide portability and increase parameterization, a platform independent bash script was created that utilizes only native bash functionality. This script requires the user to fill in only three parameters: the name of their topmodule / UCF header files (which will be passed to the

gen_config executable), the name of their input MAP file, and the number of CLB columns on the device whose dynamic data they are attempting to map. The only remaining requirement of the script is that a build capable of generating Xilinx bitfiles is present. This build can be easily obtained from [56] and only requires that the Xilinx ISE toolkit (which is free) is installed. Once set up, the script can be used to map the locations of the dynamic data on any Xilinx device (provided a MAP file has been generated for the device) with changes only needing to be made to a few parameters. A sample version of the generate_ll script is shown in Table D.1.

Compile_Results Executable

Prior to the development of the "compile_results" executable, the results of the logic allocation file generation (which contain hundreds of frame addresses and bit stream offsets) had to be manually compiled into a tabular format that could be easily understood. The results of this compilation typically produce a table such as the one outlined in Table B.1. The Xilinx formatted frame address' and bitstream offsets contained in this table then had to be converted into frame addresses relative to the configuration readback bitstream. As a result, a table such as the one shown in Table B.2, was produced. For a particular device, this process, coupled with the manual generation of logic allocation files for each CLB column, proved to be very time intensive and require days to compile. This process also left room for human error. Because the functionality of the configuration integrity checker presented depends on every bit of dynamic data being masked or removed, the occurrence of false positives due to human error is unacceptable. For these reasons, the compile_results executable was developed. This executable iteratively reads though a directory filled with the generated logic allocation files and produces a tabular representation of the data. This table includes Xilinx formatted frame addresses, absolute bitstream offsets, relative bit offsets (to the current frame), as well as the corresponding calculated frame addresses relative to the readback bitstream. This information is needed to develop the data masking FSM used to produce a static readback configuration. An example portion of a table generated for the Xilinx Virtex II Pro XC2VP7 device is shown in Table A.2

4.2.3 Identifying Auxiliary Dynamic Data

Theoretically, the removal of all LUT RAMs, in conjunction with the masking of all flip-flop data from the CLB portion of the readback configuration bitstream, should result in a completely static readback configuration. However, as in most real-world applications, this is not always the case. In practice, it was determined that there are small sections of auxiliary dynamic data that exist outside of the dynamic regions of the configuration outlined in Section 4.1.2. To produce a completely static configuration, which could be used to identify malicious alterations to the configuration, this auxiliary data needed to be identified and masked.

This was achieved by analyzing the Xilinx produced ".msd" file that contains the readback bit locations of all dynamic data Xilinx suggests be masked from the configuration. The locality of the dynamic data in the readback configuration, as outlined in this file, does support the masking procedure outlined in Section 4.2.2, further validating the flip-flop masking procedure previously described. However, the dynamic data map displayed in this file shows that there is dynamic data present in the configuration that is not masked by this procedure.

It has, however, been determined through iterative testing that this Xilinx produced file tends to, "overmask" the readback configuration for dynamic data, identifying portions of the readback configuration as dynamic when this may not be the case 100% of the time. If the dynamic data outlined by Xilinx in this file were taken literally, almost 10% of the CLB section of a given device would be masked from configuration as dynamic data. For this reason, the entire dynamic data map presented in this file cannot be masked, and its contents must be carefully analyzed to determine which portions should be considered valid. If too much of the extraneous dynamic data outlined in the ".msd" file is considered, portions of the configuration that should be monitored for malicious alteration will be left unchecked. If too little of this data is considered, it is possible that dynamic data produced by the system itself could produce false-positives when checking for malicious alterations to the configuration. For these reasons, a middle ground must be established. This middle ground must provide insurance that no dynamic

data will be present in the configuration being checked for alterations, while refraining from masking out potentially static CLB data.

Carefully analysis of the ".msd" file produced for designs on the Xilinx Virtex II / II Pro and Virtex 4 platforms, coupled with extensive testing, produced a layout of this auxiliary dynamic data that must be masked for each platform.

Virtex II and Virtex II Pro Platforms

On the Virtex II and Virtex II Pro platforms, it was determined that masking only the flip-flop bit positions specified inside frames marked to contain dynamic data was not sufficient. While these frames do in fact house all flip-flop data that must be removed from the configuration, they also contain auxiliary dynamic data scattered throughout their configuration data. For this reason, all frames marked to contain dynamic data by the method outlined in Section 4.1.1 were masked.

Virtex 4 Platform

From [2], it can be seen that on the Virtex 4 platform this auxiliary dynamic data is contained in frames outside those marked to contain flip-flip data. Through testing of the readback configuration, this auxiliary dynamic data was determined to be located in the first and last frame of each block of 16 CLBs in the readback configuration.

Hard Core Processor Instantiation

Careful analysis of the ".msd" file also yielded interesting results for devices that contain "hard core" processors built into their configuration. Through this analysis, it was determined that the 135 frame region surrounding the protected CLB space occupied by such a processor contains a significant amount of auxiliary dynamic data. Frames in this region, which do not border frames marked to contain flip flop data, were found to contain approximately 35 consecutive words of dynamic data. Frames adjacent to those containing flip-flop data, however, were found to contain

39 consecutive words of dynamic data. The location of this dynamic data inside these frames varies from device-to-device and platform-to-platform. To promote portability, the device independent version of the configuration integrity checker presented in this thesis was designed to mask out the entire contents of these frames. This can potentially over-mask the configuration data contained in the CLB section. If desired, once the system designer has chosen a device, the configuration integrity checker can be configured to mask the auxiliary dynamic data locations specific to that particular device, alleviating this over-masking.

4.3 Platform Independent Configuration Integrity Checking

To provide a configuration integrity checking solution that provides a system designer the ability to quickly and easily move the system to their desired device and platform, significant changes were made to the design of the PDCIC's components. These modified components, in conjunction with the design of a serial I/O debugging interface, are outlined in Sections 4.3.1 through 4.3.5. When combined, these contributions form a cross-platform configuration integrity checking solution capable of supporting multiple devices spanning multiple platforms.

4.3.1 Active Configuration Readback

Device**s** in and across FPGA families differ in the size of the configuration data that must be actively monitored to determine when a FPGA's configuration has been maliciously modified. From a system portability perspective, the result of this variation is the need for parameterization of the internal FSM controlling the configuration of the FPGA's ICAP. This is done to allow a system designer the ability to easily configure the system to readback configuration data streams of varying sizes. To achieve such parameterization, components, such as the commands issued to the FPGA's ICAP for readback and the control parameters of the internal readback state machine, must be redesigned.

ICAP Readback Commands

Several of the commands needed to setup and perform active readback of an FPGA's internal configuration are device and platform dependent. These platform dependent commands instruct the ICAP of the location in the devices configuration to begin readback (FAR_START_ADR), the number of configuration frames to read (FDRO_LENGTH) and the identification code (IDCODE) of the FPGA being used. In order to provide the system designer the ability to easily reconfigure the state machine issuing these commands, this FSM was designed to allow these commands to be iteratively read from memory. Because this approach was taken, the memory's contents (and therefore the readback commands) can easily be specified in tabular format in the systems HDL. This allows the system designer the ability to easily update these statically defined values to reflect the changes in device id, readback starting position, and readback length that need to be made when moving the system from one device to another. Values such as the number of configuration frames to be readback (which corresponds to the number of device frames contained the FPGA being used) needed to fill these parameters can be obtained from the Xilinx user guide [9,10,12,14] for the family of the device to which the system is being used.

It should also be noted that the ICAP module, provided by the device manufacturer (in this case Xilinx) to interface the FPGA's internal configuration registers, is not device-dependent. It is, however, platform-dependent, and the module corresponding to the platform being used must be instantiated in the configuration integrity checker to provide correct operation of the system. These modules are provided by Xilinx, and their interface needed for instantiation is outlined in the user guide for each available platform [9,10,12,14].

Internal Readback FSM Control Parameters

The only internal readback FSM parameter that needs to be considered when moving the system from one device to another is the number of bytes that will be read back in one scan of the FPGA's configuration. This parameter is denoted as TOTAL_NUM_BYTES in the system's

HDL, and should be equivalent to the value used to represent the FDRO_LENGTH command issued to the FPGA's ICAP to setup active readback of the devices configuration.

4.3.2 Checksum Computation Design and Considerations

To enable portability across multiple devices on multiple platforms, several portions of the design's functionality, including hash value generation and the device specific dynamic data parameters, must be modified.

Hash Value Generation

Moving the system from one device to another results in a change in the number of bytes of CLB configuration data that must be monitored by the configuration integrity checker. As a result, the amount of data that must be processed by the MD5 hash function varies as well. For this reason, when the system is implemented on a new device, the granularity at which CLB data should be processed by the MD5 hash function must be reevaluated.

From Section 4.1.2, it can be seen that a configuration frame is the smallest addressable entity in an FPGA configuration. Because using a granularity smaller then the size of a configuration frame would result in an asymmetry in the structure of the dynamic data masking FSM, it is then assumed that the smallest possible granularity at which data can be processed by the MD5 hash function for a given platform is a single configuration frame. Due to the divergence in the size of configuration frames across different devices and platforms, the amount of data contained in each configuration frame is not constant. While the size of a configuration frame is constant across all devices in the Virtex 4 family (164 bytes), it is device dependent on the Virtex II and Virtex II Pro platforms. For this reason, the number of configuration frames contained in a block of MD5 input data cannot be held constant across FPGA families. This value must then be evaluated on a device-by-device basis. The trade-offs of this design decision are outlined in Section 5.6.1.

For the LX 25 device on the Virtex 4 platform, the suggested number of configuration frames per MD5 input block is 20 frames, resulting in 168 MD5 hashes. On the Virtex II and Virtex II Pro platform, the suggested MD5 input block size is 22 configuration frames. This corresponds to the number of configuration frames contained in one CLB column (which is variable depending on the device chosen), and the number of hashes needed to represent the entire configuration data stream is then equal to the number of CLB columns on the device. These suggested values provide a reasonable trade-off between the amount of resources consumed by the system and the precision at which the locality of malicious attacks can be detected. The results of implementing the system using the granularities suggested in this section are outlined in Section 5.4.2 .

Platform and Device Specific Dynamic Data Masking

Due to the fundamental differences in the locality of dynamic data across different FPGA families, the task of reconfiguring the dynamic data-masking controller to accommodate different FPGA families can be somewhat involved. This dynamic data masking FSM handles the task of monitoring the current frame and byte of configuration data that is being read back. Depending on these values, this FSM is also responsible for adjusting the dynamic data masking parameters accordingly. In order to alleviate the system designer from concerning themselves with making such complex modifications to the dynamic data masking FSM when moving the system across platforms, several platform specific configurations of the dynamic data masking component are provided. The details of how these platform specific implementations of the dynamic data masking FSM select bits to be masked in each platforms readback configuration are outlined in Appendix E. The system now requires the designer only to specify the number of configuration frames on the device being used and number of bytes contained in each frame of configuration data.

4.3.3 Resource Allocation

The cross-platform configuration integrity checker supports two separate configurations that consume two opposing sets of resources. These configurations are provided to give the system designer flexibility in the set of resources required to successfully instantiate the system. The implementation details of these configurations are outlined in Appendix E.

4.3.4 Challenge Response Subsystem

The challenge-response subsystem, included in the PDCIC's design, is integrated into the design's configuration integrity checker component. As previously configured, it was only possible to turn the challenge response system "on" or "off". This prohibits an entity external from having the ability to issue a challenge and receive a response without running a full scan of the device's configuration data as well. This limitation, not only increases the execution time required to scan a FPGA's configuration, but also restricts the functionality of the challenge-response subsystem to being dependent on the status of the integrity checker. In the platform independent version of the integrity checker presented in this thesis, the challenge-response subsystem was moved outside of the configuration integrity checker and is now an independent subsystem of the design. Challenges can be issued to the challenge-response subsystem though the serial I/O subsystem at any point during the system's execution. The response to an issued challenge can be computed regardless of the state of the configuration integrity checker.

Shared Memory Considerations

The system contains a shared hash memory that is read from and written to by the challenge-response subsystem and configuration integrity checker, respectively. The challenge-response subsystem utilizes this memory to read the current hash values of the system's configuration during computation of the secret key needed to compute a response for a given challenge. The configuration integrity checker writes to this shared memory when a new current hash value has

been computed and needs to be stored. To support potentially simultaneous reading and writing, a dual port RAM with support for asynchronous reads was instantiated. This modified architecture of the system is shown in Figure 4.6.

4.3.5 Serial I/O Subsystem

Added in the platform independent version of the configuration integrity checker is a serial I/O subsystem. This subsystem allows a PC connected to the system via a NULL model serial cable the ability to issue challenges and view the subsequent responses visually on the PC's monitor. This provides the capability to receive visual confirmation of the system's health via challenge-response. When using the debugging configuration of the configuration integrity checker, the system designer can also iteratively control the scanning of the FPGA's configuration, as well the start and stop of simulated attacks against the system. Due to the security implications of providing external control of a system's design, this subsystem is intended for debugging use only, and should be disabled before the system is used for regular operation.

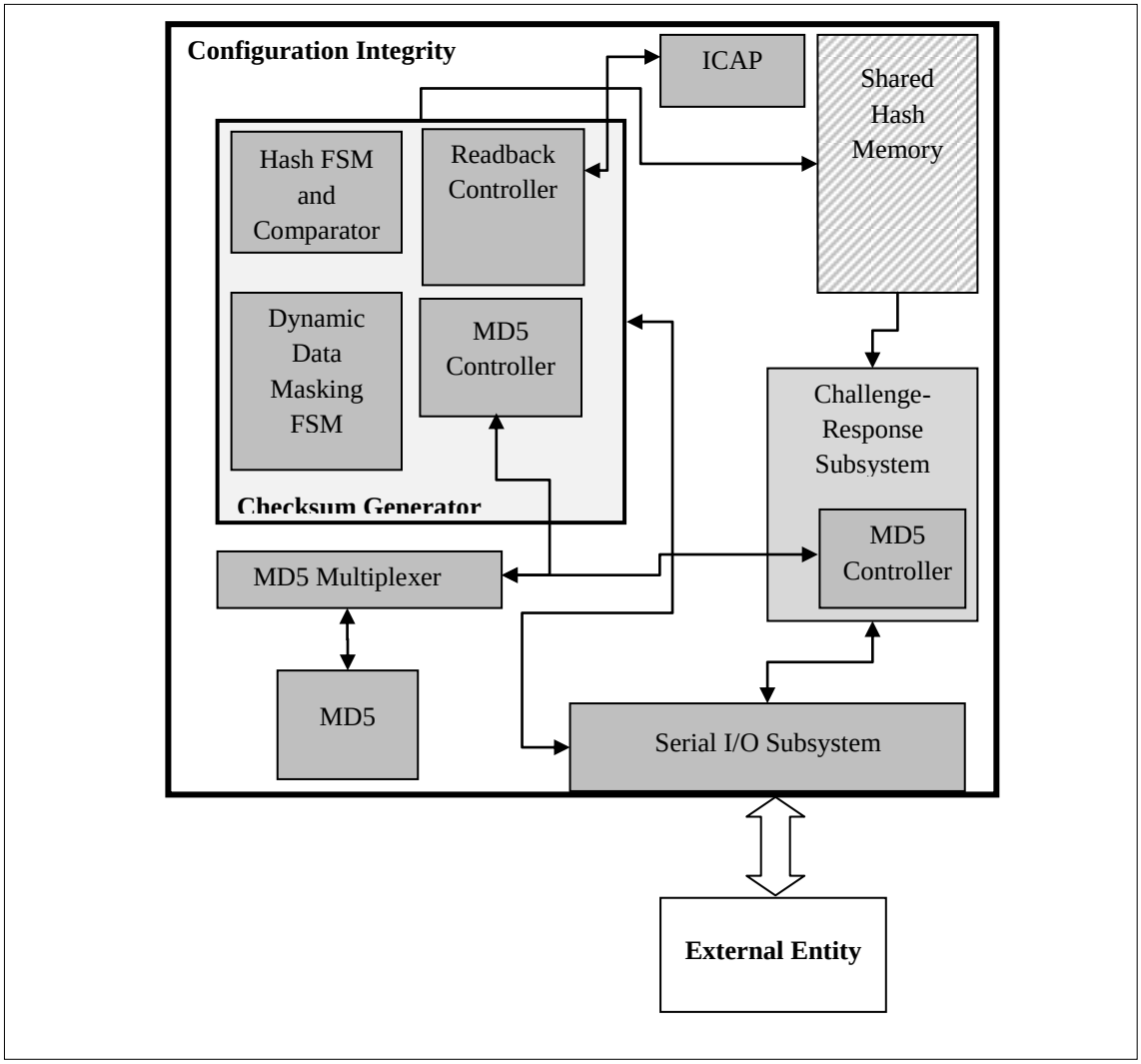


Figure 4.6: Block Diagram of Platform Independent Configuration Integrity Checker

Chapter 5

Validation and Multi-platform Analysis

This chapter first presents the results of previous work upon which the contributions presented in this thesis are based. Next, the results of the validation of the multi-platform configuration integrity checker outlined in Chapter 4 are presented. This system was validated under a series of experiments designed to emulate the semi-invasive attacks for which the system was designed to detect. A detailed resource utilization, latency analysis and security analysis are then presented. The chapter is then concluded with an in-depth multi-platform analysis of the system under the Xilinx Virtex II, Virtex II Pro, and Virtex 4 platforms. When combined, these results demonstrate that a configuration integrity checking solution can be developed, which provides support for significant number of devices, which span multiple FPGA platforms. The results in this chapter are outlined in a similar fashion to those in Chapter 5 of [2] to provide context for comparison.

5.1 Results of Previous Work

Presented in this section is a brief overview of the results of the previous work as described in Chapter 3. The results described in this overview are specific to the LX25 device running on the Virtex 4 platform. A detailed description of the validation of the PDCIC, as well as a complete analysis of the results of this previous work, can be found in [2]. These results are presented to provide a context for comparison between the results of work presented in [2] and those of the cross-platform configuration integrity checker presented in Sections 5.2 - 5.6 of this thesis.

5.1.1 Resource and Timing Analysis

Resource Analysis

The PDCIC described in Chapter 3 consumed 7509 slices on the Virtex 4 platform, which was 69% of the total slice resources available on the LX25 device. In the configuration presented, the system also consumed 4 of the available 72 BRAMs on the device, which was roughly 6% of the total available BRAM resources. Having consumed 69% of the devices available slice resources, only 31% of the devices slices are available to house the design being protected, which is clearly unacceptable. It would then be advantageous to move the design to a larger chip with more resources, however, because the design is platform-and-device dependent, this cannot be easily accomplished. This lack of portability is one of the motivating factors for the design of the device-and platform-independent configuration integrity checker presented in this thesis.

The design of the PDCIC can be broken down subsystem by subsystem to demonstrate the percentage of available resources each consumes. This break down is described in Table 5.1.

Timing Analysis and Critical Path Considerations

The most significant timing constraint in the design of the PDCIC stems from the critical path contained in the MD5 core used for hash computations.

Subsystem	Percent Resources Consumed
Configuration Integrity Checker	70%
Challenge-Response	6%
Partial Authenticator	8%
MD5	16%

Table 5.1: Resource consumption breakdown for the PDCIC

This critical path restricts the configuration integrity checker to a maximum clock frequency of 50 MHz. In the MD5 module used in this design, the critical path appears in the portion of logic that computes the output of a round in a single clock cycle. In order to reduce this critical path and therefore increase the maximum frequency the system can sustain, it was suggested in [2] that the computation that needs to take place in a single clock cycle in the MD5 core be separated into several pipelined stages. Depending on the number and placement of pipelined stages added, the critical path may be significantly shortened. As a result, the system's minimum clock period would be shortened, and thus the maximum clock frequency the system can sustain would be increased.

When operating at 50 MHz, the PDCIC took 80.12µs to scan each of the 168 blocks of CLB data, making the total time to scan all 168 blocks 13.46ms. Including the time required to skip frames that are not included in the CLB hash values, the PDCIC required 16.77ms to scan the entire CLB section.

In order to provide perspective on the effect of the critical path in the MD5 module, the PDCIC's performance was calculated after replacing the MD5 module with a simple XOR / shift checksum computation. If the MD5 module was replaced with this simple checksum calculation, the entire system could instead run at 122MHz. This increase in frequency would allow the PDCIC to scan an entire configuration in 6.8ms.

Because the challenge response subsystem on the PDCIC uses a variable input message size, the time required to compute a response for a given challenge is found in Equation 5.1.

$$T_{challenge} = \left[943+209 \cdot \langle \frac{\#\ bytes\ /\ message}{64} \rangle\right] \cdot 20\ ns \tag{5.1}$$

Security Level Classification

The majority of the attacks for which the PDCIC is susceptible require precise, transistor level modifications to the FPGA's configuration. Also, in depth knowledge of how the system's design is mapped onto the FPGA's configuration is required. As a result, Class 1 and 2 attackers as defined in [8], are not a substantial threat to the system. If Class 3 attackers are properly funded and given sufficient resources, they do, however, have the potential to succeed in design-based attacks as such as those outlined in Section 2.3.2. In the present configuration, the system's defense level would fall into the category of "MOD" as outlined by IBM in [8]. A Class 3 attacker capable of succeeding at such a design-based attack would be referred to as a "knowledgeable insider" [2].

5.2 Platform Independent Dynamic Data Identification Process

The process of mapping dynamic data on an FPGA manually is very time intensive. Because of this, it was very challenging to move the configuration integrity checker presented in Chapter 3 from platform to platform. Mapping this dynamic data for a device on a new platform would require the entire readback configuration layout of this platform to be outlined. This process can take days or even weeks to accurately complete. When combined with the time it takes to manually map and tabulate the dynamic data in the configuration layout of a device, the manual dynamic data mapping process may be an unworthy endeavor for a system designer who intends to move the system to only one particular device.

When the `generate_ll` script is combined with the compile results executable, the security system presented in Chapter 4 of this thesis can be easily moved from device to device and platform to platform. The result is a multi-platform solution capable of easily and efficiently mapping the dynamic data of any Xilinx FPGA. The time required to produce a map of all flip-flop data located in a single CLB column, and entire CLB section on several common devices is shown in Table 5.2.

FPGA Family (Device)	Execution Time (Single CLB Column)	Execution Time (Entire CLB Configuration)
Virtex II (XC2V250)	15.012 sec	8 min, 16.907 sec
Virtex II Pro (XC2VP7)	35.661 sec	35 min, 25.952 sec
Virtex II Pro (XC2VP30)	31.893 sec	47 min, 33.201 sec
Virtex 4 (LX25)	32.124 sec	39 min, 14.748 sec

Table 5.2: Execution time for generate_ll script across multiple platforms

The benchmark results shown in Table 5.2 were obtained from the generate_ll script running on a Gateway NX860X laptop. This laptop employed a 1.83 GHz Intel Core 2 Duo™ Processor. The system also contained 2 GB of external RAM.

From these results it can be seen that time required to produce a dynamic data map of a devices CLB configuration is dependent on the size of the chip being mapped. The time required to produce a map of the Virtex II XC2V250 chip is roughly 6 times less then that of the Virtex II Pro XC2VP30 chip. This is because the XC2V250 chip contains only 1,588,224 configuration bits that must be mapped, while the XC2VP30 chip contains 11,575,552. At a first glance, a map time of upwards of 30 minutes may seem large. However, when compared to the days or even weeks it takes to produce this map by hand this is relatively small. Because this process was automated, the potential chance for an error in calculation is also greatly reduced. This further supports the motivation for the design of this dynamic data mapping process.

5.3 Validation

5.3.1 Testbed

To validate the operation of the multi-platform configuration integrity checker presented in this thesis, a series of experiments were conducted to simulate the attacks for which the system was expected to withstand. These attacks were modeled using a difference based partial reconfiguration strategy. Over several iterations of testing, this method partially reconfigured small portions of the design being protected in several different areas of the protected region in an attempt to demonstrate the systems resistance to simulated fault injection attacks.

To demonstrate that every possible alteration to the FPGA's configuration could be detected, it would be necessary to generate test benches that simulate modifications to every subset of configuration bits on the FPGA. This would be unreasonable, as the portion of the configuration bitstream being monitored is on the order of thousands of bytes. As a result, for all possible combinations of alterations to this set of data to be checked, it would require on the order of millions of iterations of testing to be performed to exhaustively test the system for all possible combinations of modifications. As a compromise, the experiments conducted to validate the operation of the system demonstrate that even the smallest attacks can be detected, regardless of their locality. Using these experiments, it is shown that the configuration integrity checker can detect alterations in the configuration as small as a single bit, in any area on the chip. If modifications of this precision can be detected, by induction it follows that larger modifications can be detected as well.

5.3.2 Radix-4 FFT

When choosing a design to be protected in the testbed, several design properties and performance characteristics were targeted. To demonstrate that the configuration integrity checker was accurately masking out dynamic data, a design needed to be selected that contains a sufficiently large amount of dynamic data that exhibits poor locality. Ideally, the dynamic data

would be spread across the design, making the chances of the configuration integrity checker "accidentally" masking out all of the correct dynamic data very small. The design to be used in the testbed must also be large enough to prohibit there from being hash values computed on NULL data sets. Also, the design used must not constrain the frequency of the system clock used in the configuration integrity checker, or instatiate LUT RAM components in the CLB data space. Finally, the design selected should produce a reasonably large output data set that is being computed at a fairly high frequency. This should be done to provide the opportunity for a malicious attack on the design to be represented as a deviation from the correct output of the design.

The design chosen to be protected in the conducted experiments was a radix-4 Fast Fourier transform (FFT) core obtained from opencores.org. This design was chosen for the testbed because it meets all requirements outlined above. The FFT is, by nature, a block-oriented algorithm. As a result, the FFT operates most efficiently when input and output data samples are processed in parallel. Because FPGAs can provide proprietary data structures which support the concurrency and regularity needed to optimize the FFT algorithm, they have become a popular platform for housing such a design [19]. Because of this, the FFT's design demonstrates resource consumption and performance characteristics common to many FPGA designs, making it an ideal choice for the testbed. When implemented on-chip, the Xilinx optimization tools removed small portions of both the FFT design and the module used to simulate its input. This was deemed acceptable as correct and consistent output values were still being produced by the FFT.

The radix-4 FFT core that was used as the protected design was configured to use a 1024 point, 12-bit FFT input and a 14-bit 4-based (2-bit) reversed ordered output as shown in Figure 5.1. This was done to provide a large enough output data set to reflect malicious alterations to the designs configuration in its output waveform.

$$a_0\ a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9 \longrightarrow a_8\ a_9\ a_6\ a_7\ a_4\ a_5\ a_2\ a_3\ a_0\ a_1$$

Figure 5.1: Radix- 4 FFT output ordering

The input to the FFT was generated on-chip, and was chosen to be a dual sinusoid with constant amplitudes, and a DC offset. The generated input followed the following form:

$$I_{FFT}(t) = A_1 \cdot \sin \omega_1 t + A_2 \cdot \sin \omega_2 t + C_{dc}$$

$$\omega_1 = 2\pi f_1,\ \omega_2 = 2\pi f_2$$

$$A_1 = 100,\ A_2 = 200,\ C_{dc} = 100, f_1 = 100MHz, f_2 = 33MHz$$

To ensure that the FFT HDL was producing correct output, the Mathematica software application was used to compute the theoretical FFT values corresponding to the dual sinusoid input function that was used. Because the output generated by the Radix-4 FFT core is in 2-bit reverse order, output values obtained through simulation of this core using the Xilinx Model Sim simulation environment had to be converted to the proper format. To provide reasonable assurance that correct output was being seen, 1/10, or 102 of the 1024 generated output values per period of the input waveform obtained through simulation were converted to the correct format. These values were then compared with the theoretical output values. Because these converted values are only being used to ensure that HDL provided for the FFT core is performing correct mathematical computations, this subset of output values is large enough to form an output waveform that can be analyzed. As expected, the values obtained through simulation mimicked the theoretical output waveform of the FFT, and the operation of the designs HDL was verified.

To verify that the output generated on chip was correct, the output of the FFT core was obtained using Xilinx ChipScope software. All 1024 output values produced per period were then compared to the output that was generated though the simulation in ModelSim. While some of the output values were slightly off due to the difficulty of modeling the exact input waveform in hardware, the values obtained followed the theoretical output waveform, providing assurance that the FFT core was functioning correctly.

5.3.3 Testbed Standardization

To provide a basis for determining when the integrity of the protected design has been compromised, a standard for analyzing the results of the experiments performed was established. Unsuccessful detection of a malicious attack was defined as the lack of a divergence between a newly generated hash value and the trusted hash value for the block of CLB data that has been maliciously modified. An occurrence of such a scenario would either mean that the static portion of the designs configuration that was maliciously modified is not being included in the hash value computation for a particular block of CLB data, or hash values are being incorrectly computed. As a result, the functionality of the configuration integrity checker would be compromised, and modifications to its design would be required.

A successful experiment was deemed to be one which exhibits the following behavior. Upon the continuous operation of the protected design and configuration integrity checker, consistently correct output is produced by the protected design. The configuration integrity checker also repeatedly computes stable, uniform hash values and its operation can be verified through the challenge-response subsystem. Upon launching a simulated fault injection attack, the configuration integrity checker reliably detects that an attack has occurred and identifies which portion(s) of the design has been maliciously altered. If the attack in question was aimed at disrupting or disabling the configuration integrity checker, this should be detected by the instantiated challenge-response protocol.

The protected design was constrained to the left most 11 CLB columns on the FPGAs used to test the system. This mapping of the system's design to the Xilinx XC2VP30 FPGA's CLB resources can be seen in Figure 5.2. This was done was to segregate the FPGA design space into protected and unprotected regions that allow the system to clearly identify which portions of the chip are considered "secure". This was also done to provide the system designer the option to increase in the configuration integrity checkers performance by only securing the configuration of this 11 CLB wide section. The chosen size of the secured space is variable. Depending on the

device and family of the FPGA for which the system is being implemented, this "secured" area can be modified to meet the custom requirements of a specific FPGA. As previously described in Section 4.3.2, the size of a FPGA column varies from chip to chip. As one would suspect, as the size of the FPGA increases, so does the length of each column of CLBs, making the number of columns needed to house the secured design decrease. Conversely, the opposite holds true as the size of the FPGA is reduced. Because the FPGAs used to perform the experiments outlined were not drastically different in size, and did not lack sufficient space to fit the system on-chip, the constrained area was left uniform across these devices.

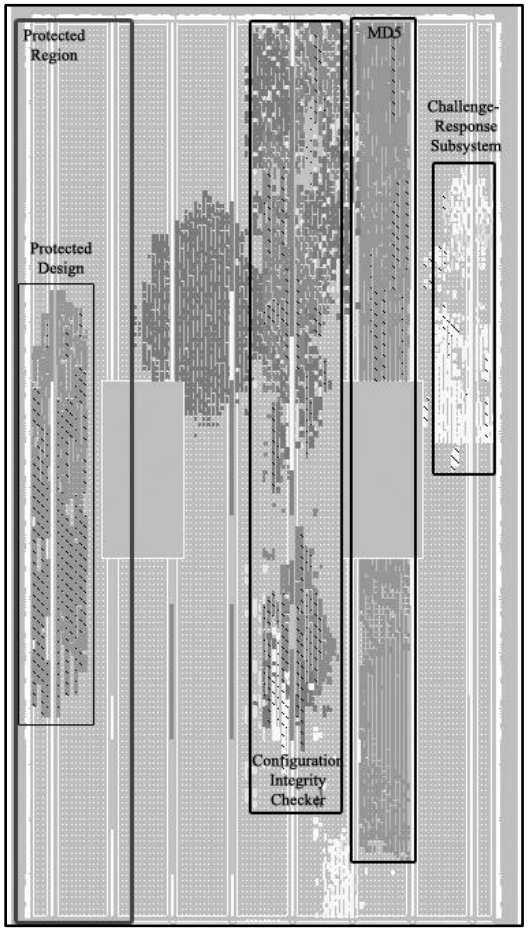


Figure 5.2: Design layout on the Xilinx XC2VP30 FPGA with protected design in secured region

The protected design used approximately 715 slices and 2 BRAMs. The resulting resource percentages across the three different FPGA families can be seen in Table 5.3.

Part					Percentage
XC2V250	Slices	715	of	1536	47%
XC2VP7	Slices	715	of	4928	15%
XC2VP30	Slices	715	of	13696	5%
V4LX25	Slices	715	of	10752	7%

Table 5.3: FFT resource utilization

5.3.4 Results

A series of experiments were performed to demonstrate that the cross-platform configuration integrity checker presented in this thesis is capable of detecting even the smallest alterations to an FPGA's configuration. These experiments simulate semi-invasive, fault injection attacks that could be used by an attacker to alter the configuration of the FPGA. To simulate these attacks, a partial reconfiguration strategy was employed.

In this strategy, a small module used to partially reconfigure an FPGA is instantiated inside the configuration integrity checker. This module consists of two on-chip memories used to store partial bitstreams, and an FSM that issues commands to the ICAP instructing it on how to reconfigure the device. An altered version of the trusted FPGA configuration can be made by making small modifications to the trusted configuration of the system using Xilinx FPGA Editor software. Once these modifications are made, two partial bitstreams can be generated using a difference-based flow provided by Xilinx [5]. These partial bitstreams represent the difference in configuration data between the trusted system, and the configuration of the trusted system that contains the alterations that were made. Using these two bitstreams, a FPGA can be reconfigured to represent the configuration containing the alterations, and then reconfigured back to its original, "trusted" configuration. Once both bitstreams are generated, they can be inserted into

the on-chip memory instantiated in the system's configuration. After the device has been configured with the new design containing these partial bitstreams, it can then be reconfigured to represent the design changes present in the partial bitstreams. This reconfiguration process is controlled through the serial I/O interface contained in the system's design. This interface can be used to dynamically simulate a fault injection attack on the system's configuration.

The first experiment performed simulated a fault injection attack on the protected design located in the secure area of the FPGA. This experiment served to demonstrate that the configuration integrity checker could detect alterations to the configuration of the system, regardless of their size or location. This experiment was composed of several steps. First, the FPGA was initially configured with the correct configuration of the system. After several scans of the FPGA's configuration had been completed (as to model an attack that occurred during a typical computation by the protected design), the FPGA was partially reconfigured to simulate an attack. The portion of the protected design that was altered by the partial reconfiguration did not target any specific component of the design and was changed over several iterations of the experiment. This was done by producing an array of partial bitstreams, each set to reconfigure different areas of the original design with a "maliciously modified" version. Over successive iterations of testing, different partial bitstreams were chosen from this array and inserted into the systems design to be used during the simulated attack. The size of the alterations which were made was held constant and kept as small as possible. Depending on the related logic that was affected by the alterations to the "trusted" designs configuration, between 18-24 frames, in 2-4 blocks of CLBs were altered. It should be noted that the number of blocks of CLBs affected for the same partially reconfigured portion of the design can vary from FPGA to FPGA due to the divergence in the size and quantity of the CLB blocks across FPGA families and devices.

In this experiment, once the partial reconfiguration was complete, the integrity checker successfully detected a change in all altered CLB blocks for each FPGA tested. This experiment was concluded by restoring the previously altered portions of the FPGA back to their original configuration again though the use of partial reconfiguration. Upon successfully reconfiguring

the FPGA back to the correct design, the configuration integrity checker determined that all hash values of the newly reconfigured design match those of the correct design. This marked the successful completion of the first experiment.

Because the FPGA configuration is scanned in iteratively, beginning at the bottom left corner and ending in the top right corner of the device, the time required to detect an alteration depends on the locality of the region effected by the change and the granularity at which the configuration is being scanned. The minimum, average, and maximum latencies for detecting an alteration to the configuration on each platform tested are shown below in Table 5.4.

Platform (Device)	Minimum	Average	Maximum
Virtex 4 (LX 25)	.099 ms	8.32 ms	16.77 ms
Virtex II Pro (XC2VP7)	2.18 ms	30.52 ms	74.12 ms
Virtex II Pro (XC2VP30_Constrained)	2.18 ms	10.9 ms	23.9 ms
Virtex II Pro (XC2VP30)	2.18 ms	50.14 ms	100.0 ms

Table 5.4: Minimum, average, and maximum scan times required to detect a malicious alteration to the FPGAs configuration

From Table 5.4, it can be seen that the ratio of the minimum to maximum scan times required to detect a change on the Virtex 4 platform is approximately 0.005, while on the Virtex II Pro platform this ratio is 0.025. This is due to the increase in checking granularity used on the Virtex 4 platform. From Table 5.4, it should also be noted that by constraining the protected design to a smaller protected area on the XC2VP30 device, the average and maximum times required to detect a change are almost 5 times less then when the entire configuration is scanned.

The next experiment conducted was used to validate the operation of the challenge-response subsystem. In this experiment, malicious alterations were made to the protected design in the same fashion as the first experiment, however, this time, prior to launching these simulated

attacks, several challenges were sent to the challenge response subsystem though the use of the serial I/O control interface. Challenges sent each included a fixed-length private message that was sent to the challenge-response subsystem as described in Section 4.3.4. All challenges sent were met with a successful response that included a correct hash value of both the private message sent and a hash of each individual block of CLBs corresponding hash value. After simulating an attack (again though the use of partial reconfiguration), another challenge was sent. This time, the response contained a incorrect hash value, indicating that the hash value of at least one block of CLBs had been changed (because the response hash value is a hash of the private key and a hash of all CLB block hash values). This successfully completed the second experiment.

Because the length of the challenge sent to the verifier in the system is held constant, the time required to produce a response (and therefore determine if the system has been compromised) is solely dependent upon the granularity at which the system's configuration is being checked. As the granularity at which the system is scanned is increased, so is the number of hash values that must be included in the computation of the secret key needed to produce a response for a given challenge. On the Virtex 4 platform, the granularity at which the system is scanned is roughly 4 times that of the Virtex II and Virtex II Pro platforms. As a result, the time require to produce a response for a given challenge on the Virtex 4 platform is 85.5 µs, while the time required to produce a response on the Virtex II / II platform is only 23.42 µs.

A final experiment was used to validate the system's capability to alert an external entity that the functionality of the system has been compromised. In this experiment, a portion of the system that would potentially cause the integrity checker to stop functioning, was targeted. More specifically, the attack was targeted at the heart of the MD5 module used to compute the hash values. These hash values are needed to validate the integrity of each block of CLBs in the protected design. Typically, attacks that target such a critical area of the system successfully compromise the system's functionality at no less than a 50% success rate. Upon simulating this attack (again though the use of partial reconfiguration), the configuration integrity checker

immediately ceased to function. When a challenge was sent to the system, no response was received and it was clear that the system had been compromised. This successfully validated the operation of the challenge-response subsystem and completed the array of experiments verifying the operation of the configuration integrity checker.

5.4 Performance Analysis

5.4.1 Resource Utilization

Detailed outlines of the resources consumed by the system for each of the two provided configurations are included in Tables 5.5 and 5.6. These configurations were provided to give the system designer flexibility as to which set of available resources would be most beneficial in instantiating the configuration integrity checker. In the first configuration, all memories instantiated in the integrity checker are comprised of LUT RAMs. As can be seen in Table 5.5, this configuration only consumes 4 BRAMs, however, total number of slices consumed is over 1.5 times greater than that of the second configuration, resulting in 7477 slices being consumed. The second configuration consumes only 4960 slices, shifting the majority of the resource utilization to the FPGA's BRAMs, with 43 BRAMs being consumed.

Resource	**Amount Used**
Slices	7477
Slice Flip Flops	7001
4 input Look Up Tables (LUTs)	9044
I/Os	12
BRAMs	4
GCLKs	2

Table 5.5: System resource utilization under reduced BRAM consumption configuration

Resource	Amount Used
Slices	4960
Slice Flip Flops	4345
4 input Look Up Tables (LUTs)	8036
I/Os	12
BRAMs	43
GCLKs	2

Table 5.6: System resource utilization under reduced slice consumption configuration

The system was implemented on three FPGAs that span two families. These include the LX25 part on the Xilinx Virtex 4 platform, and the XC2VP7 and XC2VP30 parts on the Xilinx Virtex II Pro platform. From Figures 5.3, 5.4 and 5.5, the percentage of slice resources consumed by both configurations of the system for each part in the Xilinx Virtex II, II Pro and 4 families can been seen. On the Virtex 4 platform using the LX25 chip, the system consumed 69% of the available slices for the BRAM minimization configuration, and 46% of the available slices in the slice minimization configuration. On this platform, neither configuration would be acceptable for the LX25 part. Over 46% of the available system resources were used for the configuration integrity checker, leaving approximately 54% of the chips available slices for the system designer to implement their design. An acceptable slice utilization for the integrity checker would be under roughly 25%. This would make the LX40 the smallest acceptable chip for the slice minimization configuration, and the LX60 the smallest acceptable chip for the BRAM minimization configuration. When implemented on the Virtex II Pro platform, the system was instantiated on the XC2VP7 and XC2VP30 FPGAs. When implemented using the BRAM minimization configuration, the integrity checker required 152% and 54% of the FPGA's available slices, respectively. When set to the slice minimization configuration, the checker used 101% and 36% of the available slices, respectively. Again, these resource utilizations are unacceptable, as the checker is better suited for a mid-to-large size FPGA when implemented on the Virtex II Pro platform. From Figure 5.6, it can be seen that the smallest FPGA that would

satisfy the space utilization requirement would be the XC2VP40 used in the slice minimization configuration. Because the dynamic data mapping and configuration readback procedure of the Virtex II platform is virtually identical to that of the Virtex II Pro (as the Virtex II Pro was derived from the Virtex II), the integrity checker was not tested on the Virtex II platform. If it was desired to utilize the checker on such a platform, it can be seen from Figure 5.5 that the smallest part that would appropriately house the design would be the XC2V4000 used with the slice minimization configuration.

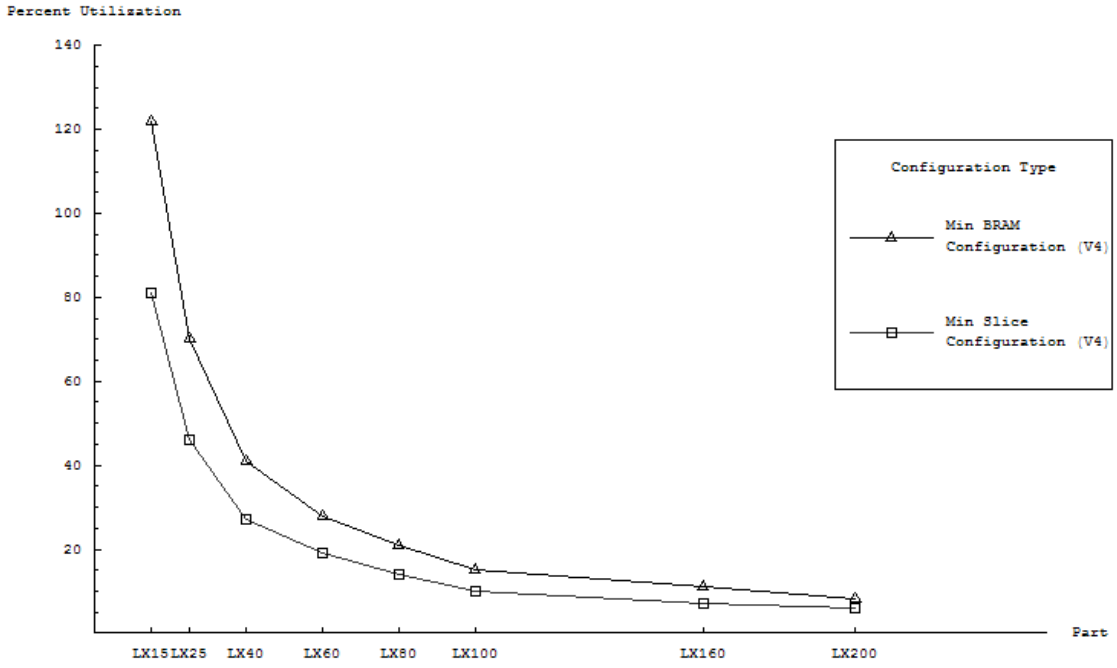


Figure 5.3: Virtex 4 system slice utilization

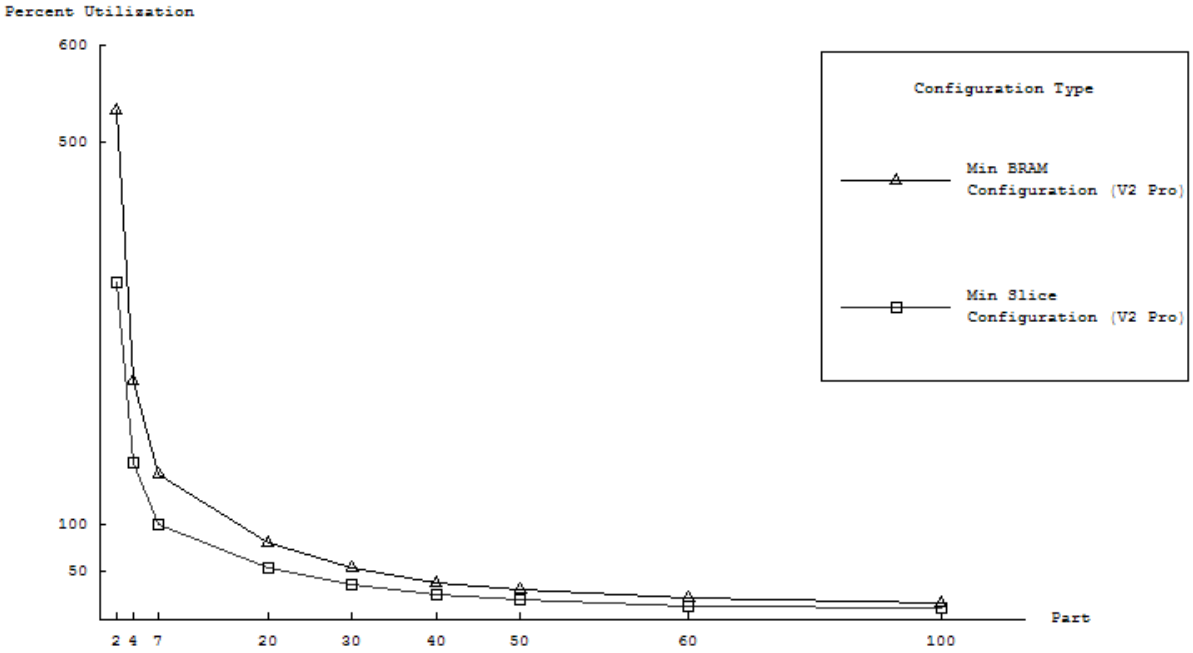


Figure 5.4: Virtex II Pro system slice utilization

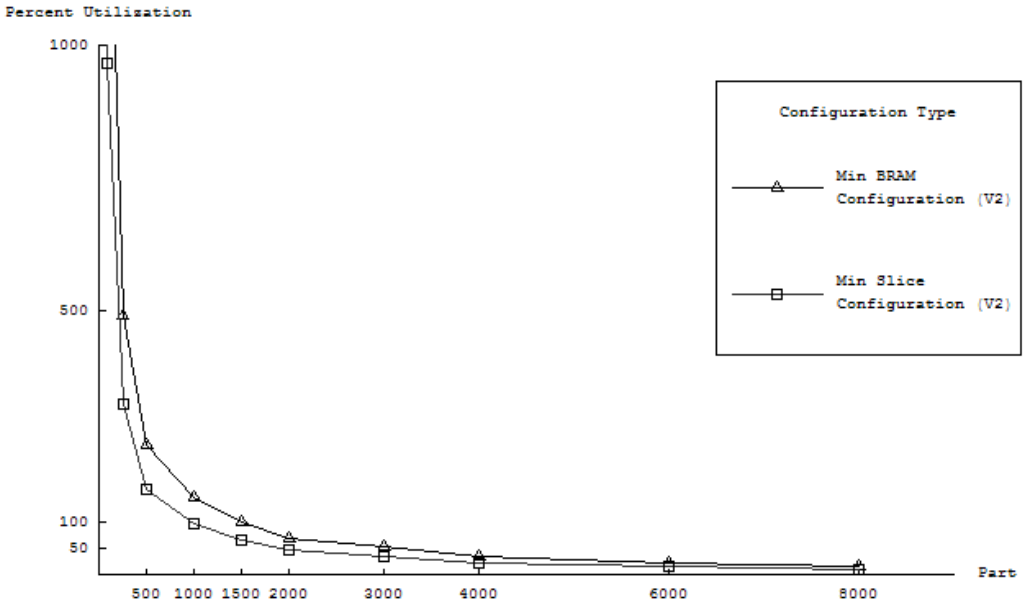


Figure 5.5: Virtex II system slice utilization

A breakdown of the resources consumed by each subsystem in the configuration integrity checker is outlined in Figure 5.6. This breakdown can be obtained by analyzing the synthesis report generated by the Xilinx XST synthesis tools. By combining the slice and BRAM resources consumed, an estimate of the total resources consumed was obtained.

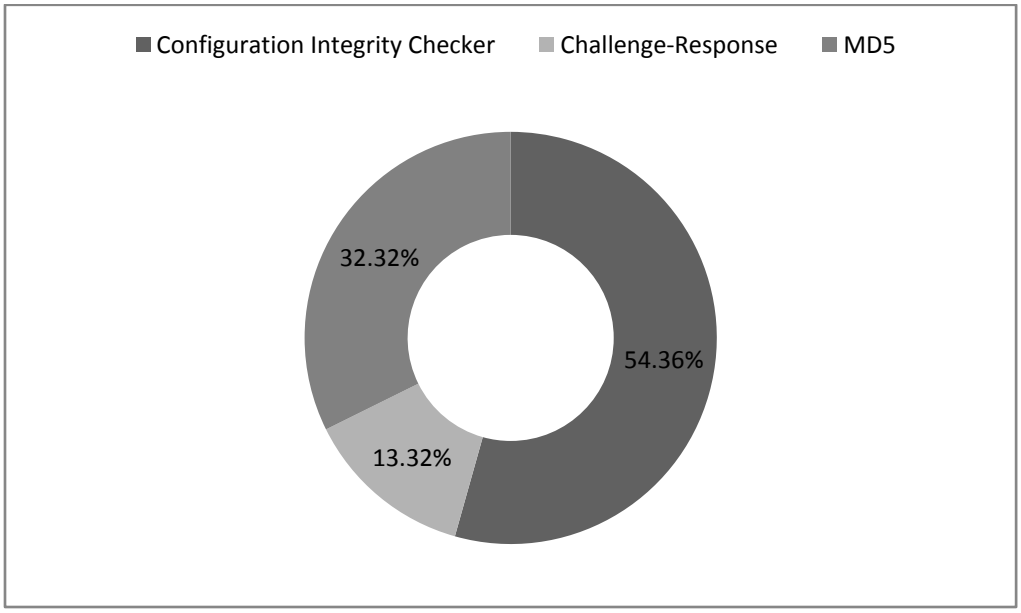


Figure 5.6: Design utilization percentage breakdown by subsystem

In the utilization breakdown in Figure 5.6, the system is broken down into three subsystems. The largest subsystem is the configuration integrity checker, which consumes 54% of the total design's resources. This subsystem contains several large FSMs which control configuration readback, as well as the data masking, and MD5 controllers. The second largest subsystem in the design was the MD5 module, requiring 32% of the design's total resources. The smallest subsystem in the design is the challenge-response module. This module contains only one FSM, which controls the MD5 module used to generate the hash value presented as the response to a challenge.

5.4.2 Multi-platform Latency Analysis

Subsystem Execution Time Analysis

In this section, both the configuration integrity checker and the challenge-response subsystem's timing performance are analyzed. For each subsystem, the number of clock cycles needed to fully complete their respective functions were computed for both the Xilinx Virtex 4 and Virtex II Pro platforms. The Virtex II platform was not included in this analysis, as its fundamental design concepts are nearly identical to those of the Virtex II Pro platform.

Configuration Integrity Checker Subsystem

In order to accurately measure the latency of the integrity checker on both the Virtex 4 and Virtex II / II Pro platforms, the operation of the configuration integrity checker must be analyzed. First, the smallest repeatable function(s) on each platform was identified. Once the operation time of this function was determined, it was then replicated accordingly to produce a total timing estimate for the system. Finally, the total configuration scan times for each platform were determined, and an analysis of the resulting latency was performed.

On the Virtex 4 platform, it was determined that the smallest repeatable functions needed to compute a total timing estimation were the latency of computing a hash value of an entire block of CLBs as well as the latency incurred in skipping a single frame of readback data. On the Virtex 4 platform, each block of CLBs took 4,006 clock cycles to compute and each skipped frame required 492 cycles. With the Virtex 4 LX25 FPGA under analysis containing 168 blocks of CLBs, the total time required to scan an entire configuration derived below in Equations 5.2 through 5.5.

$$T_{V4\_configuration} = T_{CLB\ Blocks} + T_{Frames\ Skipped} \tag{5.2}$$

$$T_{CLB\ Blocks} = T_{block} \cdot \#Blocks \tag{5.3}$$

$$T_{cycle} = \frac{1}{freq} = \frac{1}{50\ x\ 10^6} = \ 0.02\ \mu s \tag{5.4}$$

$$T_{block} = 0.02\ \mu s\ \cdot\ 4006\ cycles\ = 80.12\ \mu s$$

$$T_{CLB\ Blocks} = T_{block}\ \cdot \#Blocks \tag{5.5}$$

$$T_{CLB\ Blocks} = 80.12\ \mu s\ \cdot\ 168\ blocks$$

$$\boldsymbol{T_{CLB\ Blocks} = \ 13.46ms}$$

With there being 336 skipped frames, and the number of cycles required to skip one frame being 492 cycles, the total time to process all skipped frames is shown in Equation 5.6.

$$T_{skipped\ frame} = 492\ clock\ cycles\ \cdot\ T_{cycle} = 9.84\ \mu s$$

$$T_{Frames\ Skipped} = \ T_{skipped\ frame}\ \cdot \#\ skipped\ frames \tag{5.6}$$

$$T_{Frames\ Skipped} = \ 9.84\ \mu s\ \ \cdot 336\ frames$$

$$\boldsymbol{T_{Frames\ Skipped} = 3.306\ ms}$$

The total time required to scan the entire configuration would then be the sum of the time required to process all CLB blocks and the time required to skip all 336 frames. The resulting time is:

$$T_{configuration} = 13.46ms + \ 3.306\ ms$$

$$\boldsymbol{T_{V4\_configuration} = 16.77ms\ @\ 50\ Mhz.}$$

On the Virtex II platform it was determined that the only repeatable function needed to compute a total timing estimation was the latency of computing a hash value of an entire block

of CLBs. Each block of CLBs took 109,051 clock cycles to compute when running under the Virtex II Pro platform. With the Virtex II Pro XC2VP30 FPGA under analysis containing 46 blocks of CLBs, the total time required to scan an entire configuration is shown in Equation 5.7.

$$T_{VIIPro\_configuration} = T_{CLB_{block}} \cdot \#CLB\_blocks \tag{5.7}$$

$$T_{cycle} = \frac{1}{freq} = \frac{1}{50\ x\ 10^6} = 0.02\ \mu s$$

$$T_{CLB\_block} = \frac{\#cycles}{CLB\ block} \cdot T_{cycle}$$

$$T_{CLB\_block} = 109051 \cdot 0.02\ \mu s$$

$$T_{CLB\_block} = 2.18\ ms$$

$$T_{VIIPro\_configuration} = 2.18\ ms \cdot 46\ blocks$$

$$\boldsymbol{T_{VIIPro\_configuration} = 100.0\ ms\ @\ 50\ MHz}$$

If the configuration integrity checker's configuration was set to only scan the protected area of the chip (which as mentioned in Section 5.3.2 is located in the first 11 CLB columns), the resulting time to scan an entire configuration is:

$$T_{CLB\_block} = 2.18\ ms$$

$$T_{VIIPro\_protected\_configuration} = 2.18\ ms \cdot 11\ blocks$$

$$\boldsymbol{T_{VIIPro\_protected\_configuration} = 23.9\ ms\ @\ 50\ MHz.}$$

To provide reference to the analysis, which was done on the performance of the PDCIC in Section 5.1, the system's performance was calculated using a simple XOR / shift checksum computation. The increase in frequency provided by this substitution would allow the configuration integrity checker on the Virtex 4 platform to reduce its time to scan an entire configuration by 244%, yielding a computation time of 6.8ms. On the Virtex II Pro platform, the entire configuration could be scanned at 40.98 ms using the default configuration, and at 9.78 ms under the high performance configuration.

To provide a context for comparison across the Virtex 4 and Virtex II Pro families, the times required to scan an entire configuration were calculated for several parts in each family that could potentially house the system. The results of these computations are shown in Figure 5.7.

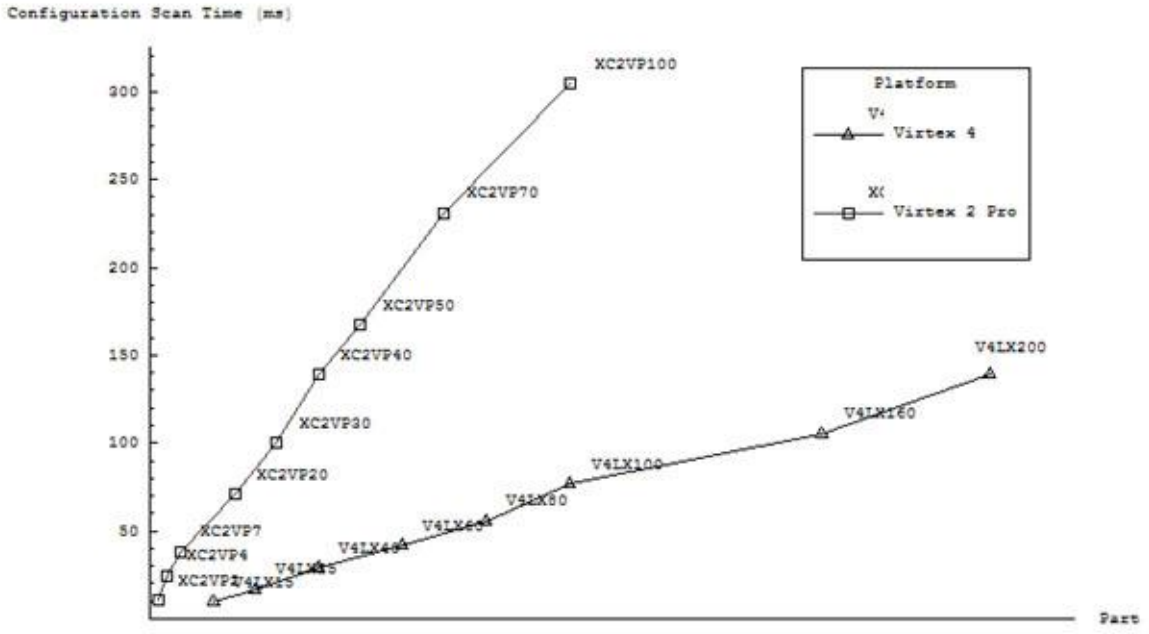


Figure 5.7: Time required to scan a entire FPGA configuration for the Virtex 4 and Virtex II Pro Platforms

Challenge-Response Subsystem

Due to the serial interface used to accept the challenge from the user, the challenge length is fixed at 128 bits. The execution time of the challenge-response protocol is then only dependent upon the number of CLB blocks in a FPGA's configuration. In Equation 5.8, the number of clock cycles required to compute a response is shown.

$$\#\ clock\ cycles = 1171\ \cdot \frac{\#\ CLB\ blocks}{46} \tag{5.8}$$

The constants in the equation represent the number of cycles required for the challenge-response FSM to operate. Once the number of clock cycles has been obtained, the number of seconds required to perform the computation can be obtained by multiplying this value by 20 ns.

$$execution\ time = \left[1171\ \cdot \frac{\#\ CLB\ blocks}{46}\right] \cdot 20\ ns$$

Using the LX25 part on the Virtex 4 platform, there are 168 CLB blocks, making the execution time:

$$execution\ time_{V4LX25} = \left[1171\ \cdot \frac{168}{46}\right] \cdot 20\ ns$$

$$execution\ time_{V4LX25} = 85.5\ \mu s.$$

Using the XC2VP30 part on the Virtex II Pro platform, there are 46 CLB blocks, making the execution time:

$$execution\ time_{XC2VP30} = \left[1171\ \cdot \frac{46}{46}\right] \cdot 20\ ns$$

$$execution\ time_{XC2VP30} = 23.42\ \mu s.$$

Again, when using the the simple XOR / Shift algorithm instead of the MD5 core, the system can run at 122 MHz, resulting in execution times for the LX25 and XC2VP30 of 35.1 µs and 9.6 µs, respectively.

5.5 System Security Analysis and Classification

While the system presented in this thesis has made substantial changes to the components it was founded upon to promote portability, the overall structure and organization of these components remains intact. As a result, the level of security the system provides is congruent to that of the system presented in [2]. At this security level, the system is resistant to Class 1 and 2 attackers due to their inability to launch precise, bit-level, fault injection attacks. A Class 3 attacker has the potential of succeeding at such a design-based attack if proper knowledge of the device being attacked is obtained. The tools necessary to make these precise, bit-level modifications to the FPGA's configuration would also be necessary. Once obtained, such an attacker could use their ability to make these strategic modifications to the devices configuration to compromise the systems security. This can be done by either (a) determining information about the system by monitoring the response to said changes in between configuration scans, or (b) attempting to render portions of the system inoperable to gain an advantage against its design. An attacker at compromising the system is this manner is known to be a, "knowledgeable insider" [2].

The only change in the system that could potentially cause a deviation from this security rating is a result of the change to orientation of the challenge-response subsystem relative to the overall structure of the design. Because the challenge response subsystem was made to operate as a stand-alone component in the newly designed system, it now responds to challenges asynchronously relative to the configuration integrity checker. As a result, an attacker can no longer be assured that a response to a given challenge will only come at the end of configuration block scan. This leaves the attacker only 85.5µs and 23.42µs on the Virtex 4 and Virtex II / II Pro platforms, respectively, to attempt to compromise the system without taking the risk of potentially being detected by the challenge-response subsystem. Even with this augmentation to

the challenge-response subsystem, the system's defense level would still fall into the category of "MOD" as outlined by IBM in [8].

5.6 Platform Analysis and Considerations

In evaluating the system's performance on both the Virtex 4 and Virtex II Pro platforms there were several significant factors that separated the two. The two most prominent factors that caused a discrepancy in performance between platforms were the granularity of the configuration integrity checker and the supporting memory system.

5.6.1 Granularity Considerations

Due to the difference in configuration layout between the Virtex 4 and Virtex II / II Pro platforms, both the size of input blocks to the MD5 algorithm and the number of hashes that were generated varied. On Virtex 4 platform, the hash computation granularity was set to be block of 16 CLBs for the LX25 device, with each block containing 20 frames at 164 bytes per frame, or 3328 bytes of configuration data per block on the V4LX25. The Virtex II / II Pro platform's hash computation granularity was set at a CLB column that contains 22 frames. At 824 bytes per frame, each block contains 18176 bytes of configuration data on the XC2VP30. This variation between block size results in the configuration of the LX25 chip on the Virtex 4 platform being represented by 168 hashes, while only 46 hashes are needed to represent the XC2VP30 chip on the Virtex II Pro platform. Therefore, it is clear to see that even though these two chips are roughly the same size, with the LX25 containing 10752 slices and the XC2VP30 containing 13696 slices, the number of hashes that must be computed is almost 4 times greater on the LX25. It would be possible to adjust the granularity used in either case, however, as mentioned in Section 4.3.2 the granularity chosen for each platform provides equilibrium between the amount of storage space required to store the resulting hash values, and the precision at which the configuration integrity checker can report areas which have been maliciously altered.

The effects of such a skew in the number of hash values that must be computed are seen in both the device utilization and execution times of each platform. As a result, there is an increase the configuration scan time on the Virtex 4 platform due to overhead from MD5 hash function computation. In addition, a reduction in device utilization is seen on the Virtex II / II Pro platform due to the decrease in hash memory size and supporting logic that is needed.

5.6.2 Memory Architecture

The majority of the increase in execution time suffered on the VII / VII Pro platform is due to the change in memory structure. In order to provide designers with a system configuration more suitable for chips with smaller slice counts (on older platforms with smaller chips) the slice utilization on the VII / VII Pro platform was reduced by moving the instantiated memories into the on-chip BRAMs. This resulted in a slice reduction of approximately 25% from that of the design used on the Virtex 4 platform. The penalty for such slice reduction was both an increase in BRAM utilization and increase in the amount time required to read and write to and from memory. The extra clock cycles required to successfully read from and write to such a shared memory (the hash memory is shared between the integrity checker and challenge-response subsystem) account for the remaining difference in execution time.

The resulting increase in execution time does not significantly affect the level of security the system can provide in relation to brute force attacks, as the time that is required to compute a second preimage on the Virtex II / II Pro platform would still be well beyond the life cycle of this design. This increase in execution time would, however, affect protection against design-based attacks (which the system was previously susceptible to under Class 3 attackers) targeted at making alterations to the systems configuration in between configuration scans, and would provide the attacker with an extra 83 ms of time in between scans.

5.6 Summary of Results

By successfully securing the radix-4 FFT design on several platforms, under an array of simulated fault-injection attacks, it was shown that the configuration integrity checking solution presented provides a viable cross-platform solution capable of successfully securing real world applications. During the validation of the system, it was found that the precision at which the configuration integrity checker is capable of successfully and reliably detecting alterations, is on the order of bits. This ability to consistently detect such minute and precise alterations to the protected configuration provides a solution that is capable of neutralizing both Class I and II attackers. The results of the timing and utilization analysis performed on the system show that, while the system does not demonstrate exceptionally low latency or device utilization, it does provide sufficient performance to be instantiated on 14 modern FPGAs across the Virtex II, Virtex II Pro, and Virtex 4 families. As a result, the system presented, while still having room for improvement, is a portable security solution for many applications in today's computing industry.

Chapter 6

Conclusion

6.1 Summary

The objective of this thesis was to develop a portable and parameterizable method for checking the integrity of FPGA configurations across multiple platforms. This objective was accomplished through the combination of several components. The first component was a multi-platform method for masking dynamic flip-flop data from FPGA readback configurations. In addition, the configuration readback, hash generation and challenge-response subsystems presented in [2] were extended across multiple platforms to form the second component of the design. The final component in the system's design was a flow used to automatically generate the relative readback bitstream locations of all dynamic flip-flop data present on any device in any Xilinx FPGA family. To supplement these components, two configurations of the system that consumed opposing sets of resources were provided. These configurations were provided to give the system designer flexibility as to which resources needed to be kept available for the design being secured by the system. To promote portability, two platform-dependent configurations of the dynamic data masking subsystem were provided. By making such configurations available, a

system designer instantiating the configuration integrity checker presented is alleviated of the complex parameterization that is typically required when moving the system from one platform to another.

The components of the cross-platform configuration integrity checker presented were designed and tested on the Xilinx Virtex II platform using both the XC2VP7 and XC2V30 devices and on the Virtex 4 platform using the LX25 device. The system was tested using several experiments that simulated several types of fault injection attacks. In each experiment, the system successfully detected that an attack was being made and reported the failing hash values corresponding to the location(s) on the chip that were targeted by the attack.

A cross-platform analysis was performed on the system to demonstrate its usefulness when deployed across multiple devices on multiple platforms. The results of this analysis showed that on each of the Virtex 4 and Virtex II / II Pro platforms there exist 7 devices in which the system required less than 25% of the total available resources. As a result, system designers are provided 14 possible devices spanning 3 platforms which can be used to implement the system and still provide over 75% of the FPGAs total resources to house the design being protected.

Finally, a security analysis was performed on the system to classify its resistance against potential attacks. Attacks that presented the capability of compromising the system were also classified. The results of this analysis showed that the system maintained the same level of resistance to attacks as the system outlined in [2], from which several of its components were derived. This analysis showed that the system was resistant to Class 1 and 2 attackers as defined in Section 2.2, however Class 3 attackers showed potential to compromise when using precise bit level fault injection attacks.

6.2 Future Work

Several specific areas were identified where the system presented shows room for improvement. The first significant area identified involves addressing the security threat presented from scanning the configuration iteratively with configuration scan times on the order of tens to hundreds of milliseconds. The second area that could be addressed through future work is the design assumption that the only dynamic data present in the CLB portion of the FPGA's configuration belongs to flip-flop data. Additionally, the assumption that the system is not being attacked on the first iteration of execution could be addressed. A final area of future work that could be addressed is securing the I/O subsystem.

6.2.1 Addressing Security Weaknesses Due to Extensive Scan Times

Random Block Readback Strategy

While a potential attack point of the system could be the time it takes in between scans of a configuration to produce and compare the hash value of a given block of CLBs, this weakness could be offset by altering the configuration integrity checker to compute hash values of each block of CLBs randomly, instead of iteratively across the chip. The attacker was previously relying on the assumption that once scanned, the same block of CLBs would not be scanned again until the rest of the configuration had been scanned. If blocks are scanned randomly inside of each scan of an entire configuration, there exists the possibility that a block could be scanned twice in succession. Thus, the expected time an attacker would have to make alterations and then restore the configuration to its previous state would be reduced to slightly above the time it takes to scan one CLB block (80 µs on the Virtex 4 platform and 2.18 ms on the VII / II Pro platforms). In Figure 6.1, a sample scenario of the same CLB block being read back consecutively is shown.

Figure 6.1: Random readback successive block readback scenario

Potential attackers would then be limited by the size of the malicious changes they can make as well as the speed in which they must make them. To implement such a strategy, the readback control FSM of the system would have to be redesigned, incorporating random number generation module into the system.

Improving the Hashing Algorithm

The extensive configuration scan times are due in part to the critical path present in the MD5 hash algorithm used in the design. Future work could address the issue of removing this critical path or finding an auxiliary hash computation method.

6.2.2 Design Assumptions Regarding Dynamic CLB Data

In its current configuration, the system assumes that all dynamic data located in the CLB section of the FPGA's configuration is produced by flip-flops contained in the user's design. This assumption is needed because the dynamic data masking strategy currently employed is only capable of masking dynamic data produced by flip-flops present in the system. This prohibits the system designer from creating designs that contain components which are synthesized into LUT-RAMs. A LUT-RAMs is a CLB, or group of CLBs configured to function as a RAM. An example of a popular design that uses LUT RAMs extensively is the Xilinx Chipscope hardware-debugging module. If such a module was to be protected by the configuration integrity checker presented dynamic data would appear in the CLB section of the design which was not a result of flip-flop data and the functionality of the configuration integrity checker would be disrupted. If the contents of the LUT RAMs that are instantiated in the design are not security sensitive, they

could be constrained to be located in a specific section of the FPGA. This area could then be removed from portion of the readback configuration bitstream that is being protected by system, thereby allowing the system to function normally. If the data contained in this LUT RAMs was in fact security sensitive, a new solution would have to be designed.

6.2.3 Trusted Hash Value Computation

Presently, it is assumed that the configuration integrity checker is not maliciously attacked during the first time it scans the configuration of the device. This assumption is in place because the system must compute, "trusted" hash values of the readback configuration bit stream during this iteration. This assumption poses a security threat to the system if an attacker did make a malicious modification to the FPGA's configuration during the first scan of the configuration. In this scenario, one of the trusted hash values would be a representation of the design including the attacker's modifications. This would make not only the response computed for a given challenge incorrect, but all subsequent hash values generated for that section of the FPGA's configuration would be compared against this incorrect value. To alleviate this problem the trusted hash values should be computed off-chip. This can be achieved by masking the expected readback data (*.rbd) using the readback mask file (*.msd) and computing the resulting trusted hash values from this bitstream. The .rdb and .msd files can be generated by the Xilinx bitgen command using the"-g readback" and "-m" options. These trusted hash values can then be statically included in the system prior to it being loaded on to the FPGA.

6.2.4 Securing the I/O Subsystem

If currently used during regular operation, the I/O subsystem used to transfer/receive challenges to/from the system presents a potential security threat, and measures would need to be taken to secure its operation. These measures could include encrypting the data being transferred through the system, as well as providing a password requirement to use the off-chip interface.

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] B. Webb, "Methods for Securing the Integrity of FPGA Configurations", Masters Thesis, Virginia Polytechnic Institute and State University, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, August 2006.

[3] 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.

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

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

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

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

[8] 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.

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

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

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

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

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

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

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

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

[17] 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.

[18] Z. Ming, "CFFT A New Radix 4 Complex FFT Processor." Opencores.org, September 2003, www.opencores.org.

[19] R. Hosking, “FPGA-Based FFT Engine Handles Four Times More Input”, in *EE Times*, August 2002.

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

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] G. McGregor, D. Robinson, and P. Lysaght, “A Hardware/Software Co-design Environment for Reconfigurable Logic Systems,” in *Proceeding of FPL’98, September* 1998, pp. 258–267.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] J. P. Graff, "A Key Management Architecture 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.

[42] I. Hadˇzi´c, S. Udani, and J. M. Smith, "FPGA Viruses," in *FPL*, 1999, pp. 291–300.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

[55] K. Siozios, D. Soudris, A. Thanailakis, "A novel methodology for designing high-performance and low-power FPGA interconnection targeting DSP applications", in IEEE *International Symposium on Circuits and Systems (ISCAS'06)*, May 2006, pp. 21–24.

[56] *XST User Guide,* Xilinx, June 2003, version 4.0.

[57] 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.

[58] 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.

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

[60] 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.

[61] K. Paulsson, M. H¨ubner, 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

[62] M.B. Blanton, "An FPGA Software-Defined Ultra Wideband Transceiver," Masters Thesis, Virginia Polytechnic Institute and State University, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, August 2006.

[63] P. Manish, "Simplified micro-controller and FPGA platform for DSP applications," in *Proceedings of the 2005 IEEE International Conference on Microelectronic Systems Education (MSE '05)*, 2005, pp.87–88.

[64] J. N. Edmison, "Electronic Textiles for Motion Analysis," Masters Thesis, Virginia Polytechnic Institute and State University, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, April 2004.

[65] M. Morales-Sandoval, "A hardware architecture for elliptic curve cryptography and lossless data compression," in *Proceedings of the 15th International Conference on Electronics, Communications and Computers (CONIELECOMP 2005)*, 2005, pp.113–118.

[66] 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.

[67] KULRD, SCARD, "Side Channel Analysis Resistant Design Flow", Deliverable *SCARD-KULARD-D4.1*, January 2005, www. scard-project.eu.

[68] P. Kochar. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems." in *Proceedings (CRYPTO96)*, Springer-Verlag, 1996, pp. 104-113.

[69] M.K. Birla, "FPGA Based Reconfigurable Platform for Complex Image Processing"**,** in *2006 IEEE International Conference on Electro Information Technology*, May 2006, pp. 204–209.

[70] David Samyde, Sergei Skorobogatov, Ross Anderson and Jean-Jacques Quisquater, "On a New Way to Read Data from Memory", Universit´e catholique de Louvain, UCL Crypto Group Place du Levant, Louvain-la-Neuve, Belgium University of Cambridge Computer Laboratory, Cambridge, United Kingdom, 2005

[71] 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.

[72] *Virtex-5 Multi-Platform FPGA*, Xilinx, www.xilinx.com

[73] O. Maslennikow, A. Sergiyenko, "Mapping DSP Alogorithms into FPGA**",** in *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, July 2006, pp. 208–213.

[74] W.R. Chang , D. H. Lee, H. J. Chi, K. S. Kwan, T. H. Kim, J. S. Park, "Design of Digital Audio DSP Core**",** in *The 1st International Forum on Strategic Technology*, October 2006, pp. 59–62.

[75] M. Bucci, L. Giancane,R. Luzzi, G. Scotti, A. Trifiletti, "Enhancing Power Analysis Attacks Against Cryptographic Devices**",** in *IEEE International Symposium on Circuits and Systems (ISCAS'06)*, May 2006, pp. 4–5.

[76] T. Guangming, L. Xu, S. Feng, N. Sun, "An Experimental Study of Optimizing Bioinformatics Applications**",** in *20th International Parallel and Distributed Processing Symposium*, April 2006, pp. 25–29.

[77] N.B. Armstrong, H.S. Lopes, C.R.E Lima, "Preliminary Steps Towards Protein Folding Prediction Using Reconfigurable Computing**",** in *IEEE International Conference on Reconfigurable Computing and FPGA's (RECONFIG'06)*, September 2006, pp. 1–7.

[78] Z. Ye, J. Grosspietsch, G. Memik, "An FPGA Based All-Digital Transmitter with Radio Frequency Output for Software Defined Radio**",** in *Design, Automation & Test in Europe Conference & Exhibition (DATE'07)*, April 2007, pp. 1–6.

[79] S. Dikmese, A. Kavak, S. Sahin, K. Kucuk, H. Dincer, "Evaluation of FPGA-based Software Radio Beamformers for 3G Wireless**",** in *2007 IEEE Radio and Wireless Symposium*, January 2007, pp. 153–156.

[80] T. Alho, P. Hamalainen, M. Hannikainen, T.D. Hamalainen, "Compact Hardware Design of Whirlpool Hashing Core**",** in *Design, Automation & Test in Europe Conference & Exhibition (DATE'07)*, April 2007, pp. 1–6.

[81] J. Daemen and V. Rijmen, "Resistance against implementation attacks: A comparative study of the AES proposals*",* In *Proceedings of the Second Advanced Encryption Standard (AES) Candidate Conference*, March 1999 [82]

[82] J.-S. Coron and L. Goubin, "On boolean and arithmetic masking against differential power analysis*",* In *Proceedings of 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 1965 of LNCS, pages 231-237, 2000, Springer-Verlag.

[83] J. J. A. Fournier and S. Moore and H. Li and R. Mullins and G.Taylor, "Security Evaluation of Asynchronous Circuits*", In Proceedings of 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 2779 of LNCS, pages 137-151, 2003, Springer-Verlag.

[84] S. Moore, R. Anderson, R. Mullins, G. Taylor and J. Fournier, "Balanced Self-Checking Asynchronous Logic for Smart Card Applications", *Microprocessors and Microsystems Journal*, 2003.

[85] A. Shamir, "Protecting smart cards from passive power analysis with detached power supplies", In *Proceedings of 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, number 1965 of LNCS, pages 71-77, 2000, Springer-Verlag.

[86] 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.

[87] 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.

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

[89] P.Alfke, R. Padovani, “Radiation Tolerance of High-Density FPGAs”, *Xilinx*, June 1998.

[91] D. Williams, “C++ Portability Guide.” Mozilla.org, August 10, 2007.

[92] E. Raymond, “The Art of Unix Programming”, *Thyrsus Enterprises*, September 2003.

Appendix A

Xilinx Virtex-II Pro Configuration Details

Virtex II XC2V250 Column	Mask Frame 1	Mask Frame 2	Space Between Columns
X0/X1	32	33	22
X2/X3	54	55	22
X4/X5	76	77	22
X6/X7	98	99	22
X8/X9	120	121	22
X10/X11	142	143	22
X12/X13	164	165	22
X14/X15	186	187	22
X16/X17	208	209	22
X18/X19	230	231	22
X20/X21	252	253	22
X22/X23	274	275	22
X24/X25	296	297	22
X26/X27	318	319	22
X28/X29	340	341	22
X30/X31	362	363	22

Figure A.1: Configuration Frames Containing Dynamic Data on the Virtex II XC2V250

116	118		756	758		1396	1398
156	158		796	798		1436	1438
196	198		836	838		1476	1478
236	238		876	878		1516	1518
276	278		916	918		1556	1558
316	318		956	958		1596	1598
356	358		996	998		1636	1638
396	398		1036	1038		1676	1678
436	438		1076	1078		1716	1718
476	478		1116	1118		1756	1758
516	518		1156	1158		1796	1798
556	558		1196	1198		1836	1838
596	598		1236	1238		1876	1878
636	638		1276	1278		1916	1918
676	678		1316	1318		1956	1958
716	718		1356	1358		1996	1998

* Note the offset between dynamic bits is held at a constant 2-40-2 interval

Figure A.2: Bit positions containing dynamic data in Virtex II XC2V250

Column	Row Description	Column	Row Description
X0	Y0:Y159.	X46	Y0:Y159.
X1	Y0:Y159.	X47	Y0:Y159.
X2	Y0:Y159.	X48	Y0:Y159.
X3	Y0:Y159.	X49	Y0:Y159.
X4	Y0:Y159.	X50	Y0:Y159.
X5	Y0:Y159.	X51	Y0:Y159.
X6	Y0:Y159.	X52	Y0:Y159.
X7	Y0:Y159.	X53	Y0:Y159.
X8	Y0:Y159.	X54	Y0:Y159.
X9	Y0:Y159.	X55	Y0:Y159.
X10	Y0:Y159.	X56	Y0:Y159.
X11	Y0:Y159.	X57	Y0:Y159.
X12	Y0:Y159.	X58	Y0:Y159.
X13	Y0:Y159.	X59	Y0:Y159.
X14	Y0:Y63, Y96:Y159.	X60	Y0:Y159.
X15	Y0:Y63, Y96:Y159.	X61	Y0:Y159.
X16	Y0:Y63, Y96:Y159.	X62	Y0:Y63, Y96:Y159.
X17	Y0:Y63, Y96:Y159.	X63	Y0:Y63, Y96:Y159.
X18	Y0:Y63, Y96:Y159.	X64	Y0:Y63, Y96:Y159.
X19	Y0:Y63, Y96:Y159.	X65	Y0:Y63, Y96:Y159.
X20	Y0:Y63, Y96:Y159.	X66	Y0:Y63, Y96:Y159.
X21	Y0:Y63, Y96:Y159.	X67	Y0:Y63, Y96:Y159.
X22	Y0:Y63, Y96:Y159.	X68	Y0:Y63, Y96:Y159.
X23	Y0:Y63, Y96:Y159.	X69	Y0:Y63, Y96:Y159.
X24	Y0:Y63, Y96:Y159.	X70	Y0:Y63, Y96:Y159.
X25	Y0:Y63, Y96:Y159.	X71	Y0:Y63, Y96:Y159.
X26	Y0:Y63, Y96:Y159.	X72	Y0:Y63, Y96:Y159.
X27	Y0:Y63, Y96:Y159.	X73	Y0:Y63, Y96:Y159.
X28	Y0:Y63, Y96:Y159.	X74	Y0:Y63, Y96:Y159.
X29	Y0:Y63, Y96:Y159.	X75	Y0:Y63, Y96:Y159.
X30	Y0:Y159.	X76	Y0:Y63, Y96:Y159.
X31	Y0:Y159.	X77	Y0:Y63, Y96:Y159.
X32	Y0:Y159.	X78	Y0:Y159.
X33	Y0:Y159.	X79	Y0:Y159.
X34	Y0:Y159.	X80	Y0:Y159.
X35	Y0:Y159.	X81	Y0:Y159.
X36	Y0:Y159.	X82	Y0:Y159.
X37	Y0:Y159.	X83	Y0:Y159.
X38	Y0:Y159.	X84	Y0:Y159.
X39	Y0:Y159.	X85	Y0:Y159.
X40	Y0:Y159.	X86	Y0:Y159.
X41	Y0:Y159.	X87	Y0:Y159.
X42	Y0:Y159.	X88	Y0:Y159.
X43	Y0:Y159.	X89	Y0:Y159.
X44	Y0:Y159.	X90	Y0:Y159.
X45	Y0:Y159.	X91	Y0:Y159.

Table A.1: Map File for the Xilinx Virtex II Pro XC2VP30 Device

Column X0/X1	Frame Address (Xilinx Format)	0x00060200	0x00060400	0x00080200	0x00080400	0x000a0200	0x000a0400
	Bit Offset (Inside Frame)	118	20	0	0	0	0
	Abosolute Bit Offset	108662	111956	0	0	0	0
	Relative Frame Offset	32	33	0	0	0	0
Column X2/X3	Frame Address (Xilinx Format)	0x00060200	0x00060400	0x00080200	0x00080400	0x000a0200	0x000a0400
	Bit Offset (Inside Frame)	0	20	118	118	0	0
	Abosolute Bit Offset	0	111956	183286	186678	0	0
	Relative Frame Offset	0	33	54	55	0	0
Column X4/X5	Frame Address (Xilinx Format)	0x00060200	0x00060400	0x00080200	0x00080400	0x000a0200	0x000a0400
	Bit Offset (Inside Frame)	0	20	0	0	118	118
	Abosolute Bit Offset	0	111956	0	0	257910	261302
	Relative Frame Offset	0	33	0	0	76	77

Table A.2: Example Table Generated by Compile Results Executable for the Xilinx Virtex II Pro XC2VP7

Appendix B

Xilinx Virtex-4 Configuration Details

STARTING FRAME ADDRESS Column	1 Frame Address	2 Frame Address	3 Frame Address	4 Frame Address	5 Frame Address	6 Frame Address
X0/X1	0x00000054	0x00004054	0x00008054	0x00400054	0x00404054	0x00408054
X2/X3	0x00000094	0x00004094	0x00008094	0x00400094	0x00404094	0x00408094
X4/X5	0x000000d4	0x000040d4	0x000080d4	0x004000d4	0x004040d4	0x004080d4
X6/X7	0x00000114	0x00004114	0x00008114	0x00400114	0x00404114	0x00408114
X8/X9	0x00000154	0x00004154	0x00008154	0x00400154	0x00404154	0x00408154
X10/X11	0x00000194	0x00004194	0x00008194	0x00400194	0x00404194	0x00408194
X12/X13	0x000001d4	0x000041d4	0x000081d4	0x004001d4	0x004041d4	0x004081d4
X14/X15	0x00000214	0x00004214	0x00008214	0x00400214	0x00404214	0x00408214
X16/X17	0x00000294	0x00004294	0x00008294	0x00400294	0x00404294	0x00408294
X18/X19	0x000002d4	0x000042d4	0x000082d4	0x004002d4	0x004042d4	0x004082d4
X20/X21	0x00000314	0x00004314	0x00008314	0x00400314	0x00404314	0x00408314
X22/X23	0x00000354	0x00004354	0x00008354	0x00400354	0x00404354	0x00408354
X24/X25	0x00000394	0x00004394	0x00008394	0x00400394	0x00404394	0x00408394
X26/X27	0x000003d4	0x000043d4	0x000083d4	0x004003d4	0x004043d4	0x004083d4
X28/X29	0x00000494	0x00004494	0x00008494	0x00400494	0x00404494	0x00408494
X30/X31	0x000004d4	0x000044d4	0x000084d4	0x004004d4	0x004044d4	0x004084d4
X32/X33	0x00000514	0x00004514	0x00008514	0x00400514	0x00404514	0x00408514
X34/X35	0x00000554	0x00004554	0x00008554	0x00400554	0x00404554	0x00408554
X36/X37	0x00000594	0x00004594	0x00008594	0x00400594	0x00404594	0x00408594
X38/X39	0x000005d4	0x000045d4	0x000085d4	0x004005d4	0x004045d4	0x004085d4
X40/X41	0x00000614	0x00004614	0x00008614	0x00400614	0x00404614	0x00408614
X42/X43	0x00000654	0x00004654	0x00008654	0x00400654	0x00404654	0x00408654
X44/X45	0x00000694	0x00004694	0x00008694	0x00400694	0x00404694	0x00408694
X46/X47	0x000006d4	0x000046d4	0x000086d4	0x004006d4	0x004046d4	0x004086d4
X48/X49	0x00000714	0x00004714	0x00008714	0x00400714	0x00404714	0x00408714
X50/X51	0x00000754	0x00004754	0x00008754	0x00400754	0x00404754	0x00408754
X52/X53	0x00000794	0x00004794	0x00008794	0x00400794	0x00404794	0x00408794
X54/X55	0x000007d4	0x000047d4	0x000087d4	0x004007d4	0x004047d4	0x004087d4

Figure B.1: Xilinx formatted frame addresses of flip-flop data on the Virtex 4 LX25

STARTING FRAME Column	1	2	3	4	5	6		Space Between Columns (in frames)	Space Between Frames (in frames)
X0/X1	50	782	1514	2246	2978	3710		22	732
X2/X3	72	804	1536	2268	3000	3732		22	732
X4/X5	94	826	1558	2290	3022	3754		22	732
X6/X7	116	848	1580	2312	3044	3776		22	732
X8/X9	138	870	1602	2334	3066	3798		22	732
X10/X11	160	892	1624	2356	3088	3820		22	732
X12/X13	182	914	1646	2378	3110	3842		22	732
X14/X15	204	936	1668	2400	3132	3864		43	732
X16/X17	247	979	1711	2443	3175	3907		22	732
X18/X19	269	1001	1733	2465	3197	3929		22	732
X20/X21	291	1023	1755	2487	3219	3951		22	732
X22/X23	313	1045	1777	2509	3241	3973		22	732
X24/X25	335	1067	1799	2531	3263	3995		22	732
X26/X27	357	1089	1821	2553	3285	4017		55	732
X28/X29	412	1144	1876	2608	3340	4072		22	732
X30/X31	434	1166	1898	2630	3362	4094		22	732
X32/X33	456	1188	1920	2652	3384	4116		22	732
X34/X35	478	1210	1942	2674	3406	4138		22	732
X36/X37	500	1232	1964	2696	3428	4160		22	732
X38/X39	522	1254	1986	2718	3450	4182		22	732
X40/X41	544	1276	2008	2740	3472	4204		22	732
X42/X43	566	1298	2030	2762	3494	4226		22	732
X44/X45	588	1320	2052	2784	3516	4248		22	732
X46/X47	610	1342	2074	2806	3538	4270		22	732
X48/X49	632	1364	2096	2828	3560	4292		22	732
X50/X51	654	1386	2118	2850	3582	4314		22	732
X52/X53	676	1408	2140	2872	3604	4336		22	732
X54/X55	698	1430	2162	2894	3626	4358			732
Jump to Next Column	-83	-83	-83	-83	-83				

Figure B.2: Absolute frame address' of flip flop data on the Virtex 4 LX25 device

.

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

* Note the offset between bits containing dynamic data jumps from 22 bits to 43 bits after bit 634

Figure B.3: Bit positions containing dynamic data in Virtex 4 LX25 device.

	Physical CLB Columns	0 – 13	14- 27	28-39	40-55
Physical CLB Rows	Readback Frame Address	(Column Address, Row Address)	(Column Address, Row Address)	(Column Address, Row Address)	(Column Address, Row Address)
96- 191	Top / Bottom Bit = 1	(0x 0, 0x 2)	(0x 1, 0x 2)	(0x 2, 0x 2)	(0x 3, 0x 2)
	Top / Bottom Bit = 1	(0x 0, 0x 1)	(0x 1, 0x 1)	(0x 2, 0x 1)	(0x 3, 0x 1)
	Top / Bottom Bit = 1	(0x 0, 0x 0)	(0x 1, 0x 0)	(0x 2, 0x 0)	(0x 3, 0x 0)
0-95	Top / Bottom Bit = 0	(0x0,0x2)	(0x1,0x2)	(0x2, 0x 2)	(0x 3, 0x 2)
	Top / Bottom Bit = 0	(0x 0, 0x 1)	(0x 1, 0x 1)	(0x 2, 0x 1)	(0x 3, 0x 1)
	Top / Bottom Bit = 0	(0x 0, 0x 0)	(0x 1, 0x 0)	(0x 2, 0x 0)	(0x 3, 0x 0)

Figure B.4: Virtex 4 LX25 FPGA frame address layout

Column	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54

Row	Hash Number																											
160- 191	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167
128-159	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139
96-127	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	18	109	110	111
64-95	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83
32- 63	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55
0 -31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

Figure B.5: Hash number to physical location mapping for the Virtex 4 LX25 device

Appendix C

Generate_Config Usage Guide

The generate_config executable is invoked as follows:

“gen_config [top_module_header.v] [ucf_file_header.ucf] [fpga_map.map] <start_x> <start_y> <bits / counter>”

The parameters to “generate_config” executable are described in Table C.1. When the generate_config is invoked, the executable first determines if the appropriate input files exist and can be opened. If successfully opened, the top module and ucf header files are read in. The map file is then parsed and a custom core is generated from the values of the row and column parameters specified on the command line. The custom core’s hdl is then inserted into the verilog top module provided, and the constraints corresponding to the columns to be filled are inserted into the constraints file header. In order for proper insertion to be achieved, the input top module header must contain the markers, “//1” and “//2” to specify the area of the file between which the core should be inserted. This custom “counter” module must then be instantiated in one of two ways. The top_module header can include the “counter” module as shown in the example top_module header in Table C.2, or the module can be included as a separate entity in the design that must be specified in the synthesis stage of the Xilinx build. In order for the user constraints file to be properly updated, a line containing the phrase “END_HEADER” must appear in the

file as shown in the UCF Header section of Table C.2. Examples of topmodule and UCF files that could be passed to the gen_config executable are shown in Table C.2. Once the gen_config executable has been invoked, the updated topmodule and UCF files are ready to be used by the Xilinx build process invoked by the generate_ll script.

Parameter	Function	Required (Yes / No)
top_module_header.v	This parameter specifies the name of the topmodule header you wish to have the custom core used to fill a CLB column inserted into.	Yes
ucf_file_header.ucf	This parameter specifies the name of the ucf header file that you wish to have the location constraints of the custom core used to fill a CLB column inserted into	Yes
fpga_map.map	This parameters specifies the input map files containing the layout of the target device	Yes
start_x	This parameter specifies the column of the device you wish to be constrained	No (Default: 0)
start_y	This parameter specifies the row of the device you wish to be constrained	No (Default: 0)
bits / counter	This parameter specifies the number of bits per counter you would like used in the custom core which is generated. This value corresponds to the number of slices contained in one CLB for the FPGA family you are targeting. (This value is 4 for the Virtex 4, Virtex II, and Virtex II Pro families.)	No (Default: 4)

Table C.1: Gen_Config Executable Parameter Description

Top Module Header	**UCF Header**
module ml300_top(sys_clk, system_reset, leds_1, pb);	#-------------------------------------#
input sys_clk;	# Clock Period Constraints
input system_reset;	#-------------------------------------#
wire clk;	
wire rst;	Net sys_clk LOC=B13;
assign clk = sys_clk;	Net sys_clk IOSTANDARD = PCI33_3;
assign rst = system_reset;	
input [2:0] pb;	Net system_reset LOC=P3;
	Net system_reset IOSTANDARD = PCI33_3;
output [3:0] leds_1;	Net system_reset TIG;
parameter size = 4;	#Net sys_clk LOC=AJ15;
	#Net sys_clk IOSTANDARD = LVCMOS25;
////////INSERT CUSTOM GEN_CONFIG CORE BELOW///////	#Net system_reset LOC=AH5;
//?1	#Net system_reset IOSTANDARD = LVTTL;
	## System level constraints
//?2	#Net sys_clk TNM_NET = sys_clk;
//	#TIMESPEC TS_sys_clk = PERIOD sys_clk 10000 ps;
	#Net system_reset TIG;
assign leds_1[0] = count;	
assign leds_1[1] = pb[0];	## FPGA pin constraints
assign leds_1[2] = pb[1];	#Net leds_1<0> LOC=AC4;
assign leds_1[3] = pb[2];	#Net leds_1<0> IOSTANDARD = LVTTL;
	#Net leds_1<0> SLEW = SLOW;
endmodule	#Net leds_1<0> DRIVE = 12;
	#Net leds_1<1> LOC=AC3;
module counter (clk, rst, count);	#Net leds_1<1> IOSTANDARD = LVTTL;
input clk;	#Net leds_1<1> SLEW = SLOW;
input rst;	#Net leds_1<1> DRIVE = 12;
parameter size = 2;	#Net leds_1<2> LOC=AA6;
output reg [size-1:0] count;	#Net leds_1<2> IOSTANDARD = LVTTL;
always@(posedge clk) begin	#Net leds_1<2> SLEW = SLOW;
if(!rst) begin	#Net leds_1<2> DRIVE = 12;
count <= 0;	#Net leds_1<3> LOC=AA5;
end	#Net leds_1<3> IOSTANDARD = LVTTL;
else begin	#Net leds_1<3> SLEW = SLOW;
count <= count + 1;	#Net leds_1<3> DRIVE = 12;
end	
end	
	#----------------END HEADER--------------#
endmodule	

Table C.2: Example Topmodule and UCF Header Files

Appendix D

Generate_ll Script Source

```
#generate_ll script
MODULE=ml300
MAPFILE=v2p7.map

for a in 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90
#for a in 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91

do
cd constraints/gen_config/
./gen_config.exe ${MODULE}_top.v ${MODULE}.ucf $MAPFILE $a 0 4
rm -f ../../hdl/${MODULE}_top_gen.v
mv ${MODULE}_top_gen.v ../../hdl/${MODULE}_top.v
rm -f ../${MODULE}_gen.ucf
mv ${MODULE}_gen.ucf ../
cd ../../
make clean
make bitfile
mkdir results/X$a
cp ${MODULE}.ll  results/X$a/
cp ${MODULE}_routed.ncd  results/X$a/
cp scripts/${MODULE}.srp results/X$a/
cp ${MODULE}_routed.par results/X$a/
cp netlist/${MODULE}.bld results/X$a/
done

for a in 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90
#for a in 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91
do
./compile_results.exe ../X$a/$MODULE.ll results.tab
done
```

Table D.1: Example “generate_ll” Script Configured for the Xilinx XC2VP7

Appendix E

Auxiliary Configuration Implementation Details

Dynamic Data Masking FSM Configurations

It can be seen from Figure B.2 that the frames containing dynamic data on the Virtex 4 platform increase in 22 frame intervals for the first 204 frames, and then a jump of 43 frames is encountered. The interval of 22 frames is then resumed until frame number 357 is encountered, and then a jump of 55 frames is needed. Once the pattern of 22 frame intervals is resumed, another random jump of 83 frames is needed after the 698^{th} frame. This general trend does continue on a column-by-column basis across all devices on the Virtex 4 platform, but the number of 22 frame increments contained in each interval is variable across different devices depending on the size of the particular device. This lack of uniformity across the dynamic data set on the Virtex 4 platform makes reconfiguring this FSM for different devices across the platform somewhat challenging, as both the number and length of these intervals must be adjusted. Because the length of configuration frames on the Virtex 4 platform is held constant,

the individual bits which must be masked inside each frame remain constant across all devices on the platform. These bits that must be masked for LX25 device on the Virtex 4 platform are outlined in Figure B.3

Unlike the Virtex 4 platform, the Virtex II and Virtex II Pro platform's dynamic data locality does display relative uniformity across both of the platforms. On each platform, the interval seen between frames containing dynamic flip-flop data comes in a 1-22-1 fashion, and can be seen in Figure A.1 (Figure of dynamic data location on VII platform). As was shown on the Virtex 4 platform, the number of intervals on a particular device is again dependent on the size of the device. Unlike the Virtex 4 platform, the length of a configuration frame on the Virtex II / II Pro platform is not constant, and is dependent on the size of the device. As a result, the number of individual bits which must be masked in each frame varies from device to device, however, the interval between these bits is constant in the form of a 2-40-2 bit pattern. This pattern can be seen in Figure A.2 (figure of dynamic data bit pattern in VII Pro device).

Because this divergence in dynamic data locality on the Virtex 4 and Virtex II / II Pro platforms is so significant, the multi-platform configuration integrity checker presented in this thesis provides a separate configuration of the dynamic data masking controller for each supported platform.

Alternate Resource Consumption Configurations

As mentioned in Section 4.3.3, the design of the cross-platform configuration integrity checker provides system designers with two configurations of the system that consume opposing sets of resources. These configurations were designed to provide flexibility in the set of resources needed to instantiate the system.

The first configuration provided minimizes the slice resource consumption of the system, with the system only consuming 4760 slices. This configuration was intended to protect a

complex design that requires a very high slice count, but has a relatively small data set. An example of such a system would be cryptography core, such as AES, DES/ Triple DES, or RSA. The second configuration of the system minimizes the number of BRAMs consumed by the design, only requiring four available BRAMs to instantiate the system. This configuration was intended to protect a data intensive design that must instantiate large data buffers. An example of such a system would be a video processing application. The slice and BRAM utilization percentages for each device under each configuration of the system on the Virtex II, Virtex II Pro and Virtex 4 platforms is shown in Figures 5.3, 5.4 and 5.5 in Section 5.4.1. These configurations were achieved by manipulating the implementation of the design's memory subsystem. In the slice minimization configuration, the memory subsystem was configured to utilize the FPGA's block RAM resources, requiring only a few slices to provide an interface to the memory subsystem. The BRAM minimization configuration instantiates the memory subsystem using the FPGA's CLBs configured as LUT RAMs.

Vita

Matthew Aaron Benz was born in Bridgeton, New Jersey, and grew up in nearby Rosenhayn, New Jersey, located roughly 35 miles south of Philadelphia, Pennsylvania. In May of 2001, Matthew graduated from Cumberland Regional High School, and later that year began his undergraduate career as an engineering student at Virginia Polytechnic Institute and State University (Virginia Tech). During his 4 years as an undergraduate at Virginia Tech, Matthew was an active member in several prominent collegiate clubs and organizations, in addition to founding the TenFour Web Design company and working as intern in the Wireless and Securities group at Luna Innovations in Blacksburg, Virginia. After struggling to make the transition from a lower tier high school to one of the most prominent undergraduate engineering programs in the United States, through hard work and sacrifice he managed to maintain 3.5 Grade Point Average (GPA) over his last 60 hours of study. This paved the way for his successful completion of a B.S. degree in Computer Engineering with a minor in Computer Science, as well as his acceptance into Virginia Tech's graduate Computer Engineering program. In addition to making the *dean's list* during each of his four semesters of graduate study, Matthew worked as both a Graduate Teaching Assistant (GTA) in the universities Computer Engineering Lab (CEL) and as a Graduate Research Assistant (GRA) in Virginia Tech's Configurable Computing Lab (CCM). In Summer of 2007, Matthew completed his M.S. degree in Computer Engineering and moved to San Diego, California, where he currently works as a Security Verification Engineer for Qualcomm Inc.