Spring 2018

SECURING FPGA SYSTEMS WITH MOVING TARGET DEFENSE MECHANISMS

Zhiming Zhang
University of New Hampshire, Durham

SECURING FPGA SYSTEMS WITH MOVING TARGET DEFENSE MECHANISMS

BY

ZHIMING ZHANG

Bachelor of Engineering, Qingdao University of Science and Technology, Qingdao, China, 2016

DISSERTATION

Submitted to the University of New Hampshire

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in

Electrical and Computer Engineering

May, 2018

This thesis will be examined by and approved in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering by:

Thesis Director, Qiaoyan Yu, Ph.D.
Associate Professor
Department of Electrical & Computer Engineering

Edward Song, Ph.D.
Assistant Professor
Department of Electrical & Computer Engineering

Mehmet Kayaalp, Ph.D.
Assistant Professor
Department of Electrical & Computer Engineering

On May, 2018

Original approval signatures are on file with the University of New Hampshire Graduate School.

ACKNOWLEDGEMENTS

First and foremost, I want to say thank you to my advisor Dr. Qiaoyan Yu. I received great amount of help form her in both my course work and the project we worked on together. Besides the academic aspect, she also gave me a lot of advice and encouragement when I made important decisions about my career. Most importantly, she helped me build up my interest and confidence to my research work.

I would also like to thank Dr. Song, Dr. Kayaalp and Prof. Whitney for their willingness to provide help on my thesis and other graduation work as members of my thesis committee.

I would also like to thank my fellow graduate students in the UNH Reliable VLSI Systems Lab: Jaya Dofe, Chenghua She, and Sean Kramer. I got a lot of help and advice from them on my research and I really had a great time working with them.

Finally, I wish to thank my family. Their support and care removed any concerns I could have living in a foreign country so I could fully focus on my research work.

Contents

List of Tables

List of Figures

ABSTRACT

SECURING FPGA SYSTEMS WITH MOVING TARGET DEFENSE MECHANISMS

by

Zhiming Zhang

University of New Hampshire

Field Programmable Gate Arrays (FPGAs) enter a rapid growth era due to their attractive flexibility and CMOS-compatible fabrication process. However, the increasing popularity and usage of FPGAs bring in some security concerns, such as intellectual property privacy, malicious stealthy design modification, and leak of confidential information. To address the security threats on FPGA systems, majority of existing efforts focus on counteracting the reverse engineering attacks on the downloaded FPGA configuration file or the retrieval of authentication code or crypto key stored on the FPGA memory. In this thesis, we extensively investigate new potential attacks originated from the untrusted computer-aided design (CAD) suite for FPGAs. We further propose a series of countermeasures to thwart those attacks. For the scenario of using FPGAs to replace obsolete aging components in legacy systems, we propose a Runtime Pin Grounding (RPG) scheme to ground the unused pins and check the pin status at every clock cycle, and exploit the principle of moving target defense (MTD) to develop a hardware MTD (HMTD) method against hardware Trojan attacks. Our method reduces the hardware Trojan bypass rate by up to 61% over existing solutions at the cost of 0.1% more FPGA utilization. For general FPGA applications, we extend HMTD to a FPGA-oriented MTD (FOMTD) method, which aims to thwart FPGA tool induced design tampering. Our FOMTD is composed of three defense lines on user constraints file, random design replica selection, and runtime

submodule assembling. Theoretical analyses and FPGA emulation results show that the proposed FOMTD is capable of tackling three levels' attacks from malicious FPGA design software suite.

Chapter 1

Introduction

1.1 Trend of FPGA Utilization

Field programmable gate arrays (FPGAs) as a group of programmable integrated circuitries are famous for their flexibility, low cost and efficiency. The programmable architecture and internal connection allow any functions of an ASIC to be realized in a FPGA chip. In fact, FPGAs have played very important roles in many fields. Both the usage and popularity of FPGAs kept increasing in the past several decades and it is reasonable to expect a even better trend of its development. In 2016, according to the data provided by [1], the FPGA market was already valued at USD 5.34 Billion and in 2023, this number is expected to be increasing to 9.50 Billion. The great amount of FPGA applications covering a variety of areas make a great contribution to the FPGA market size. The applications are broadly distributed in Aerospace, high performance computing, wireless communication, and security including, in detail, digital signal processing, ASIC prototyping, super computer, etc. FPGA can be widely involved in the designs of electronic systems because of its programmable and parallel nature, the low cost of updating compared to

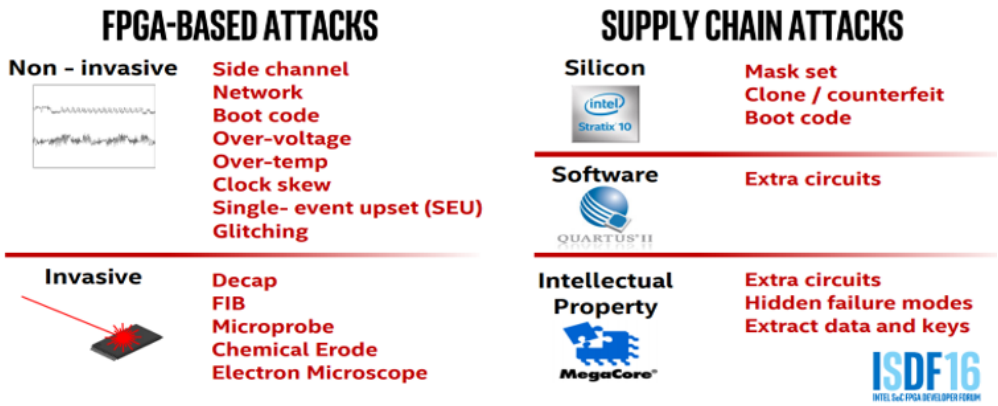


FIGURE 1.1: Security threats to FPGA system [3]

ASIC, and easy-to-design property for faster time-to-market [2]. However, nothing is perfect. Besides so many above mentioned advantages, FPGA systems can also face security problems.

1.2 Security Concerns on FPGA Applications

The security threats to FPGA systems can be from both the FPGA device and the supply chain including intellectual property (IP) theft, reverse engineering, logic tempering, and hardware Trojan [3] as can be seen in the figure 1.1. The rapidly increasing speed of FPGA market size can also bring some security issues because it also attracts attackers' attention for the high improper interest which may be obtained once the FPGA applications are manipulated.

Although many protections schemes have been developed to address the security concerns, there are still some blind spots being ignored. If we can detect those unrevealed potential security vulnerabilities, we are going to make a contribution for keeping the development of FPGA systems along a healthy path. In this thesis, we focus on hardware

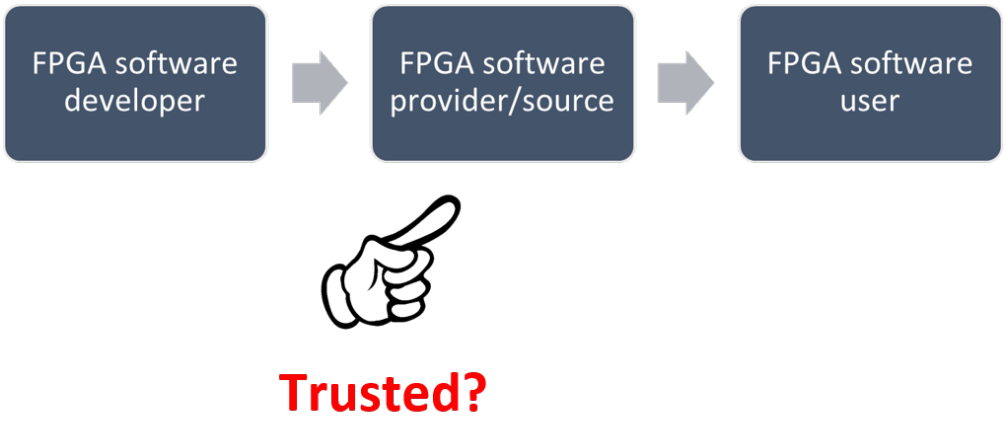


FIGURE 1.2: Potential security vulnerability in FPGA supply chain

Trojan attacks on FPGA systems and reveal that one potential security threat can be from the FPGA design software, which is usually called a CAD tool. Figure 1.2 shows a basic FPGA supply chain and we doubt the reliability of the CAD tool during the stage that after it is developed but before it is used. Furthermore, we propose a series of countermeasures to thwart this type of attacks.

1.3 Key Contributions

To address the security threat discussed in the previous section, we propose our protection mechanisms to thwart the attacks of hardware Trojan which is inserted through a untrusted CAD tool. More specifically, our main contributions are as follows.

1. We investigate new potential attacks originated from the untrusted FPGA design software and further propose a series of countermeasures to thwart those attacks.

2. To removed the security concerns when replacing obsolete aging components in legacy systems with FPGAs, we propose a RPG scheme to ground the unused pins

and check the pin status at every clock cycle, and exploit the principle of MTD to develop a HMTD method against hardware Trojan attacks.

3. For general FPGA applications, we extend HMTD to a FOMTD method, which aims to thwart FPGA tool-induced design tampering, FOMTD is composed of three defense lines in the user constraints file, random design replica selection, and runtime submodule assembling to defend the attacks from three different levels.

1.4 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2, an overview of FPGA architecture and the FPGA design suite will be introduced. Then we will talk about some existing research on the security problems of FPGA systems and a classic moving target defense countermeasure will be illustrated.

In Chapter 3, we address the security problems of using FPGAs to replace the aging components of legacy system.

1. To the best of our knowledge, this is the first work that investigates the countermeasure against the security threats that occur during the FPGA deployment for legacy systems. The primary goal of this work is to address the security attacks from the untrusted FPGA vendor and the CAD tools for FPGA configuration, rather than the IP piracy and side-channel attacks on FPGAs.

2. We propose a RPG scheme. Compared with the conceptual proposal in [4], we implemented the pin grounding concept on a Nexys-3 Spartan-6 FPGA board that successfully prevents the communication between the external environment and the

FPGA device. Moreover, our scheme additionally performs runtime checking to examine whether all user-unused I/O pins are truly grounded at every clock cycle, thus thwarting the countermeasure mutation by the FPGA CAD tool.

3. We propose a HMTD method. In our method, the hardware description of the aged functional module in the legacy system is replicated multiple times. Two of the replicas are randomly selected by an on-chip random number generator to examine the consistency between the two groups of outputs. Furthermore, instead of leaving the FPGA CAD tool to place and route the replacement module with default settings, we propose to explicitly specify the slice physical distance between the replicas in a FPGA user constraint file. Our method is able to thwart the stationary hardware Trojan insertion by the CAD tool.

In Chapter 4, we continue to work on the security vulnerabilities of the CAD tool and propose corresponding countermeasures to create unpredictabilities to the attacker.

1. We exploit the principles of moving target defense (MTD) and propose a FPGA-oriented MTD (**FOMTD**) countermeasure to resist the attacks from malicious FPGA tools. To the best of our knowledge, this work together with our prior work [5] are the first attempt to assess the feasibility of applying the MTD concept to defeat hardware Trojans from malicious FPGA software.

2. We propose **three defense lines** to generate three types of unpredictabilities to thwart the stealthy modification from compromised FPGA software.

3. We emulate a hardware Trojan hit rate for each defense line and analyze the emulation results.

In Chapter 5, the main contributions of the thesis are summarized and future work related to the topic is proposed.

Chapter 2

Background

2.1 FPGA Architecture

FPGA's reconfigurable architecture, which is illustrated in figure 2.1, allows it to be programmed after fabrication to perfectly mimic almost any logic functions that basic digital circuits can do. A standard FPGA is usually composed of three portions which are programmable logic blocks, programmable interconnections, and I/O blocks plus some additional advanced on-chip integrated circuitries, such as ALUs, block RAM, or DSP-48, for corresponding specific operations [6].

In general, programmable logic blocks are mainly made of look up tables (LUTs) which can complete combinational logics efficiently and flip-flops/latches. Figure 2.2 shows the LUTs and flip-flops in one slice of a Xilinx FPGA. The programmable routing usually includes pass transistors and buffers. The I/O blocks are used for off-chip connections.

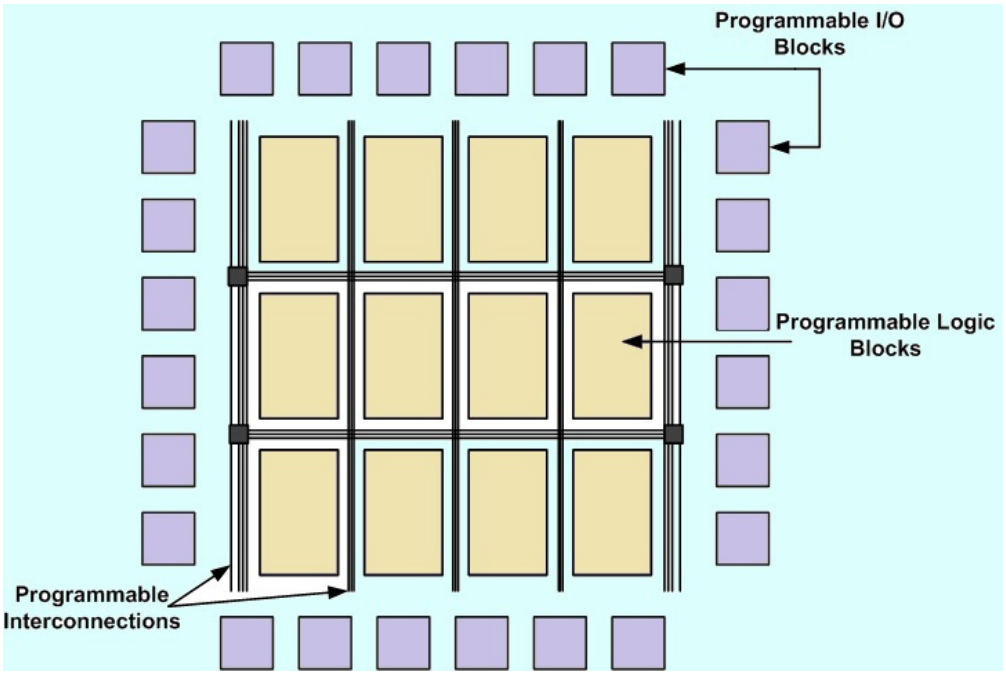


FIGURE 2.1: FPGA internal architecture [6]

2.2 FPGA Design Suite

FPGA behavior is driven by a bitsream file which is generated through FPGA design suite CAD software. Two main FPGA design suites in the market are Xilinx ISE and Altera Quartus. The design flows in these two kinds of CAD tools have some common features, as shown in figure 2.3. In Xilinx ISE [7], the design programmed in hardware design language is first synthesized. The output file of this stage will be combined with user constraints to be send to NGDBuild. After this step, the translated design is mapped then placed and routed according to the specified FPGA device. Finally, bitstream will generated. A similar procedure can also be performed in Altera Quartus in command line by quartus_map, quartus_fit, quartus_tan, and quartus_asm [8].

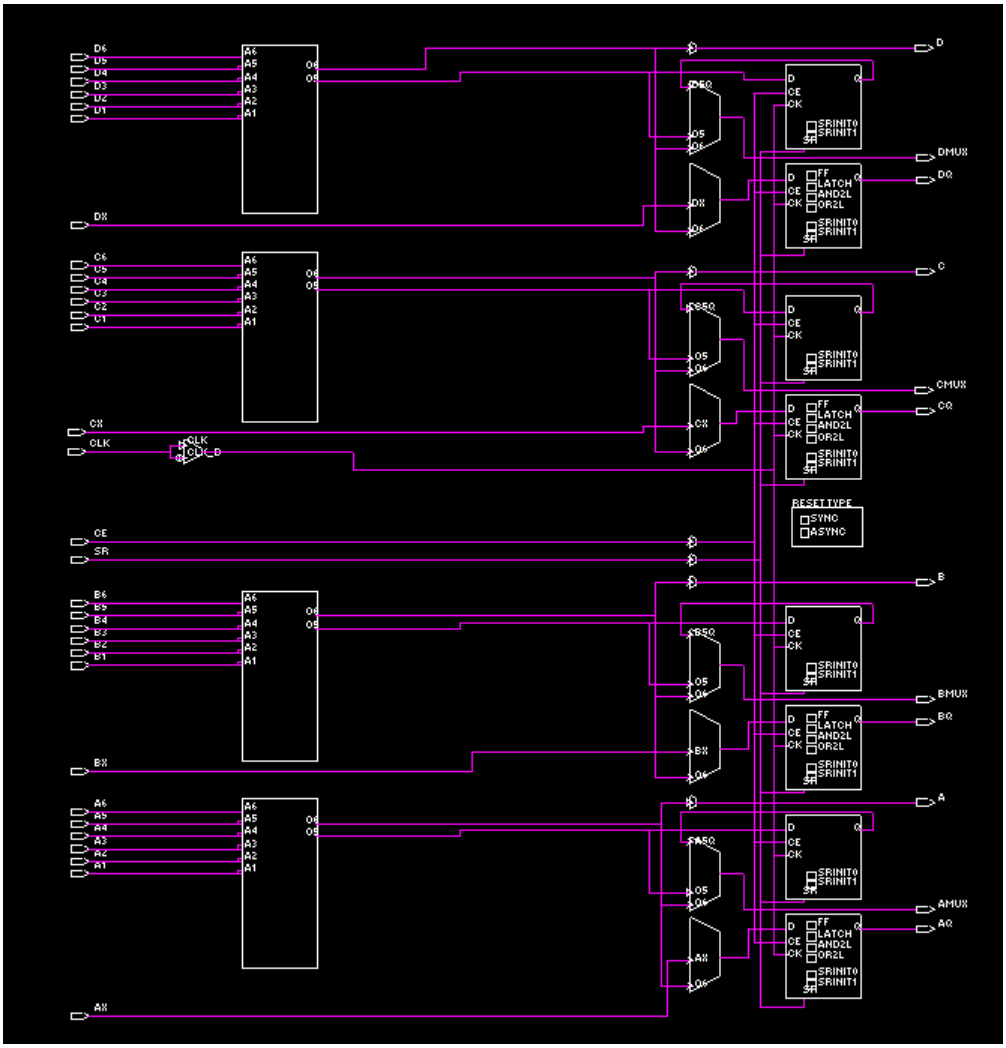


FIGURE 2.2: Architectures of Xilinx LUT and flip-flop

2.3 Existing Researches on FPGA Security

2.3.1 IP Piracy

Intellectual property (IP) of an electronic design always carries the key knowledge and technique of the research team who developed it. This also makes it become the target

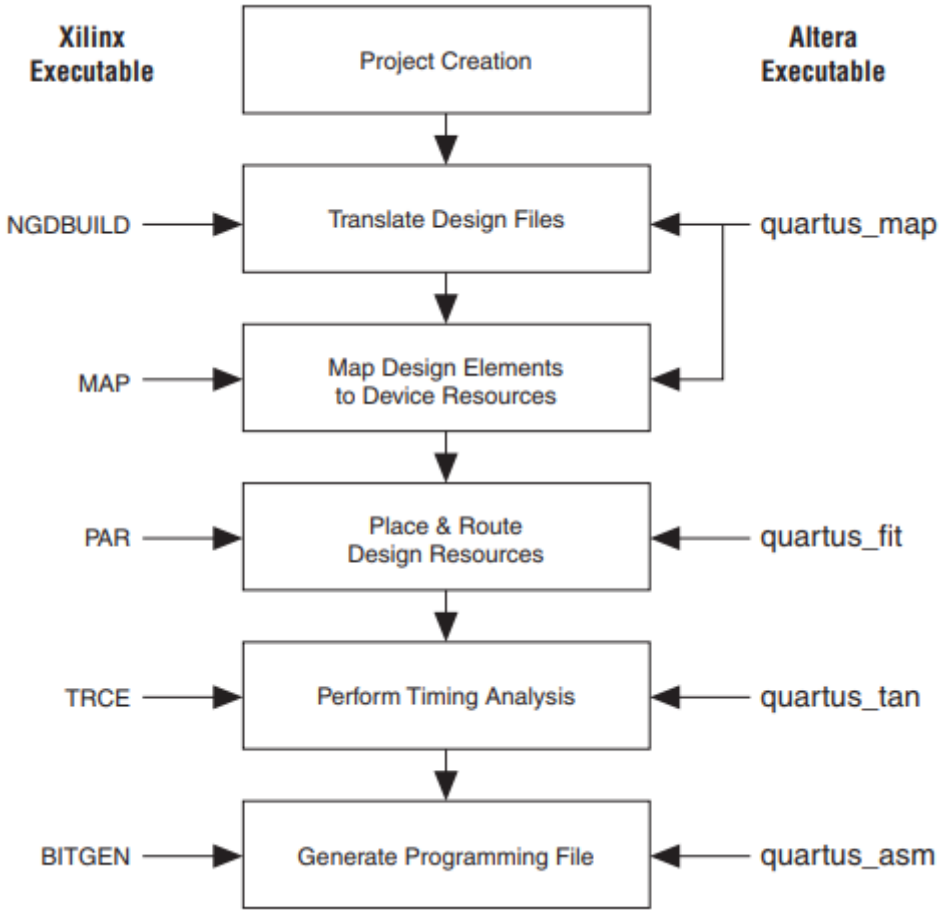


FIGURE 2.3: Typical FPGA configuration flow [8]

of attackers because by pirating the IP, attackers have a chance to significantly reduce the cost of developing an original one including money, time and research effort. In [9], it has already been proven that it is feasible to extract the FPGA design of IP using a Xilinx ISE FPGA design suite.

There are also some countermeasures being proposed to protect the IP integrity of FPGAs. A group of traditional countermeasures for IP piracy is watermarking. In [10], a method is presented which generates the watermarking for helping to identify the authorship of the design by manipulating the state transition graph to make it create a very

rare property. In this way, the watermarking can be very hard to be removed. A novelty PUF-FSM binding protection mechanism, in which a FSM is embedded in IP and it is activated by the response from the PUF embedded in FPGA, is proposed in [11] to restrict the IP to be only executed in an authorized FPGA device to avoid the IP being pirated. The approach demonstrated in [12] also provides a novel approach of device identification which proposes to assign each FPGA device a unique architecture and the architecture information will be used to encrypt the bitstream. By doing this, only the authorized device can work with the encrypted the bitstream so that the economic motivation of reverse engineering IP will be reduced.

2.3.2 Hardware Trojan

Hardware Trojan is another group of security threats which can be harmful for FPGA designs. The Trojans as extra and malicious circuitries can be inserted to FPGA systems through the vulnerable stages of FPGA design flow [13]. The purpose of a hardware Trojan can be disrupt the normal operation or leaking significant information and different with non-programmable devices, the hardware Trojans in FPGA can impact the design after configuration by exploiting the programmability. A taxonomy of FPGA-specified hardware Trojans is provided in [13]. It categorizes the Trojan into two main groups in terms of their triggers and payloads, as shown in figure 2.4.

The Morph Onion-encryption Replication partially runtime reconfiguration (PRR) hardware abstraction layer (HAL) (MORPH) architecture proposed in [14] is a very good defense mechanism, as shown in figure 2.5. It combines multiple levels of protections schemes including morph operation, onion encryption, replication, PRR as well as HAL and is able mitigate the Trojan from both fabrication-time and design-time. To solved

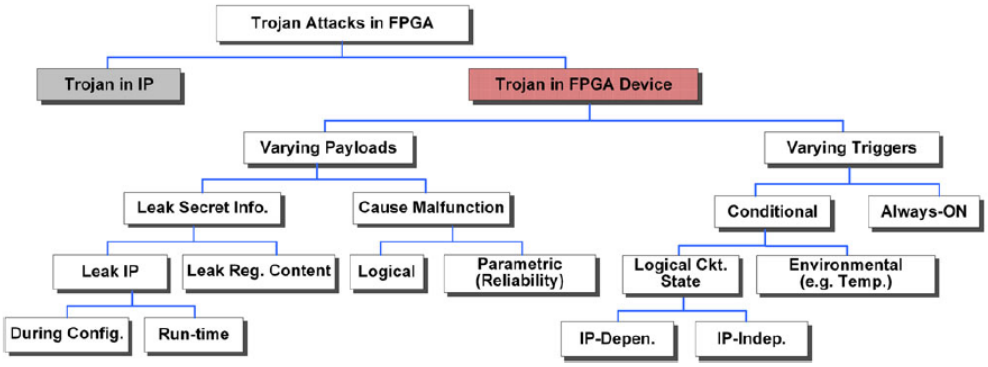


FIGURE 2.4: Taxonomy of FPGA-specific hardware Trojans [13]

the problem that hardware Trojan insertion protection may leave chip resources unused for attackers to manipulate, a solution is proposed in [15] which fills up the unoccupied space with low-level dummy logics. By doing this, there is no room in the bitstream of design for hardware Trojan insertion anymore. Adapted triple modular redundancy (ATMR) is another effective countermeasure to mitigate hardware Trojan attacks. In [13], a specific taxonomy of FPGA-based hardware Trojan attacks is first illustrated and in which the attacks are categorized according to the trigger and payload. Then a Adapted TMR aiming at detecting the hardware Trojan on chip is presented which replicates the design into three copies and the third copy is only activated when mismatch is found between first two. In [16], a Hardware Trojan Threats (HTT) detectability metric (HDM) is proposed to detect hardware Trojan in which the normalized physical parameters, such as power consumption and timing variation, will be weighted combined to be compared with threshold. If the combined result value is higher than the threshold, the FPGA will be determined to be malicious.

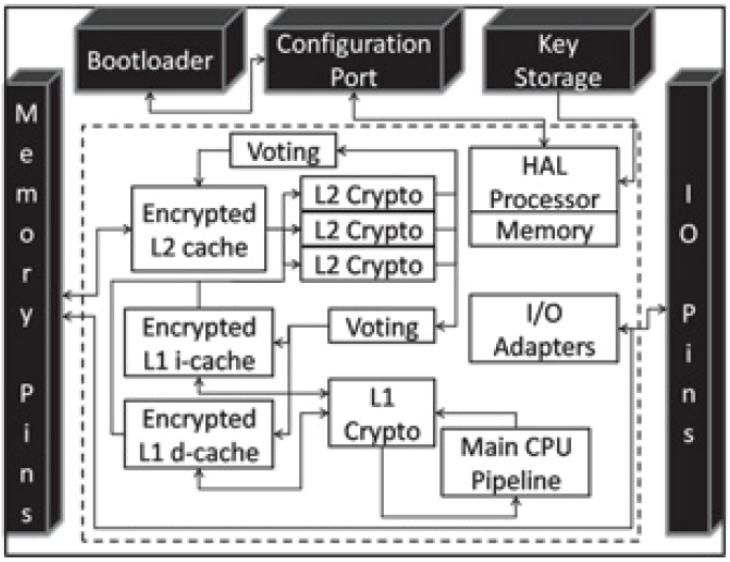


FIGURE 2.5: Taxonomy of FPGA-specific hardware Trojans [14]

2.3.3 Side Channel Analysis

Side channel analysis is one type of serious security problems, which is mainly composed of power analysis, timing analysis and electromagnetic emanation analysis [17], for digital circuits because of its ability to extract data and it also commonly threatens FPGA systems. There are also existing solutions to deal with this threat. A fake key based countermeasure is introduced in [18], in which a runtime changed fake key is randomly selected to perform the AES algorithm aiming at confusing the attacker. Extracting data by running SCA, no matter power consumption analysis or electromagnetic radiation analysis, on the fake AES algorithm cannot provide attackers any important information about the protection method. In the method proposed in [19], an interfering power signal is generated by a uncorrelated power noise generator according to the manipulated data

and an interfering key. The aim of doing this is to let the interfering power signal interfere with the attacker's analysis by breaking the correlation between the power measured and the encryption key. An asynchronous FPGA architecture, SCAR-FPGA, is designed in [20] to thwart power-based side channel attacks. It is based on pre-charged logic and a proposed LUT structure that can help balance the power consumption when reading the values "1" and "0" so that it generates data-independent power signal.

2.4 Moving Target Defense

2.4.1 Principle of MTD

The key idea of the whole protection system proposed in this thesis is based on the classic moving target defense method. It can be well illustrated with a "shell game" [21]. The game host randomly switches the positions of the shells to mess up the inference of audience about which one the ball is covered with. Moving target defense has been broadly used in the designs of protection for electronic systems. The key idea of it is developed into dynamically shifting the attacker surfaces, such as network IP address and port conditions, to increase the unpredictability, complexity and cost to attackers when implementing attacks.

2.4.2 MTD Applications in Electronic Systems

The method of moving target defense has been implemented in electronic systems in varies of domains. In [22], moving target defense is developed to mitigate the privacy-related attacks in IPv6 by reducing the ability of attackers to determine the two communicating hosts. In [23], the idea of moving target defense is adopted to thwart distributed

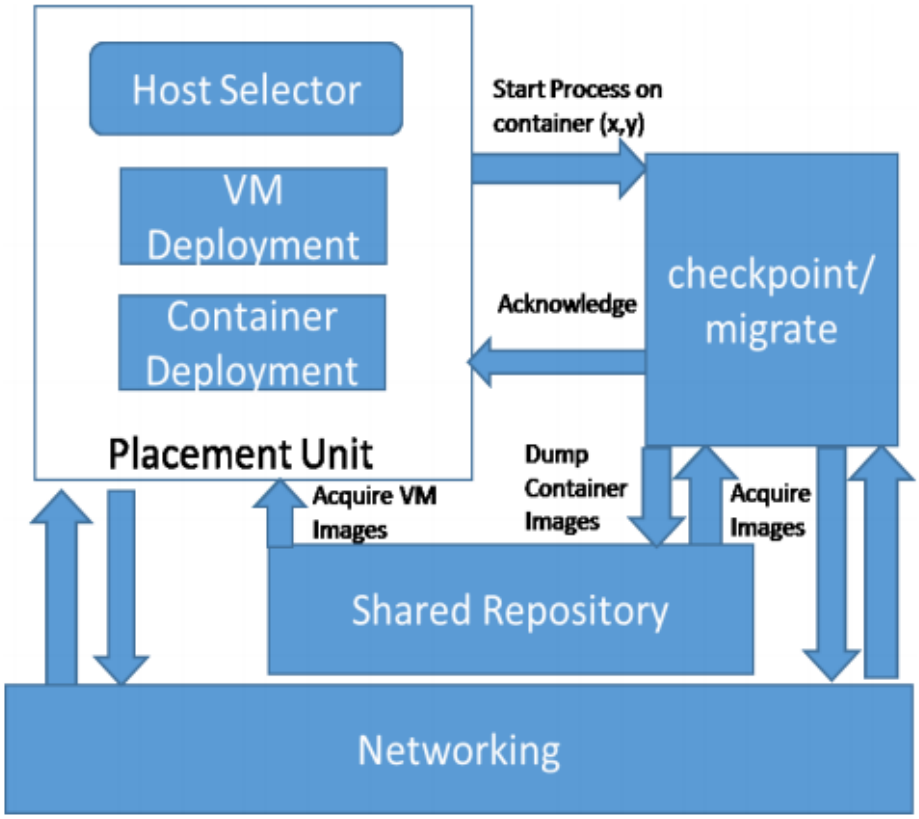


FIGURE 2.6: MIGRATE Architecture [24]

denial of service attacks. A MIGRATE, as shown in figure 2.6, is proposed in [24] which is a real-time moving target defense for creating obfuscations to defend the co-residency side channel attacks on computing clouds. The proposed countermeasures in the thesis are also inspired by moving target defense and develop the idea of it to provide better protection for the FPGA systems in hardware level.

Chapter 3

Securing FPGA-Based Obsolete Component Replacement for Legacy Systems

3.1 Motivation

The lifetime of electronics systems is always expected to be long in civil use, industry, military, etc. In a legacy system, component-aging is unavoidable and some electronic components may experience aging earlier than others. Unfortunately, the aged components may no longer be manufactured or available on the market. A straightforward solution is to re-design the entire system, but the total cost for re-designing, testing, and installation could be 10 times that of other alternatives, such as component replacement [25]. An obsolete component can be substituted by an equivalent device from gray market, application-specific integrated circuit (ASIC), field-programmable gate array (FPGA) [26], or uncommitted logic array (ULA).

Traditionally, functionality matching is the primary focus when we replace the aged

module with a functional equivalent. Little or no attention is paid to the security threats originated from the component replacement. Unfortunately, the trustworthiness of the FPGA supply chain has become a serious concern now, so it is imperative to address those security threats, in particular, from untrusted FPGA manufacturers and computer-aided design (CAD) tools associated with FPGA deployment.

3.2 Existing Solutions

Many protection schemes have been proposed to protect against FPGA security threats. To detect the hardware Trojans carried in the FPGA configuration bitstream, Chakraborty et al. [4] suggest the following: grounding the unused I/O pins; monitoring the temperature of the FPGA device; filling up the unused resources of the FPGA; or scrambling the bitstream file. Bloom et al. [27] propose to morph on-chip resources for moving target defense (MTD) against fabrication-time Trojans. Their method heavily utilizes encryption on the FPGA configuration for initialization boot and hardware description of functional modules. Moreover, process memory, L1 cache, and L2 cache are encrypted separately using multi-layer encryption. Although the alteration of two instances for the same CPU implementation can thwart random hardware Trojans, the multi-layer encryption is too costly for many real-time systems. The ideas proposed in [4] and [27] remain at the conceptual level, and no practical experiments have been conducted to demonstrate the method's feasibility.

To protect the FPGA configuration bitstream against piracy, reverse engineering, and tampering, Karam et al. [28] obfuscate the FPGA bitstream by inserting additional functions in the look-up tables (LUTs) that are configured for the true functionality of the

design. Jyothi et al. [29] utilize ring-oscillator arrays to measure process variation among FPGA slices, which may be modified by the untrusted FPGA manufacturer. Then, the FPGA region where the process variation is below the acceptable threshold is identified as a trust zone. The authors place the hardware design only in the trusted FPGA zones. This method assumes that the malicious FPGA slices lead to significant changes in delay, and the FPGA CAD tool is trusted. Mal-Sarkar et al. [13] propose an adapted triple modular redundancy (ATMR) technique to detect the hardware Trojan inserted in one of the design replicas. To reduce the overhead on power consumption, the third replica is activated once the output mismatch is detected from the other two replicas. The limitation of this method is that the three copies of the design module are allocated by the FPGA CAD tool in a stationary manner. Because the untrusted CAD tool has the prior knowledge of the place and route rules, theoretically, the tool can insert the same Trojan in the two replicas of the design. Thus, the ATMR method may not detect the Trojan.

The aforementioned methods assume that the FPGA CAD tool is trusted. These methods do not consider the scenarios in which hardware Trojans in the bitstream configuration can be inserted during the place and route stage. Another challenge is how to prevent the countermeasure from being removed/muted by the untrusted FPGA CAD tool.

3.3 Proposed Method

To address the security concerns discussed in the previous section, we propose a framework to detect the hardware Trojan inserted by an untrusted FPGA manufacturer or CAD tools. The proposed countermeasure, which is composed of two parts: (1) RPG and (2) HMTD, protects against the hardware Trojan attacks by appending commands in user

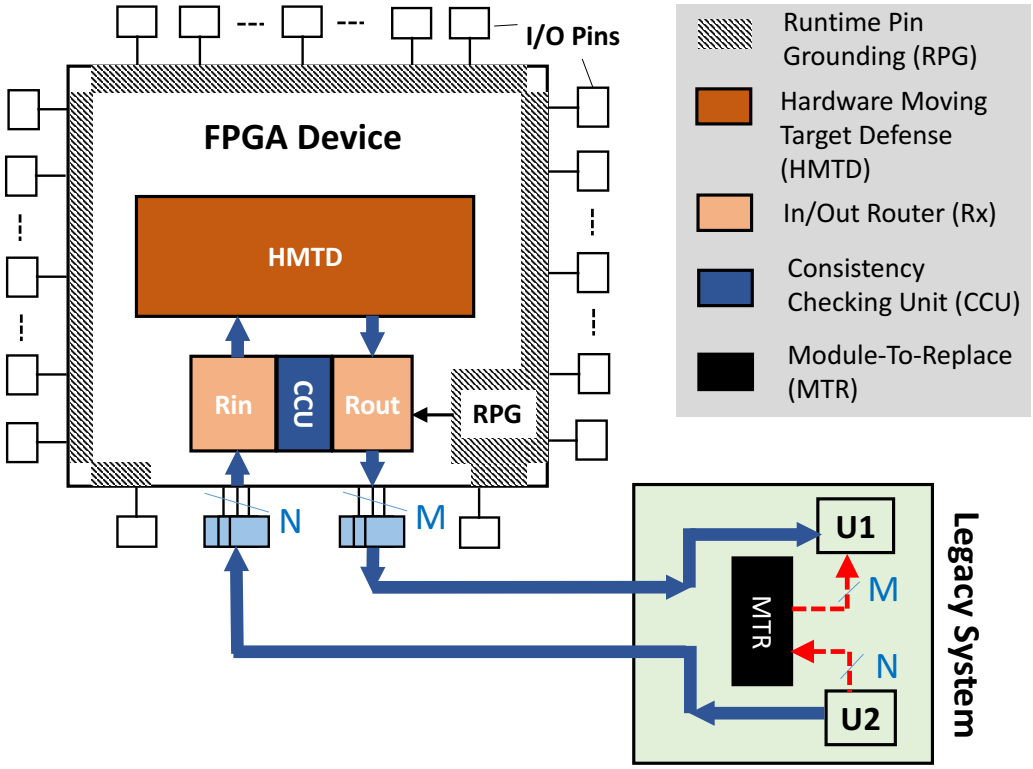


FIGURE 3.1: Overview of proposed countermeasure to secure the FPGA replacement for a legacy system. To replace the aged module (MTR), the proposed method connects a group of FPGA modules (HMTD+Rin+Rout+CCU) to the original modules U1 and U2 in the legacy system.

constraints file and modifying the original Verilog design files. The RPG scheme is to terminate the hardware Trojans that communicate with the external environment through unused I/O pins on the FPGA device. The HMTD method prevents the Trojan horses induced by the malicious FPGA CAD tools from interfering with the FPGA replacement in legacy systems. Figure 3.1 depicts the overview of the proposed countermeasure against hardware Trojans on the FPGA device.

3.3.1 Proposed Runtime Pin Grounding

Inspired by the idea proposed in [4], we apply the pin grounding scheme to the unused FPGA I/O pins by using a user constraint file. In this work, we continue to use the Nexys-3 FPGA board to introduce the procedure of our RPG scheme. This scheme is implemented in the top level of the hardware description module as shown in the black shadowed area of Fig. 3.1.

First, we assign every unused pin a net name in the top level of the hardware design file. Then, each NET name is linked with an unused I/O pin in the user constraint file by using the command (1).

NET “net_name" LOC = pin_name (1)

After that, we proceed to ground those I/O pins through the command (2).

NET “net_name" PULLDOWN (2)

Even if we have grounded all of the unused pins through the user constraint file, the malicious FPGA CAD tool can alter the user-specified pin configuration by modifying the native circuit description (.ncd) file. This phenomenon has been observed in our FPGA deployment environment Xilinx ISE 14.1 [30] when we manually ground the pin reserved for the power supply. Because the .ncd file is not readable, the hardware Trojans placed by the CAD tool are stealthy. To thwart the unrevealed modification from the CAD tool, we enhance our pin grounding scheme by adding a runtime detection circuit. Since we have assigned a net name for each unused I/O pin, we can simply use the logic of *NOR* to examine the grounding status of those unused pins.

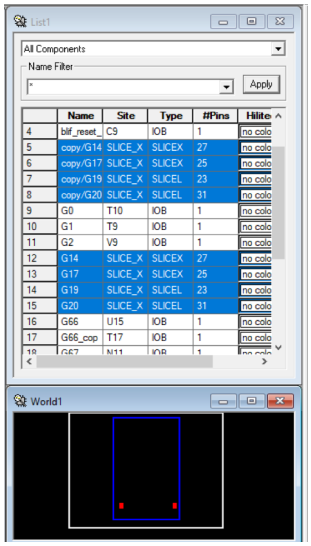


FIGURE 3.2: Detailed slice assignment shown in the FPGA Editor. The two red dots represents the locations for the two replicas of MTR equivalence that are specified in our method through FPGA Editor.

3.3.2 Proposed Hardware Moving Target Defense

To prevent the malicious CAD tool from successfully sabotaging the original FPGA configuration, we use the principle of MTD to develop the HMTD method. We assume that the functionality of the module-to-replace (MTR) in the legacy system is known by the FPGA deployment team, who is trusted. Our HMTD method replicates the MTR into multiple copies CP_0, CP_1, $\cdots$, CP_j. We use the "RLOC" command to specify the relative physical distance between two replicas in the user constraint file. For instance, we can assign CP_0 and CP_1 to the two corners of the FPGA device by setting **RLOC = X36Y61** and **RLOC = X1Y60**, respectively. Alternatively, we can utilize the FPGA Editor tool to perform the similar operation. Figure 3.2 shows that two replicas of MTR equivalent are successfully placed to two FPGA corners by our method.

In the next step, we add a low-cost, random number generator in the *Rin* unit to select two replicas of the function module to feed the N-bit inputs from *U2* in the legacy

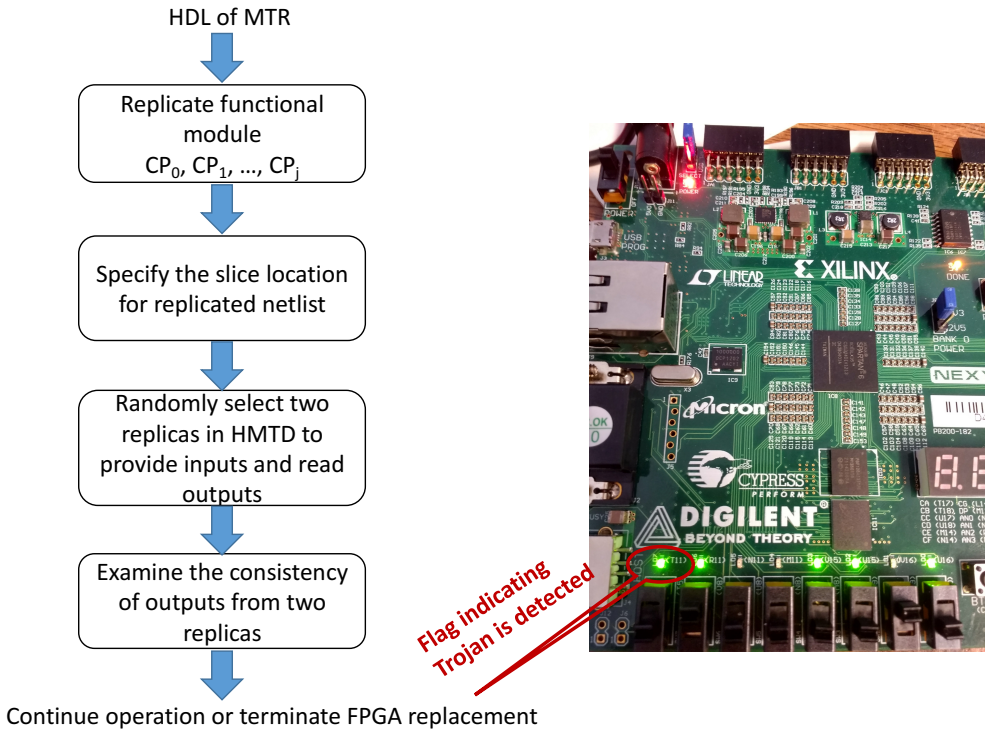


FIGURE 3.3: Flowchart of proposed hardware moving defense method.

system. This setting is essentially a power-gating technique to reduce the power consumption. For sequential circuits, state restore will be required in order to use the input gating technique. Note that the random number generator is implemented on the FPGA, and thus the random selection is performed at runtime. The random number generator also controls the *Rout* unit to choose which two replicas for the Trojan detection in the consistency checking unit (CCU). Once the output inconsistency is found, the M-bit output pins are grounded immediately and the flag for the Trojan detection is turned on. The flowchart of our HMTD method is summarized in Fig. 3.3.

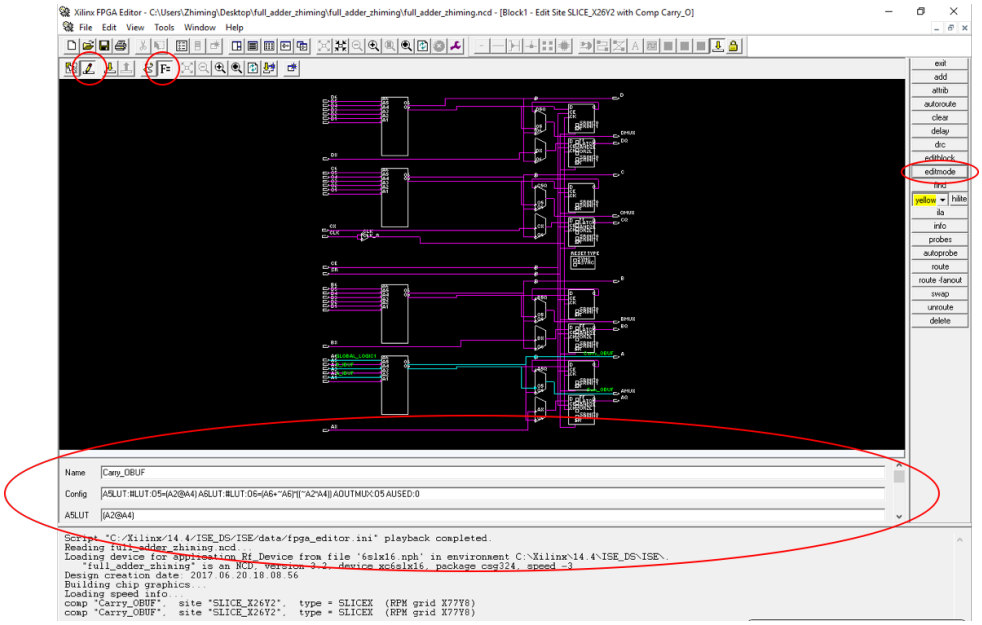
3.4 Experiment Results

3.4.1 Experimental Setup

The following experiments were performed on the Nexys-3 Board, which contains a Xilinx Spartan-6 XC6SLX16 CSG324C FPGA. This FPGA device includes 324 I/O pins (232 of which are user I/O pins) and 2,278 slices, each containing four 6-input LUTs and eight flip-flops. We used the Xilinx ISE 14.1 version to synthesize, place and route the Verilog HDL design files and generate bitstreams. The hardware overhead assessment was based on the ISCAS'85 benchmark circuits. We inserted the hardware Trojans on the FPGA device through two techniques: one is through the FPGA Editor, and the other is via editing the native circuit description file. Both of these techniques do not require changing the Verilog HDL file of the function module. The slice assignment shown in the FPGA Editor (see Fig. 3.4(a)) demonstrates that the FPGA CAD tool can successfully alter the configuration of one unused FPGA slice without disturbing the logic netlist. Although a native circuit description (.ncd) file is not readable, we can use an xdl program to translate that .ncd file to a readable file. Figure 3.4(b) also demonstrates that the hardware Trojan has been successfully placed in an un-occupied slice. We compared the Trojan resistance strength of our method and the ATMR approach [13] in the following subsection.

3.4.2 Hardware Trojan Bypass Rate

We validated the proposed HTMD method on the Nexys-3 board. Whenever a Trojan is detected, the flag light on the board will be turned on as shown in the right side of



(A) The FPGA Editor

```
75  inst "Carry_OBUF" "SLICEX",placed CLEXM_X16Y2 SLICE_X26Y2  ,
76    cfg " A5FFSRINIT::#OFF A5LUT:Mxor_Sum_xo<0>1:#LUT:O5=(A2@A4)
      A6LUT:Carry1:#LUT:O6=(A6+~A6)*((~A2*A4))
77        AFF::#OFF AFFMUX::#OFF AFFSRINIT::#OFF AOUTMUX::O5 AUSED::0
          B5FFSRINIT::#OFF
78        B5LUT::#OFF B6LUT::#OFF BFF::#OFF BFFMUX::#OFF
          BFFSRINIT::#OFF BOUTMUX::#OFF
79        BUSED::#OFF C5FFSRINIT::#OFF C5LUT::#OFF C6LUT::#OFF
          CEUSED::#OFF
80        CFF::#OFF CFFMUX::#OFF CFFSRINIT::#OFF CLKINV::#OFF
          COUTMUX::#OFF
81        CUSED::#OFF D5FFSRINIT::#OFF D5LUT::#OFF D6LUT::#OFF
          DFF::#OFF DFFMUX::#OFF
82        DFFSRINIT::#OFF DOUTMUX::#OFF DUSED::#OFF SRUSED::#OFF
          SYNC_ATTR::#OFF
83        "
84  ;
```

(B) The .xdl file converted from a .ncd file

FIGURE 3.4: FPGA hardware Trojans inserted without disturbing the hardware description file. The modified slice can be observed.

Fig. 3.3. To extensively assess the success rate of different FPGA hardware Trojan detection methods, we modeled the Trojan insertion and the detection methods in MATLAB.

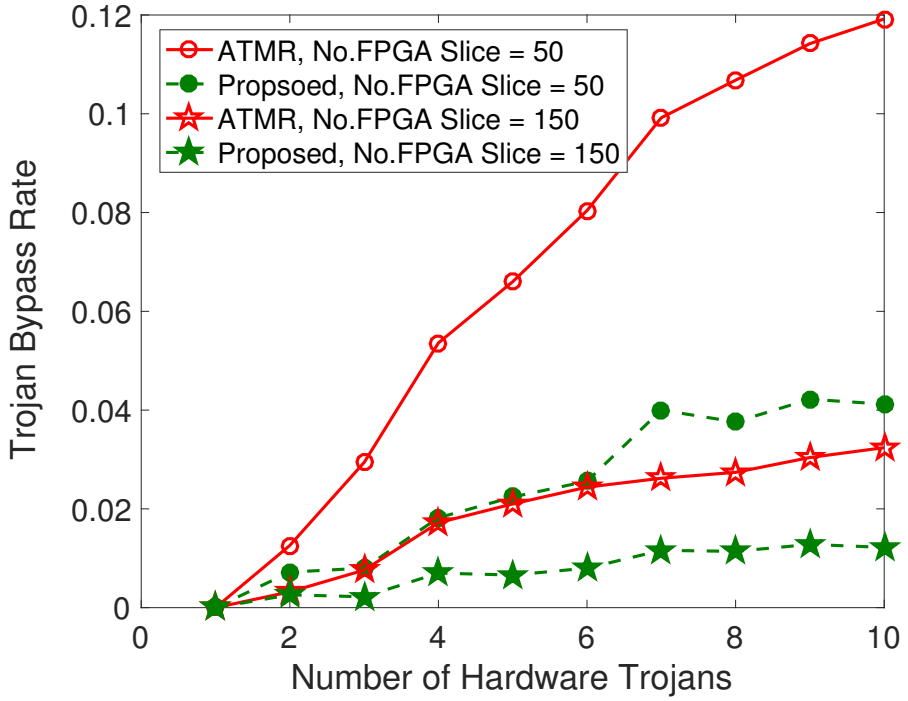


FIGURE 3.5: Hardware Trojan bypass rate versus number of hardware Trojans inserted in the FPGA device.

We randomly selected 10 slices for hardware Trojan insertion. This operation was conducted after the slices for the original design module were configured. The hardware Trojan bypass rate is defined as the number of incorrect outputs, due to Trojans, over the number of test cases.

The impact of the number of the hardware Trojans on the Trojan bypass rate is shown in Fig. 3.5. As can be seen, for the range of 1 to 10 Trojans, the Trojan bypass rate almost monotonically increases with the number of injected hardware Trojans. As the number of Trojans increases, the probability for multiple replicas of the functional module simultaneously containing Trojans increases. Hence, comparison of the two copies' outputs gradually loses the Trojan detection capability, and thus the Trojan bypass rate increases.

We vary the number of FPGA slices to examine the impact of the FPGA size on the

hardware Trojan bypass rate. In Fig. 3.6, we can observe that the Trojan bypass rate for a larger FPGA is lower than that for a smaller one. This is because the number of Trojans placed in the FPGA device is fixed per each FPGA size. The chance for a Trojan slice colliding with a design slice is higher in a smaller FPGA than in a larger one. The ATMR method compares the two of three copies for the design module using a fixed algorithm, which can only resist truly random Trojans. In contrast, our method randomly selects any two replicas for Trojan detection at runtime; moreover, our method is capable of assigning each replica to a specific location. Thus, the design location specified by our method is not predictable to the CAD tool. Hence, the randomness and unpredictability of our method strengthens the FPGA replacement resistance against the security threats from the untrusted FPGA manufacturer and CAD tool vendor.

To have a comprehensive view, we plot the Trojan bypass rate versus the FPGA size and the number of Trojans in Fig. 3.7. As shown, the 3D mesh sheet of our method is lower than that of the ATMR method [13]. On average, our method reduced the Trojan bypass rate by 61%.

3.4.3 Overhead on Hardware Cost and Performance

We applied the RPG scheme to the ISCAS'85 benchmark circuits. Because more unused I/O pins lead to more overhead for pin grounding, we chose the benchmark circuits with a small number of inputs/outputs. As shown in Table 3.1, the number of utilized LUTs go as high as 40, whereas the number of occupied slices go up to 16. These hardware implementations consume 0.044% more LUTs and 0.07% more slices, respectively.

After the FPGA place and route step, we measured the worst-case delay of the c432, c1355, and c6288 benchmark circuits with and without the proposed HMTD method. As

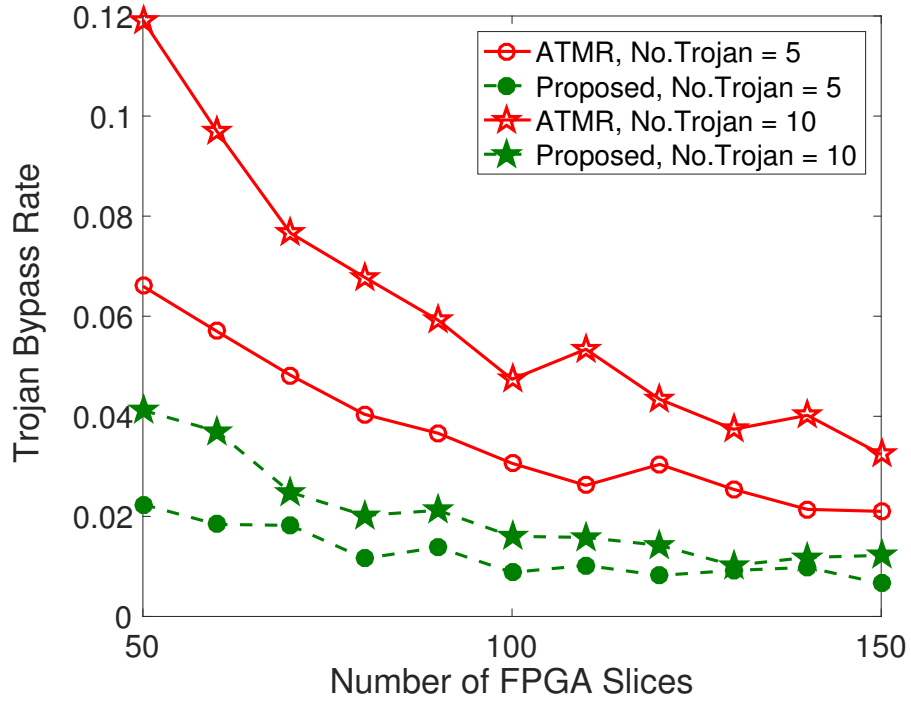


FIGURE 3.6: Impact of the number of FPGA slices on hardware Trojan bypass rate.

TABLE 3.1: FPGA Overhead of Proposed Runtime Pin Grounding

Overhead\Circuits	s298	s344	s444	s526	s1488
Increased No. LUTs	40	30	38	40	39
Increased No. Slices	12	6	11	12	16

we mentioned in Section IV.B, we manually added a physical distance between the replicas of the functional module to thwart the Trojan attack from the CAD tool. The induced separation may result in longer routing interconnects than the baseline. Depending how the replicas are assigned to the FPGA slices and the amount of distance is added between two copies, the delay overhead of our method varies. We recorded the minimum and maximum delay overhead as observed in our case study. As shown in Fig. 3.8, the average minimum (maximum) delay overhead of HMTD is 37% (70%).

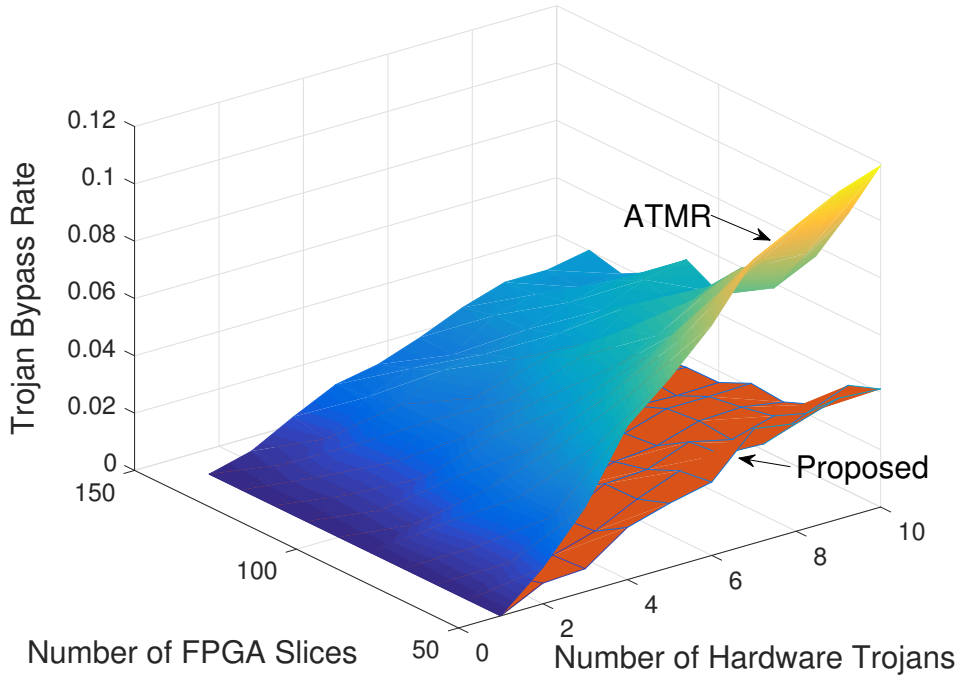


FIGURE 3.7: Three-dimensional plot for the dependent factors for hardware Trojan bypass rate.

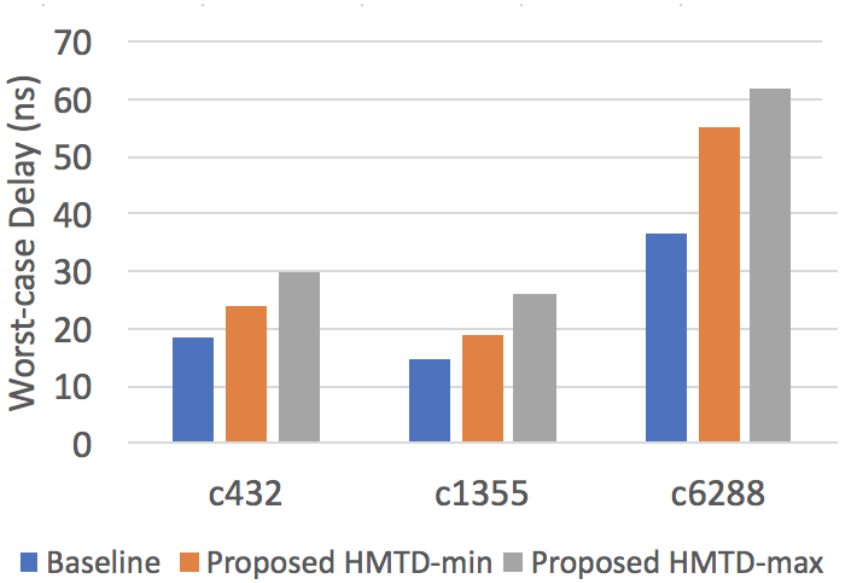


FIGURE 3.8: The delay overhead of proposed HMTD applied on benchmark circuits.

Chapter 4

FPGA-Oriented Moving Target Defense against Security Threats from Malicious FPGA Tools

4.1 Motivation

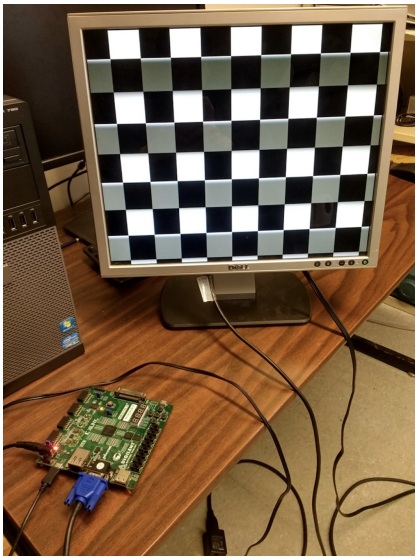
Field Programmable Gate Arrays (FPGAs) enter a rapid growth era due to their attractive flexibility and CMOS-compatible fabrication process. Because of the high demand on the FPGA usage in data processing, industrial, automotive, consumer electronics, telecom, military & aerospace, FPGA market achieves a compound annual growth rate of 8.4% [31]. Global Market Insights predicts that the FPGA market size is expected to reach 9.98 billion US dollars by 2022 [31]. The increasing popularity of FPGA may drive more attackers to compromise FPGA-based systems through various channels.

The work [32] highlights that FPGA security embraces four aspects: (1) the secure operations conducted by FPGA devices, (2) the utilization of FPGAs for the system security enhancement, (3) the secure bitstream delivery to FPGA devices, and (4) the exploitation

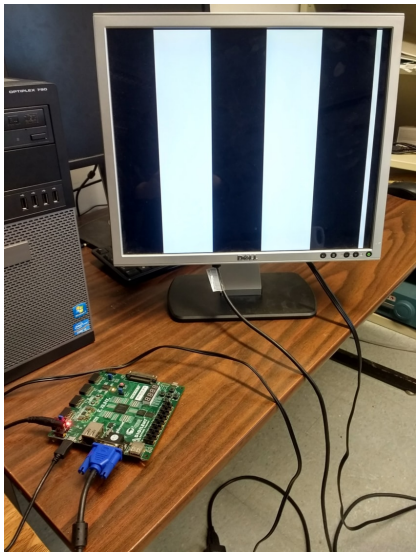
of FPGA devices as an attack surface of FPGA-based systems. The aspects (1) and (2) are toward the benefits we could obtain by utilizing FPGAs. The programmable features of FPGAs have been exploited to address the security challenges that ASIC chips are facing. For example, the embedded FPGA is used to perform locking key authentication [17, 33]. Whereas, FPGAs have their own security vulnerabilities. The surveys [34, 35] and literatures [17, 36, 37] extensively discuss the aspects (3) and (4). If a FPGA device is not carefully deployed its security vulnerability would eventually lead the FPGA-based system to be compromised.

The security threats from malicious FPGA design software is paid much less attention however, can cause serious problems. Here we use a simple example to demonstrate how significantly attackers can impact the FPGA configuration through the exploitation of a FPGA software. First, we connected a Xilinx FPGA board with a monitor through a VGA cable. Next, we implemented a functional module in the FPGA device to draw a "chess board" on a screen by sending a VGA signal to the monitor. Our attack goal was to modify this "chess board" without disturbing the functional module in Verilog and the user constraints for FPGA configuration.

In the process of the attack, we opened the project with the Xilinx FPGA editor, located the slice that controls the VGA pins of the board, and then we only modified a *single* logic function on that slice. Next, we generated the bitstream for the modified .ncd file output from the FPGA editor and download the bitstream to the Xilinx FPGA. As a result, the output picture became white bars, not a chess board. Meanwhile, the width of the bars is doubled compared to the original picture. The pictures displayed on the monitor for before and after attacks are shown in Figs. 4.1(a) and (b), respectively. In this demonstration, we manually performed the attack behaviors in the FPGA design suite, but the



(A) Before modification



(B) After modification

FIGURE 4.1: Monitor displays (A) before and (B) after modification

attack operations can certainly be implemented in a stand-alone software.

4.1.1 Three Levels of Attacks

More precisely, this work assumes that three levels of attacks can take place due to the malicious software implanted in the FPGA design suite.

- *L-1:* Based on attackers' experiences, an attacker places hardware Trojans in the most popular FPGA die area. At this level, the attacker does not have to have any knowledge of the design to be configured on the FPGA.
- *L-2:* The attacker is able to extract information like which slices are utilized by the current design from the FPGA placelist. Although the attack at this level does not analyze the exact function of the design, the attack exploration space is significantly smaller than the L-1 attacks.

- *L-3:* The malicious software can search for the identical portion of the design, which may be protected with duplication technique, and insert the same Trojan to each replica. The attack at this level is the most challenging one, but costing the attacker more resource to guarantee the success of attacks.

4.2 Existing Solutions

Since the security threats associated with FPGAs [4, 13, 34–37] were identified, countermeasures against those threats receive increasing attention. One category of countermeasure is to address the intellectual property (IP) theft issues during the FPGA deployment phase [11, 27, 32, 35]. Another category is to resist the attacks originated from malicious FPGA devices [38, 39]. The attacks on FPGAs in the existing work are mainly from untrusted IP designers, system integration engineers, or malicious end users [39]. Although the FPGA vendors [40] adopt bit encryption, authentication, and key/register zeroization techniques to prevent bitstreams from being tampered, those techniques do not thwart the design tampering happened *before* the bitstream is generated by the FPGA software. There are limited work addressing the security threats from malicious FPGA design software, which could harm the integrity of a design running on a SRAM FPGA device [4].

4.3 Proposed Method

We exploit the principle of moving target defense (MTD) as a mean to proactively address the security threats from malicious FPGA software. Different with the traditional MTD methods applied in the domain of cyber security, our proposed FPGA-oriented moving target defense (FOMTD) method explores the unpredictability of the way that a hardware

design is configured on FPGAs to deter attackers from precisely inserting hardware Trojans. More specifically, the key idea of FOMTD is to make the output of FPGA placement and routing unpredictable, such that attackers who mounts a malicious program on the original FPGA design suite cannot easily alter the original implementation on a FPGA. The proposed FOMTD is implemented by appending commands in user constraints file and modifying the original Verilog design files.*Note, our method does not guarantee to completely prevent all hardware intrusions but it will increase the difficulty of a Trojan successfully landing on one (or more) of the FPGA slices occupied by the design.*

The desired unpredictability can be achieved by the three defense lines provided by our method. In the domain of hardware (i.e. FPGA), we exploit the following configuration resources to realize the FOMTD method: (i) the availability of multiple replicas of the intended design, (ii) random selection of one replica for operation at runtime, (iii) random designation of FPGA slice positions for the selected lookup tables (LUTs), and (iv) hot-swappable submodules for runtime design assembling.

4.3.1 Defense Line 1 (DFL1): Slice Position Selection through User Constraints File

Method description

Instead of using default FPGA setting for placement and routing, we specify the slice positions on the FPGA die for the selected LUTs, so that the default design mapping on the FPGA grid can be modified. Figure 4.2 shows the effect of the proposed defense line 1. By specifying few LUTs (black squares in Fig. 4.2), we change the slice locations for the three parts of the intended design. This specification can be performed by appending command

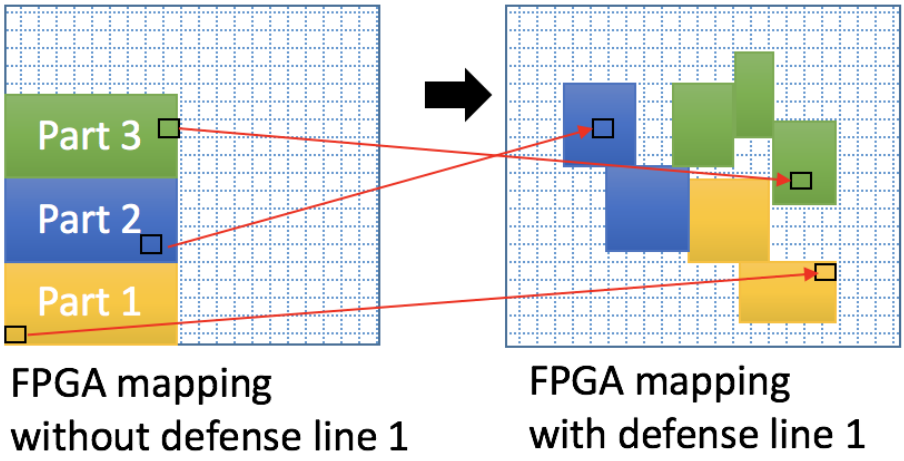
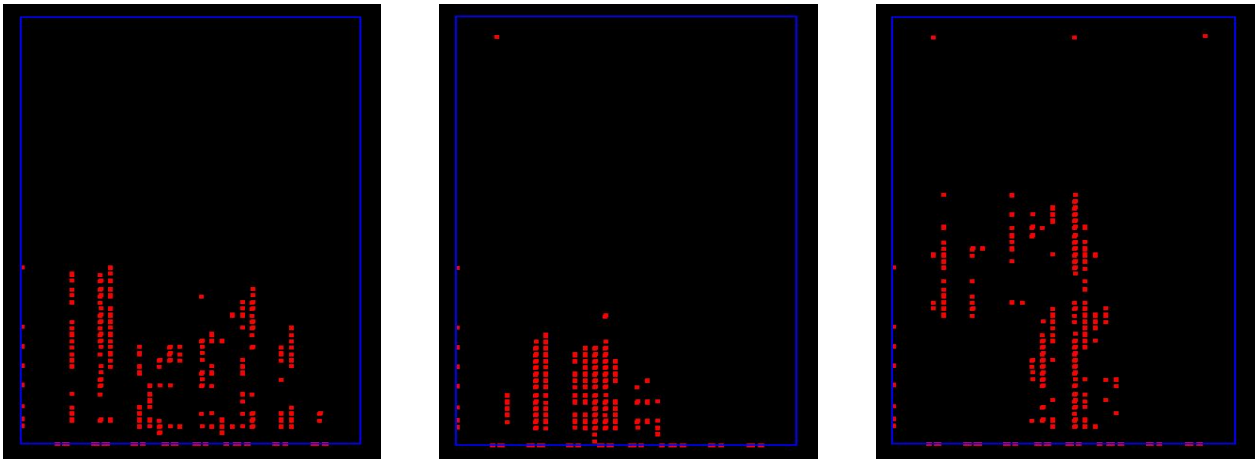


FIGURE 4.2: FPGA mapping modified by proposed defense line 1. Three parts in different colors represent three partitions of the intended design. Black squares are three LUT configurations. Proposed defense line 1 alters the default LUT mapping on the FPGA grid.

to the user constraints file, which is typically used to specify pin and timing constraints. *Note, the selection of slice positions is conducted by FPGA users at the FPGA deployment stage, which is after the implementation of the malicious FPGA software.* Hence, attackers (malicious software designers) will have hard time to decide where to place effective hardware Trojans. Blindly inserting Trojans may not effectively impact the design on the FPGA.

Case study

We used the ISCAS benchmark circuit c6288 as an example to show the effect of slice position specification. In the first case, we followed the default setting of the Xilinx ISE 14.1 to generate the placelist for c6288. In the second case, we chose *one slice position* for four randomly selected LUTs (we refer this is the *single-slice case*). In the third case, *three slice locations* are designated to twelves LUTs (we refer this is the *triple-slice case*). We can observe the design placement details in the FPGA editor, which is available in the Xilinx



(A) Default setting (B) Single-slice-selection (C) triple-slice selection

FIGURE 4.3: Design placement observed from the Xilinx FPGA editor for (A) default setting, (B) single-slice selection, and (C) triple-slice selection cases.

design suite.

Figure 4.3 shows the slice occupation results (red dots) for the three cases described above. From Fig. 4.3 we can see, our defense line 1 indeed significantly changes the design placement on the FPGA die. To quantify the location difference, we define a metric, *non-similarity rate*, to assess the degree of changes that have been made by our defense line 1. Non-similarity rate represents the ratio of the number of the LUT instances being placed to a new position due to our method over the total number of slices in use. A higher non-similarity rate obtained from less number of slice designations is better. In the case study of c6288, the single-slice case achieves a non-similarity rate of 33.2% and the triple-slice case increases the non-similarity rate to 35.9%. We expect that more slice specification can allow us to further improve the non-similarity rate and thus enhance the unpredictability of slice utilization.

Theoretical bound for defense line 1 thwarting different Trojan attacks

The baseline below is the original design without any protection. We assume that the intended baseline design occupies ϕ slices, the entire FPGA die is composed of Φ user controllable slices, and the control logic for replica selection is small enough (compared to ϕ) to be ignored for the simplicity of analysis. If a hardware Trojan will impact the design once it is triggered, we call a *Trojan Hit*. We define the hardware Trojan hit rate, Γ, as the probability that a randomly-picked slice is indeed one of slices utilized by the design. If an attacker blindly inserts a hardware Trojan to the FPGA die (i.e., blind attack), the Trojan hit rate is equal to Eq. (4.1).

$$\Gamma_{baseline_{vs.\,blind\,attack}} = \frac{\phi}{\Phi} \tag{4.1}$$

When the attacker has knowledge of the commonly used slice area (i.e. L-1 attack), the target FPGA area will be smaller than the entire FPGA die. We assume $\xi \in (0, 1)$ is the coefficient for how much Trojan insertion space is narrowed by the attacker based on the attacker's experience. Hereafter, we name ξ as the space coefficient of Trojan attack. The non-linear function $f(\xi)$ represents the degree of accuracy regarding whether the real design placement matches to the attacker's prediction. Now, the hardware Trojan hit rate for the design without any protection against L-1 attack is calculated in Eq. (4.2). If $f(\xi)$ reaches its maximum value, the entire design will be covered in the attack space, and $\Gamma_{baseline_{vs.\,L-1}}$ will decrease with the increasing space coefficient of Trojan attack ξ.

$$\Gamma_{baseline_{vs.\,L-1}} = \frac{\phi}{\Phi'} = \frac{f(\xi) * \phi}{\xi * \Phi} \tag{4.2}$$

When the L-2 attacker has the knowledge of the detailed slice utilization, each inserted hardware Trojan will absolutely impact the original design because the Trojan exploration space is equal to the injection space. This is expressed in Eq. (4.3). As the attacker has more understanding on the placed design, the $\Gamma_{baseline_{vs.\,L-3}}$ will be equal to Eq. (4.3).

$$\Gamma_{baseline_{vs.\,L-2}} = \frac{\phi}{\Phi''} = \frac{\phi}{\phi} = 1 \tag{4.3}$$

In contrast, our proposed defense line 1 (*DFL1*) does not use the default FPGA mapping settings. Thus, the target FPGA area remains as the entire FPGA die Φ. Our Trojan hit rate turns to Eq. (4.4). Comparing Eq. (4.2) and Eq. (4.4) we can see that, the denominator of Eq. (4.4) is larger than that in Eq. (4.2). Hence, our defense line 1 reduces the Trojan hit rate in the scenario of L-1 attack. Once the attacker knows the exact slice utilization, the proposed defense line 1 cannot thwart L-2 and L-3 attacks.

$$\Gamma_{DFL1\,vs.\,L-1} = \frac{f(\xi) * \phi}{\Phi} \tag{4.4}$$

4.3.2 Defense Line 2 (DFL2): Pseudo-Random Replica Selection

Method description

FPGA has a nature of reconfiguration and redundancy. We exploit this nature to implement the principle of MTD on FPGAs. The design to be implemented on the FPGA is first duplicated by n copies. However, only one of the replicas will be active at a time, and the rest of the replicas are inactive by using input gating technique. The replica selection and input gating are controlled by a pseudo-random selector, which is *not* a true random number generator. This is because we only have a limited number of replicas

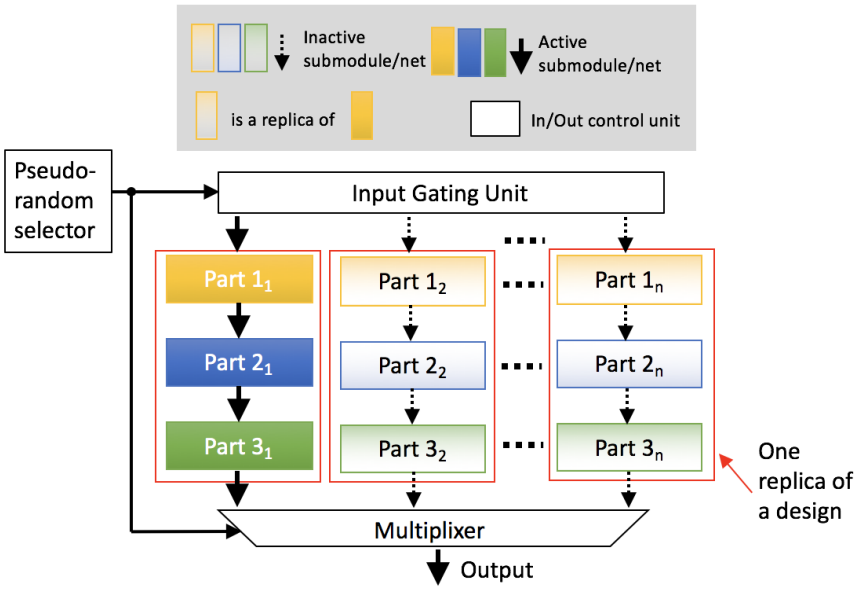


FIGURE 4.4: Pseudo-random replica selection provided by the proposed method.

on the FPGA, and thus the range of the random number is not large. A user-defined arbitrary logic function and a set of external inputs are good enough to pseudo-randomly choose one of the replicas. Figure 4.4 shows the concept of our defense line 2. Note, in this defense line, we do not have a comparison logic to examine the consistency among the n replicas for the purpose of power saving. As the fact that which replica will be active is determined after the FPGA configuration, an attacker (at L-1) needs to blindly place the hardware Trojan to the entire FPGA die to make a successful attack.

Theoretical bound for defense line 2 thwarting different Trojan attacks

Figure 4.5 depicts an example of exploration expansion by our proposed defense line 2. A complete design (including replication) consists of multiple units. Because of the slice position specification, the rough size of the Trojan exploration space S_{FOMTD} can be

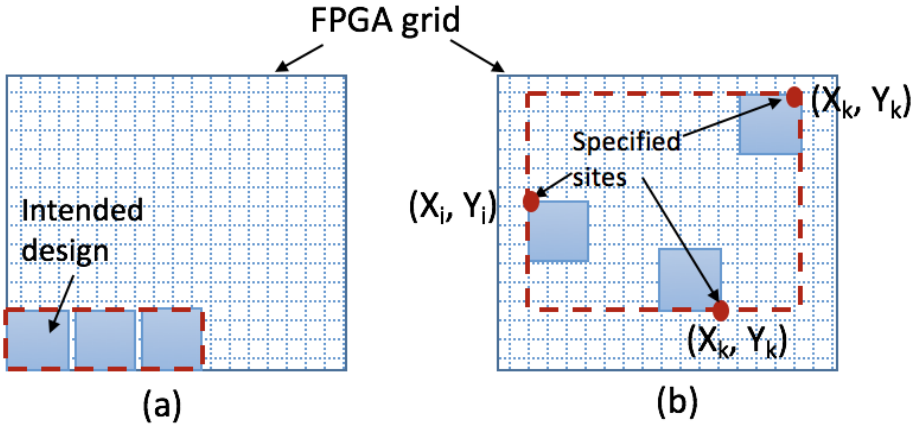


FIGURE 4.5: Hardware Trojan attack exploration space for (a) the design placement with default FPGA setting, (b) the design protected with FOMTD defense lines 1 and 2.

expressed by Eq. (4.5).

$$S_{FOMTD} = max\left(|X_i - X_j|\right) * max\left(|Y_i - Y_j|\right) \tag{4.5}$$

Compared to the baseline, our method achieves the theoretical worst-case hardware Trojan hit rate for L-2 and L-3 attacks as described in Eqs. (4.6) and (4.7), respectively. If L-2 attacks are taken place in the design, $\Gamma_{baseline_{vs.\,L-2}}$ increases to 1; in contrast, $\Gamma_{DFL1\&2\,_{vs.\,L-2}}$ remains low due to the expanded Trojan exploration space by the proposed defense line 2. The exact Trojan hit rate depends on the size of the design unit for duplication, ν. Under the condition of L-3 attack, our Trojan hit rate will not go beyond $1/n$ (theoretically the worst-case hit rate is a uniform distribution of random replica selections). In our simulation section, we observe that the actual Trojan hit rate of our method never reaches this upper bound due to many random factors.

$$\frac{\phi}{S_{FOMTD}} \leqslant \Gamma_{DFL1\&2\,_{vs.\,L-2}} \leqslant \frac{\phi}{n * \nu + (\phi - \nu)} \tag{4.6}$$

$$\Gamma_{DFL1\&2\,vs.\,L-3} \leqslant \frac{\phi}{n * \phi}(= \frac{1}{n}) \tag{4.7}$$

4.3.3 Defense Line 3 (DFL3): Runtime Design Assembling

Method description

Our defense line 3 is the hot-swappable submodule assembling technique, as shown in Fig. 4.6. We partition the original design into *m* submodules and each submodule is duplicated by *n* times. In the moment of interest, only one replica of each submodule will be assembled into a complete design. The pseudo-random selector is utilized to determine which replica to choose at runtime. After a period of time, the selection of submodule replicas will be changed without stopping the normal operation (i.e. hot-swappable assembling). The total number of design configurations we can achieve is n^m. This large number of configurations further increases the difficulty for the attacker to recognize the entire design for attack.

The hot-swappable assembling technique shown in Fig. 4.6 is directly applicable for combinational circuits. We tailor this technique to make it suitable for sequential circuits. As shown in Fig. 4.7, two styles are available for the circuit composed of combinational logic and memory elements. In style I, we do not duplicate the registers, thus the submodule assembling technique for combinational and sequential circuits is the same. In style II, the registers have replicas, too. To realize the hot-swappable feature, we copy the content of active registers to the hot-swap registers (*HS Reg.* in Fig. 4.7) before the runtime submodule swapping happens. Then, we load the value saved in *HS Reg.* to all register replicas to resume the operation after runtime submodule swapping.

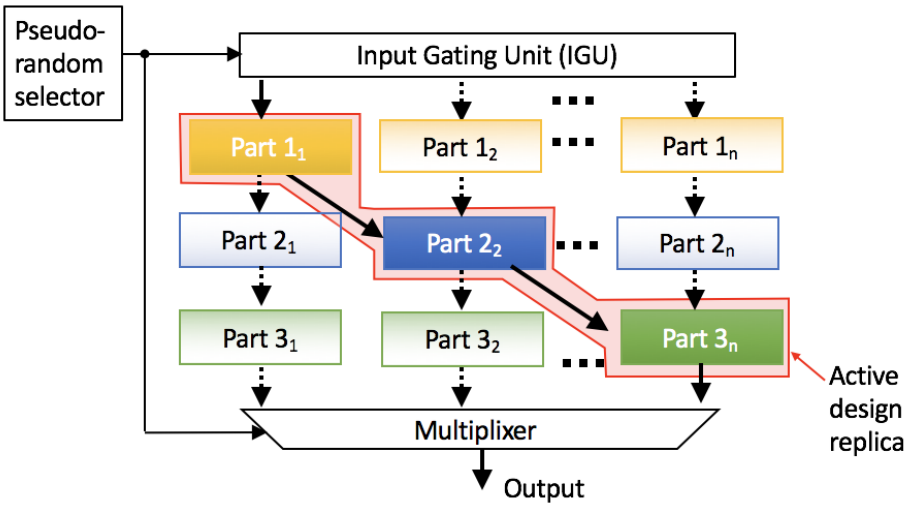


FIGURE 4.6: Hot-swappable submodule assembling provided by defense line 3.

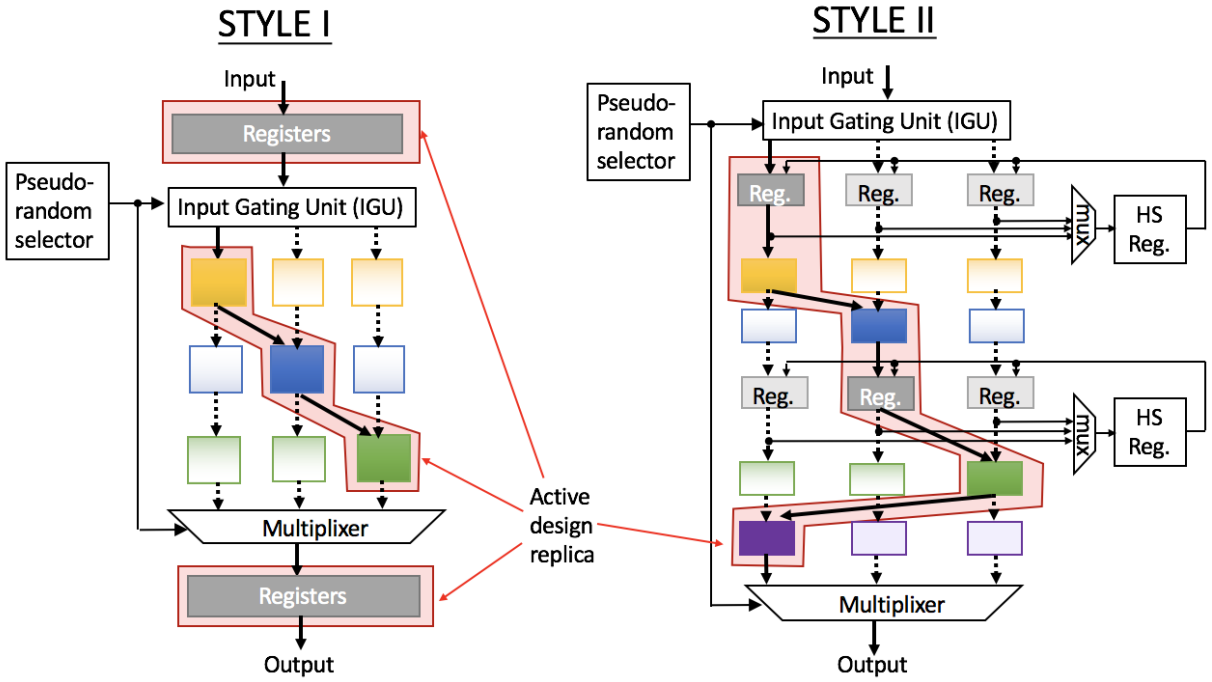


FIGURE 4.7: Two styles of applying defense line 3 to sequential circuits.

Additional option 1: input gating. To thwart L-3 attacks, we could further strengthen our defense line 3 by loosing the input gating and enabling two complete replicas active, such that the two replicas can examine the consistency between their final outputs. However, the enhanced defense capability comes with more power consumption.

Additional option 2: gate replacing on replicas. To better defeat L-3 attacks, we enhance our defense line 3 by bringing diversity to the replicas for hot-swappable submodules. In the work [39], the diversity on implementation is introduced by using different hard macros, which are obtained by applying different constraint conditions during FPGA synthesis. Inspired by the work [39], we create hard macros at gate level so that we have more flexibility to facilitate the implementation of heterogeneous replicas for submodules. Those gate-level hard macros are used to replace some gates in one of the replicas. As a result, even if an attacker searches the same FPGA configuration pattern between two replicas, the success rate of finding two identical copies for Trojan insertion will be extremely low.

The flowchart for the proposed gate replacing on replicas is depicted in Fig. 4.8. First, we randomly choose one (or more) type(s) of logic gates, for instance *nand (c, a, b)*, in one replica. Next, we apply the de Morgan's laws to replace the chosen gate with other types of logic gates, while maintaining the same Boolean function. For the 2-input nand gate, we can replace it with *or (c, $\sim a$, $\sim b$)*. Note, all the gate replacement is done in the Verilog description. To prevent the FPGA synthesis tool from removing our gate replacement during the logic optimization process, we implement the *or (c, $\sim a$, $\sim b$)* with three customized hard macros, *HM_OR ($\bar{a}$, $\bar{b}$, c)*, *HM_NOT (a, $\bar{a}$)* and *HM_NOT (b, $\bar{b}$)*. *HM_OR* and *HM_NOT* are defined as Verilog modules which will complete the logic OR and inversion operations. By using hard macros, the gates for replacement can be mapped

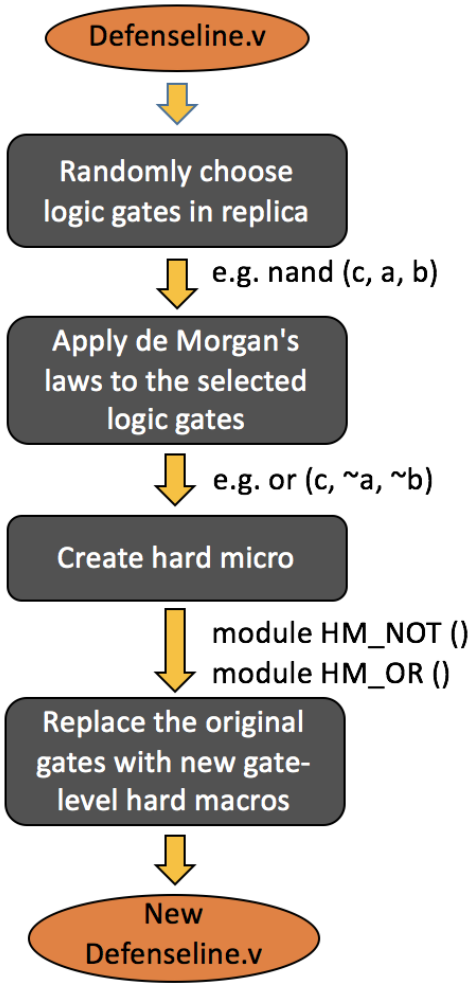


FIGURE 4.8: Gate replacement for the security enhancement of defense line 3.

into one independent slice and they will not be merged with other LUT configuration. We can conduct gate replacement for one or multiple replicas so that the identical LUT configurations will be removed. Consequently, our enhanced defense line 3 is capable to thwart L-3 attacks.

Theoretical bound for defense line 3 thwarting different Trojan attacks

With respect to L-2 attacks, the attacker knows which slices are occupied by the design but cannot differentiate which submodule belongs to which replica. Hence, the target slice for Trojan insertion is not clear. The attacker has to randomly chooses ϕ slices out of all the occupied slices $n * \nu + (\phi - \nu)$. Because defense line 3 changes the complete design by re-assembling the submodules from different replicas, only the Trojan placed in the common non-duplicated area $(\phi - \nu)$ will lead to Trojan hit constantly. The corresponding Trojan hit rate for this scenario is expressed in Eq. (4.8).

$$\Gamma_{DFL3_{vs.\,L-2}} = \frac{(\phi - \nu)}{n * \nu + (\phi - \nu)} \tag{4.8}$$

In L-3 attacks, the attacker has full knowledge of which slices are configured for the design with the defense line 3, but he/she could only form the complete design by guessing which submodule replicas will be used. Without gate replacement, the corresponding Trojan hit rate is shown in Eq. (4.9).

$$\Gamma_{DFL3_{vs.\,L-3}} = \left(\frac{(n-1)^m}{n^m}\right)^{sp} \tag{4.9}$$

In which, m and n are the number of submodules per design and the number of design replicas, respectively. sp is the number of different hot-swapping configurations. As the proposed defense line 3 changes design configuration over time, there are n^m configurations in total and $(n-1)^m$ configurations do not contain the inserted Trojan. The more swapping happens during the runtime operation, the less Trojan hit rate the attacker could achieve.

4.4 Experimental Results

4.4.1 Experimental Setup

In the following experiments, we synthesized, placed and routed the Verilog HDL codes for four ISCAS'85 and ISCAS'89 benchmark circuits through the Xilinx ISE 14.1 design suite, and generated the corresponding bitstreams. Those bitstreams are specific for a Xilinx Spartan-6 XC6SLX16 FPGA. The detailed slice utilization of each circuit was analyzed by our Python script to extract the occupied FPGA slice positions. We used MATLAB programs to insert hardware Trojans blindly or purposely (depending on the experiment goal) and then measured the hardware Trojan hit rate. We assume the Trojan inserted in the design will be triggered and affect the original design function. The FPGA slice utilization and worst-case delay were obtained from the tools available in the Xilinx design suite.

4.4.2 Variation on FPGA Slice Utilization

Variation on the position of all the slices occupied by the design is critical to ensure the high unpredictability of FPGA utilization and the success of our method. Hence, we first examined the impact of our defense line 1 on the FPGA slice utilization. We compared the positions (on the FPGA die) of all the slices used by the baseline design and the one applied user-specified slice designations. The baseline means the original benchmark circuits without any protection. The *non-similarity rate* defined in Section V.B.2) is adopted as a metric for evaluation.

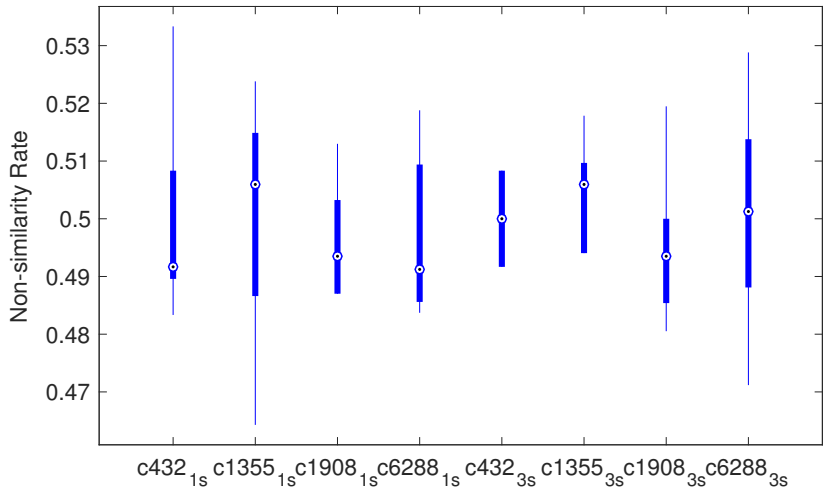


FIGURE 4.9: Non-similarity rate achieved by proposed defense line 1. Non-similarity rate between one slice-position designation case and the baseline. The subscripts 1s and 3s means the location of a single slice or three slices are specified in the user constraints file for the FPGA implementation. On each bar, the central mark indicates the median, and the bottom and top edges of the box indicate the 25^{th} and 75^{th} percentiles, respectively.

As shown in Fig. 4.9, compared to the baseline, our method achieves an average non-similarity rate in the range of 0.49 to 0.51. This means, on average, about 50% of the LUT instances for each benchmark circuit being placed to different positions on the FPGA die due to our defense line 1. We further examined the variation on the different positions of the occupied slices due to the different designated destinations on the FPGA die.

We repeated the simulation on non-similarity rate for sequential circuits and summarized the median values for all non-similarity rates in Table 4.1. As shown, the proposed defense line 1 approximately achieves a non-similarity rate of 0.5. The increase on the number of user specified slice locations only slightly enhances the non-similarity rate. Each non-similarity rate in Table 4.1 was based on five test cases. According to our case study, we observe that the difference on the specified slide locations yields the average

TABLE 4.1: Medians of Non-Similarity Rate

Circuits	$c432_{1s}$	$c1355_{1s}$	$c1908_{1s}$	$c6288_{1s}$	Std. deviation
Median	0.49167	0.50595	0.49351	0.49123	0.0070
Circuits	$c432_{3s}$	$c1355_{3s}$	1908_{3s}	$c6288_{3s}$	Std. deviation
Median	0.5000	0.50595	0.49351	0.50125	0.0051
Circuits	$s344_{2s}$	$s526_{2s}$	$s1488_{2s}$	$s13207_{2s}$	Std. deviation
Median	0.48333	0.42105	0.4878	0.43367	0.0340

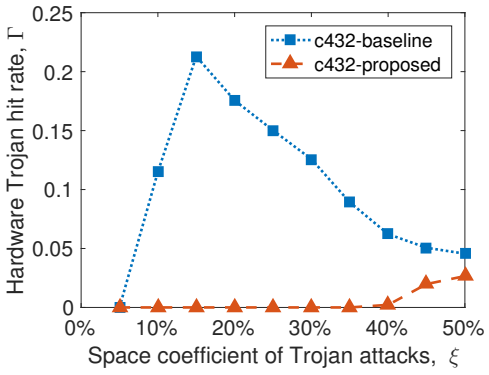
standard deviation on the median value in the range of 0.0070 to 0.034, which is very small.

4.4.3 Assessment on Attack Resilience

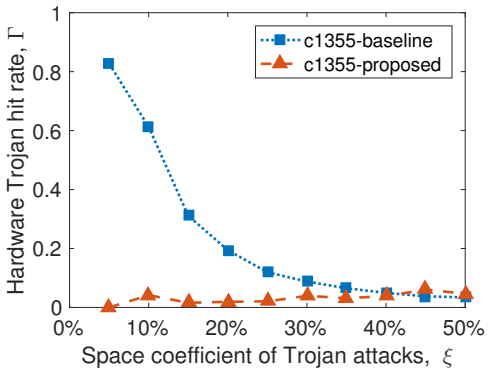
The attack resilience of baseline and our method are compared in the subsections below. Three attack levels mentioned in Section 4.1.1 are considered in the following assessment.

Hardware Trojan Hit Rate for L-1 Attacks

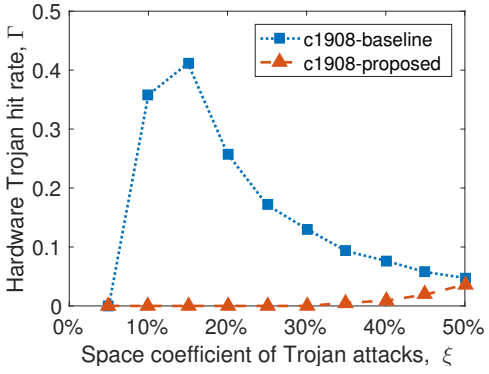
Recall that attackers who execute L-1 attacks do not know the locations of all the occupied slides for the design of interest. We varied the range of attack exploration space from 5% to 50% of the entire FPGA die in the following experiments. Figure 4.10 shows that the proposed method achieves a lower hardware Trojan hit rate Γ (defined in Section 4.3.2) than the baseline in a wide range of the attack exploration space. This is because our defense line 1 makes the LUT placement unpredictable and not targetable for L-1 attackers. The hardware Trojan hit rate for c432, c1908, c6288, s444, and s13207 first increases with the increasing ξ because $f(\xi) * \phi$ in Eq.(4.2), the number of occupied slices falling in the attack space, grows faster than $\xi * \Phi$, the attack space. As the maximum value of $f(\xi)$ is 1, $\Gamma_{baseline}$ starts to drop after ξ exceeds a threshold. In our case studies, the ξ thresholds for



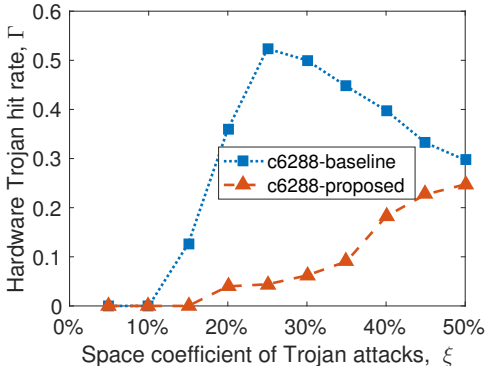
(A) c432



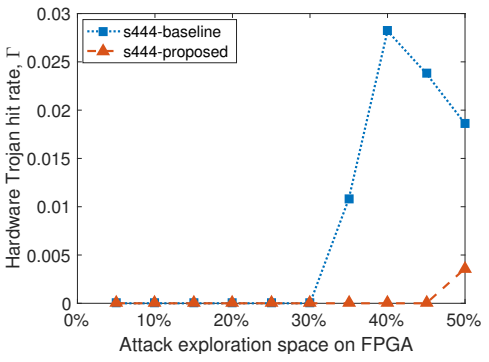
(B) c1355



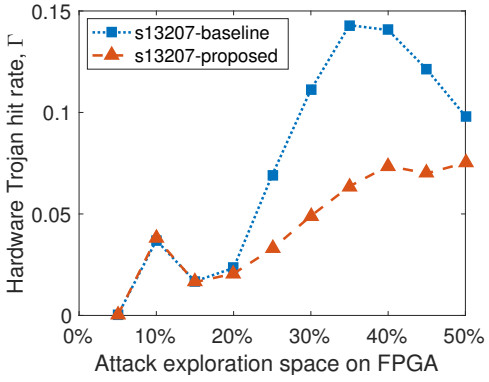
(C) c1908



(D) c6288



(E) s444



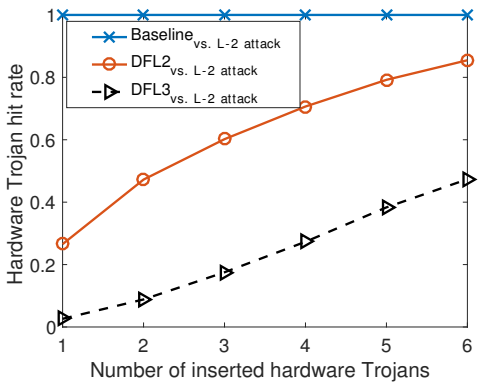
(F) s13207

FIGURE 4.10: Hardware Trojan hit rate reduction by proposed defense line 1 applied in the benchmark circuit in the condition of L-1 attacks.

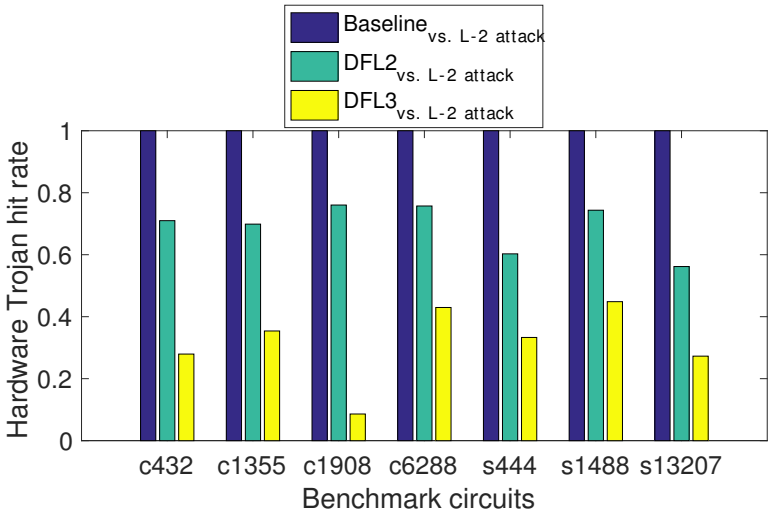
c432, c1355, c1908, c6288, s444, and s13207 are 15%, 5%, 15%, 25%, 40%, and 35%, respectively. The case of c1355 has a smaller ξ threshold than the other benchmark circuits, so we do not observe that the corresponding $\Gamma_{baseline}$ increases with ξ. The hardware Trojan hit rate of our method increases much slower with the increasing ξ than the baseline. Our method reduces the hardware Trojan hit rate to 0.213, 0.8272, 0.4114, 0.49, 0.0036, 0.0752 for c432, c1355, c1908, c6288, s444, and s13207, respectively. When the attack exploration space is large enough to cover the entire design placed on the FPGA die, the Trojan hit rate of proposed method will be equal to the Trojon hit rate of the baseline eventually.

Hardware Trojan Hit Rate for L-2 Attack

Different with L-1 attacks, L-2 attacks are able to retrieve the exact locations of the occupied slices. Consequently, the baseline design does not have any resilience against L-2 attacks. The proposed defense line 2 (DFL2) activates one complete design replica according to the pseudo-random selection and defense line 3 (DFL3) assembles the hot-swappable submodules at runtime . Thus, our method further increases the unpredictability of the truly activated design copy and achieves a lower Trojan hit rate over the baseline. As shown in Fig. 4.11(a), the baseline without any protection yields a hardware Trojan hit rate of 1, which means each triggered Trojan is definitely located in one of the occupied slices. In contrast, our DFL2 and DFL3 significantly reduce the Trojan hit rate over the baseline especially for the small number of inserted Trojans. When more Trojans are placed in the utilized FPGA slices, our Trojan hit rate eventually increases due to the limited number of design replicas used (we used two replicas in our experiment). If more copies are available, the slope of the Trojan hit rate will be less than what is shown in Fig. 4.11(a).



(A) c432



(B) Seven benchmark circuits

FIGURE 4.11: Hardware Trojan hit rate for (A) c432, and (B) seven benchmark circuits suffering from four hardware Trojans inserted via L-2 attacks.

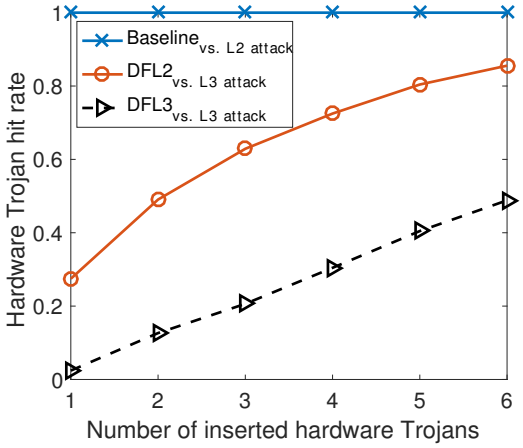
We examined the Trojan hit rate for seven benchmark circuits, which suffer from different numbers of Trojan insertions via L-2 attacks. Each hardware Trojan hit rate was obtained from 10,000 test cases. The average Trojan hit rate of DFL2 (DFL3) is 69% (31%). As shown in Fig. 4.11(b), the DFL2 reduces the hit rate by up to 40% over the baseline. The reduction on the Trojan hit rate can be further improved by DFL3 up to 91%.

Hardware Trojan Hit Rate for L-3 Attack

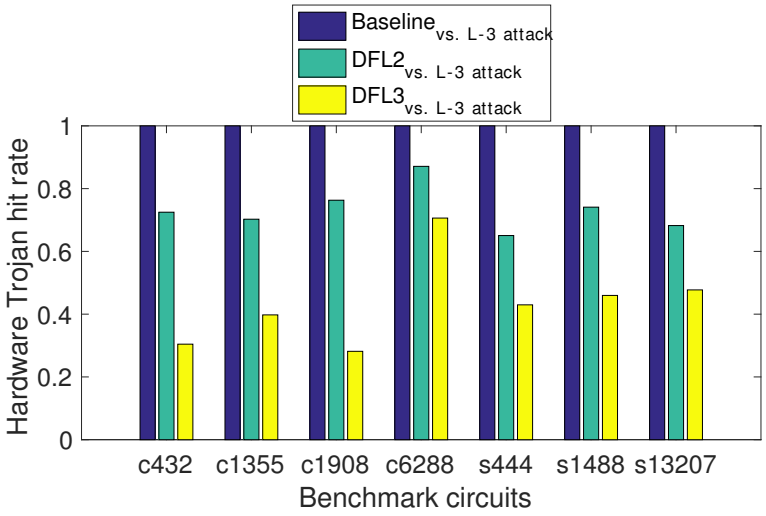
The L-3 attack can recognize the multiple replicas of the design that will be implemented on the FPGA by searching for the exact same or approximately similar LUT configuration. We repeated the same experiments as we did for Section 4.4.3, except a different attack level. As shown in Fig. 4.12(a), the Trojan hit rate for the design under L-3 attack increase with the increasing number of Trojans, which is similar with that for the L-2 attack case. However, the average Trojan hit rate of DFL2 (DFL3) increases to 73% (44%), which is higher than that resisting L-2 attacks. As shown in Fig. 4.12(b), the DFL2 reduces the hit rate by up to 35% over the baseline. The reduction on the Trojan hit rate can be further improved by DFL3 up to 72%.

From Figs. 4.11(b) and 4.12(b), we can also conclude that L-3 attack indeed is more powerful than L-2 attack. This is due to L-3 attack can search for the matched LUT configuration pattern. We subtracted the Trojan hit rate for DFL3 against L-3 attack from that against L-2 attacks and calculated the increase on the hit rate in Fig. 4.13. The peak of Trojan hit rate increase appears at the number of inserted Trojans equal to 2. When more Trojans are introduced to the design, the increase on Trojan hit rate gradually reduces. The reason for this phenomenon is, statistically, the number of exactly matched LUT configurations due to the application of replicas is approximate to 2. This means, for some cases in L-3 attack, two hardware Trojans can be respectively placed to two identically configured LUTs. Thus, those cases can successfully launch the Trojan without being detected.

Figure 4.14(a) shows that the average number of exactly matched LUT configurations per each benchmark circuit is close to 10^0 (i.e. 1). If we search for the LUT configuration with similar format but different input/out pins (i.e *approximate matching*), the number of



(A) c432



(B) Seven benchmark circuits

FIGURE 4.12: Hardware Trojan hit rate for (A) c432, and (B) seven benchmark circuits suffering from four hardware Trojans inserted via L-3 attacks.

matched cases increases. To address this issue, we apply the gate replacement technique to the defense line 3. As can be seen from Fig. 4.14(a), our enhanced method can increase the number of exact matching LUT configurations for two replicas. Now, the same LUT configurations do not stand for the identical logic function for the benchmark circuit any

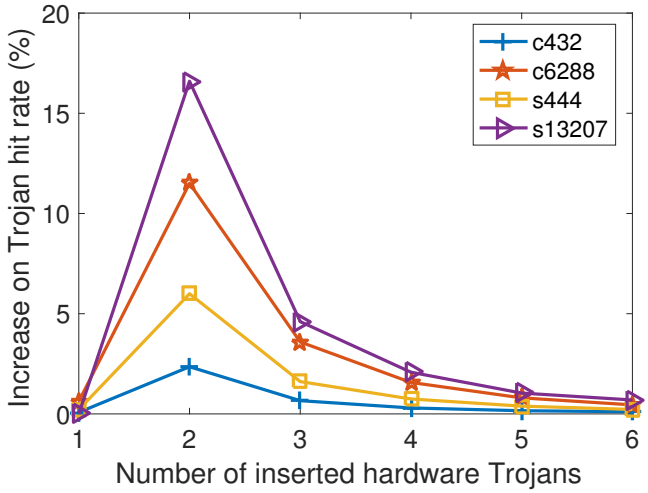
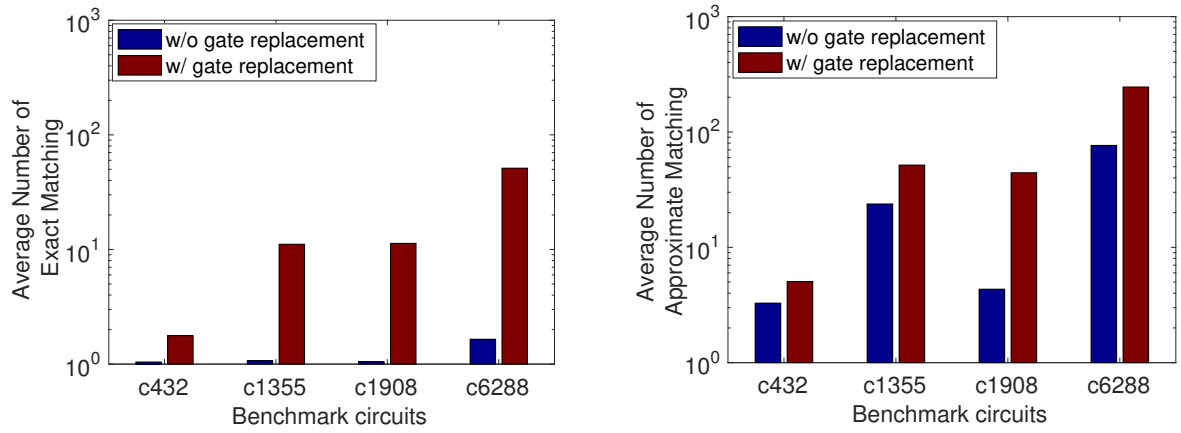


FIGURE 4.13: Increase on hardware Trojan hit rate due to advanced attacks.

more. Therefore, when the attacker performs L-3 attacks, the Trojan hit rate due to L-3 attack can be reduced. Not only increasing the number of exact matching cases, our gate replacement technique also increases the number of approximate matching patterns, as shown in Fig. 4.14(b). As a result, our enhanced DFL3 reduces the hardware Trojan (HT) hit rate. From Fig. 4.15 we can see, the proposed gate replacement technique reduce the Trojan hit rate for different circuits. On average, our method makes the Trojan hit rate decrease by 62% and 88% for the attacker searches for exact matching and approximate matching configures, respectively.

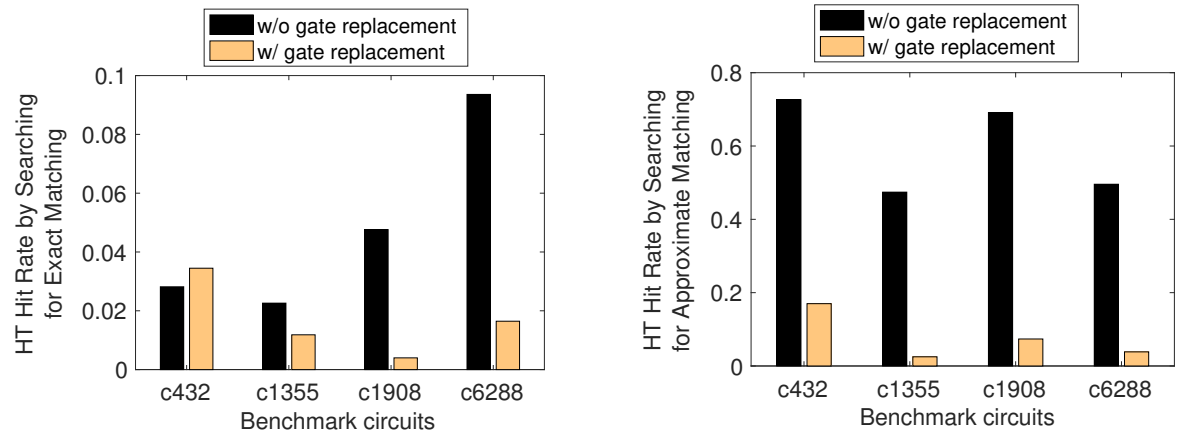
4.4.4 Dependent Design Factors on Trojan Hit Rate

In proposed defense line 3, our method swaps the replicas of submodules at runtime. We examined the impact of number of hot swaps on the Trojan hit rate. As depicted in Figs. 4.16(a) and (b), a larger number of hot swaps used in the design yields a lower hardware Trojan hit rate. However, as the number of inserted hardware Trojans increases,



(A) Exact matching (B) Approximate matching

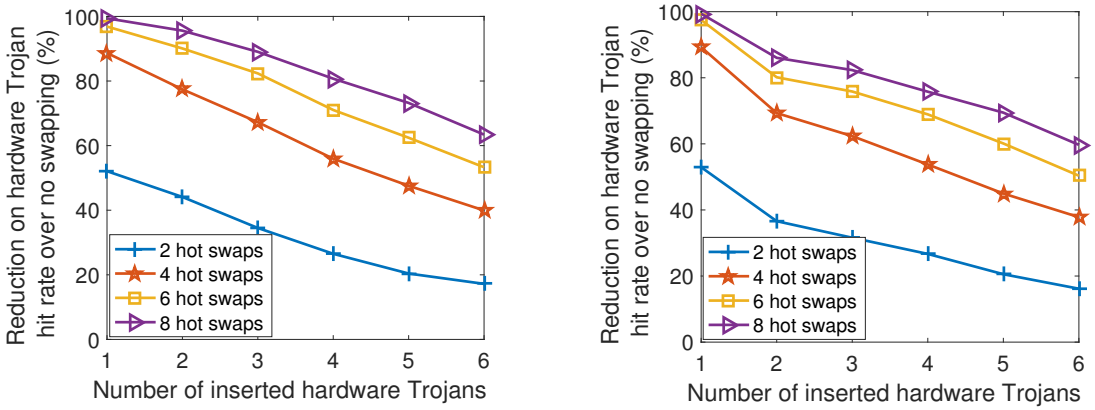
FIGURE 4.14: Comparison of number of Trojan hits for without and with gate replacement to thwart L-3 pattern searching attack. (A) Exact matching and (B) Approximate matching.



(A) Exact matching (B) Approximate matching

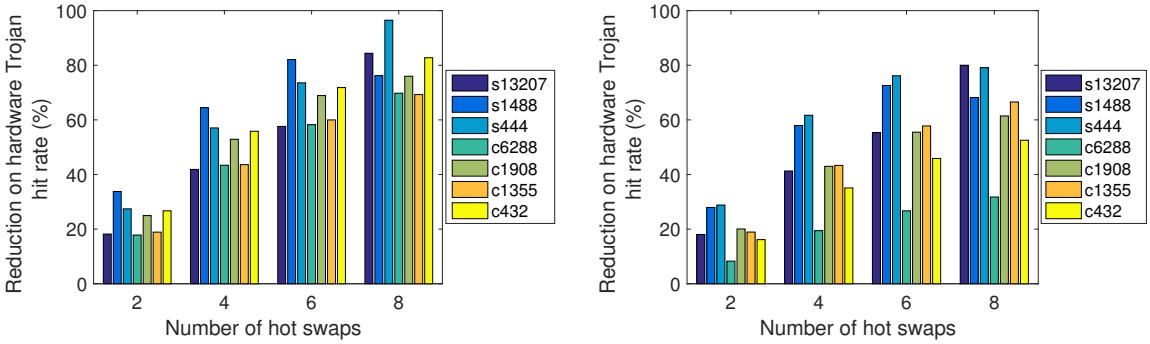
FIGURE 4.15: Comparison of hardware Trojan hit rate for without or with gate replacement to thwart L-3 pattern searching attack. (A) Exact matching and (B) Approximate matching.

the Trojan hit rate reduced by hot swapping gradually decreases. This conclusion applies to all benchmark circuits we tested, which is confirmed in Figs. 4.17(a) and (b).



(A) L-2 attack (B) L-3 attack

FIGURE 4.16: Impact of number of hot swaps on hardware Trojan hit rate for c432 under (A) L-2 attack, and (B) L-3 attack.



(A) L-2 attack (B) L-3 attack

FIGURE 4.17: Impact of number of hot swaps on hardware Trojan hit rate for seven benchmark circuits affected by four hardware Trojans inserted via (A) L-2 attack, and (B) L-3 attack.

4.4.5 Assessment on Hardware Cost, Delay and Power

Hardware Utilization

Table 4.2 summarize the number of utilized LUTs for different methods. Since the proposed defense line 1 only changes the location of designated slices, on average, our

TABLE 4.2: Number of FPGA LUTs utilized by different methods.

Circuits	c432	c1355	c1908	c6288	s444	s1488	s13207
Baseline	58	62	58	530	33	117	180
Defense line 1	58	62	59	530	33	117	181
Defense line 2	158	156	178	1123	67	261	429
Defense line 3.G	173	167	216	1157	84	296	443
Defense line 3.NG	110	158	181	1118	96	259	433

method consumes 0.33% more LUTs than the baseline. In the proposed defense line 2, we duplicated the design under protection once and utilized a pseudo-random selection unit for replica selection. The unselected replica is muted through input gating. For the small circuits, the increase on the LUT utilization could be large due to the relative large size of pseudo-random selection and input gating. However, the object for protection is large, the FPGA overhead can be reduced through optimization. The LUT overheads for the largest combinational circuit c6288 and sequential circuit s13207 in our case studies are 111.89% and 138.33%, respectively. During the hot-swapping process, the proposed defense line 3 interleaves multiple sections of the original design and its replica. In addition to the primary inputs, the input gating technique is also applied to the inputs for hot-swappable submodules. As a result, the LUT overheads for c6288 and s13207 increase to 116.79% and 145%, respectively. When we remove the input gating (i.e. NG) option, the corresponding overheads on utilized LUTs for the largest circuits are reduced to 10.94% and 140.56%, respectively. Certainly, removing the input gating will cost more power consumption. Although our method (defense line 3) incurs similar LUT utilization for double modular redundancy, our runtime replica selection ensures lower power consumption than DMR and provides good unpredictability.

TABLE 4.3: FPGA Total Power Consumption (mW) by Different Methods

Circuits	Baseline	Defense line 2	Defense line 3
c432	10.37 (100%)	11.05 (107%)	11.82 (114%)
c1355	48.66 (100%)	50.56 (104%)	50.56 (104%)
c1908	40.14 (100%)	42.50 (106%)	43.56 (109%)
c6288	217.41 (100%)	232.75 (107%)	233.67 (107%)
s444	20.01 (100%)	21.68 (108%)	21.85 (109%)
s1488	12.50 (100%)	15.25 (122%)	15.56 (124%)
s13207	303.33 (100%)	329.03 (108%)	335.07 (110%)

Power Consumption

We synthesized the Verilog code for the four benchmark circuits in the Synopsys Design Compiler. The clock frequency was set to 100 MHz for all the designs. We measured the power consumption in the tool Design Compiler and reported in Table 4.3. On average, the proposed defense line 2 leads to an increase on total power of 8.86% over the baseline. Our defense line 3 with input gating provides better resilience against advanced attacks, at the cost of consuming 11% more total power, on average, than the baseline. The increased power consumption is due to the pseudo-random selection and input gating logic, as well as the multiplexers before the final outputs.

Worst-case Delay

We measured the worst-case delays for different designs using the PlanAhead tool in Xilinx ISE 14.1 design suite. As shown in Table 4.4, slice designation used in the proposed defense line 1 could lead to more or less worst-case delay, depending on where the slice is designated. To examine the impact of the slice designation on the worst-case delay, we

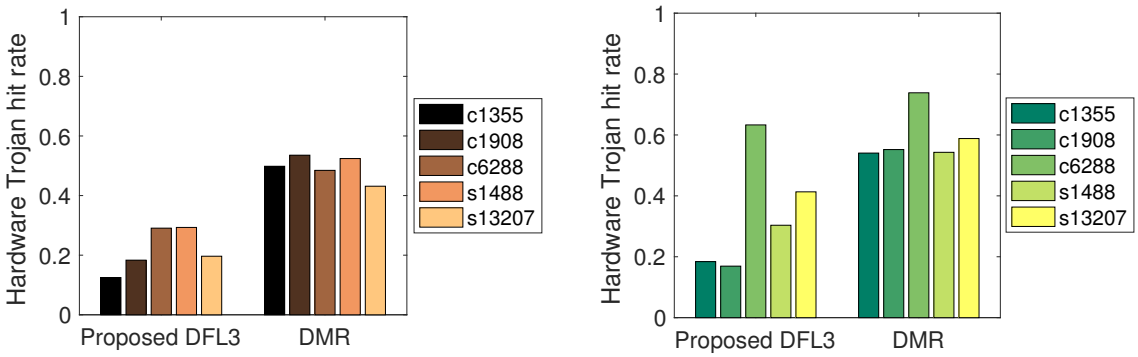
TABLE 4.4: Comparison of worst-case delay. Unit: ns

Circuits			c432	c1355	c1908	c6288	s444	s1488	s13207
Baseline			5.659	4.677	5.241	10.181	1.43	4.105	3.328
Defense line 1	single-slice designation	case 1	5.713	4.677	5.500	10.128	1.314	4.051	3.328
		case 2	5.603	4.677	5.458	10.358	1.322	3.997	3.274
		case 3	5.549	4.677	5.322	10.18	1.43	3.947	3.274
		case 4	5.711	4.622	5.257	10.013	1.322	3.979	3.274
		case 5	5.657	4.679	5.278	9.905	1.376	4.049	3.328
		+/- delay	-1.94%∼0.95%	-1.18%∼0	0∼0.49%	-1.65%∼1.74%	-0.81%∼0	-3.85%∼0	1.62%∼0
	triple-slice designation	case 1	5.607	4.57	5.287	10.234	1.378	4.009	3.272
		case 2	5.715	4.731	5.406	9.966	1.378	3.979	3.328
		case 3	5.553	4.679	5.335	9.979	1.378	4.049	3.328
		case 4	5.661	4.677	5.448	10.51	1.378	4.049	3.328
		case 5	5.606	4.669	5.334	9.899	1.322	4.009	3.272
		+/- delay	-0.96%∼1.93%	0%∼3.52%	0∼3.05%	-3.27%∼2.7%	-4.06%∼0	-0.75%∼1%	0%∼1.17%
Defense line 2			6.164 (+8.92%)	5.249 (+12.23%)	5.702 (+8.80%)	10.612 (+4.23%)	1.578 (+10.35%)	4.699 (+14.47%)	3.900 (+17.19%)
Defense line 3			6.528 (+15.36%)	5.707 (+22.02%)	6.177 (+17.86%)	10.925 (+7.31%)	1.637 (+14.48%)	4.785 (+16.57%)	3.900 (+18.81%)

varied the number of designated slices from 1 to 3, and performed five test cases for each designation condition. Based on our case studies, the proposed defense line 1 induces a delay overhead as large as 1.74% and 3.52% for the single-slice designation and triple-slice designation, respectively. Compared to the baseline, our defense line 2 leads to the worst-case delay increase in the range of 4.23% to 17.19% for different benchmark circuits. Due to the hot-swapple logic, the delay overhead induced by the proposed defense line 3 can be as high as 22.02%.

4.4.6 Comparing Proposed FOMTD with Static Trojan Detection Method

In this section, we compare our proposed moving target defense method with a static Trojan detection method, which is based on double modular redundancy (DMR). The DMR based static Trojan detection increases the number of LUTs. Even though the attacker who is performing L-2 attacks can see the utilized LUTs, the chance of hitting two identical LUTs is low. Some Trojans inserted on the replica comparison logic will not be



(A) L-2 attacks (B) L-3 attacks

FIGURE 4.18: Comparison of hardware Trojan hit rate for proposed defense line 3 and DMR affected by four Trojans inserted via (A) L-2 and (B) L-3 attacks.

detected by DMR. Therefore, the Trojan hit rate cannot be reduced to zero. When we advance the attack method to L-3 attack, our defense line 3 effectively reduces the Trojan hit rate. Together with the runtime hot-swapping feature, fewer number of exact matching LUT configurations available in the netlist of our method benefits us to reduce the success rate of a Trojan inserted by L-2 and L-3 attacks. Figures 4.18(a) and (b) indicate that our method achieves a lower Trojan hit rate than DMR. On average, our defense line 3 reduces the Trojan hit rate by 63.25% and 42.51% against L-2 and L-3 attacks, respectively. Indeed, L-3 attack can search for the identical LUT configurations; unfortunately, the number of exact matching LUT configurations is not high in FPGA mapping (which is different with ASIC design). Figure 4.19 shows that our defense line 3 can effectively reduce the number of exact matching cases over DMR. Thus, our defense line 3 obtains a better attack resilience than DMR. Defense line 3 also reduces the power consumption, as indicated in Fig. 4.20.

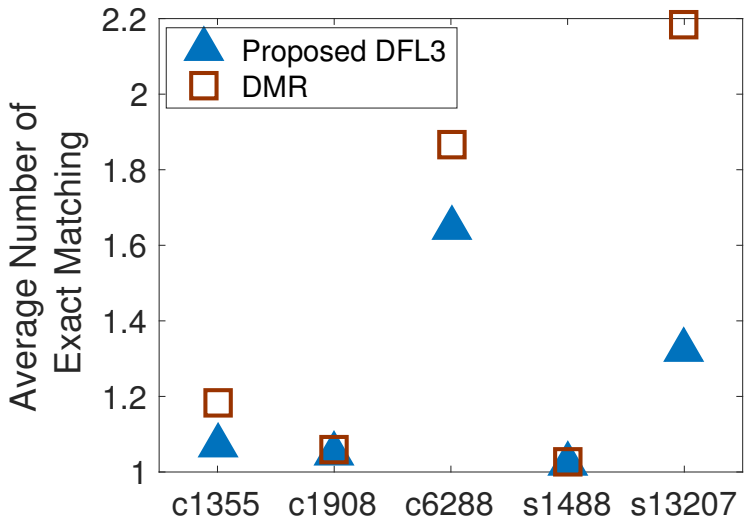


FIGURE 4.19: Comparison of number of exact matching on LUT configuration.

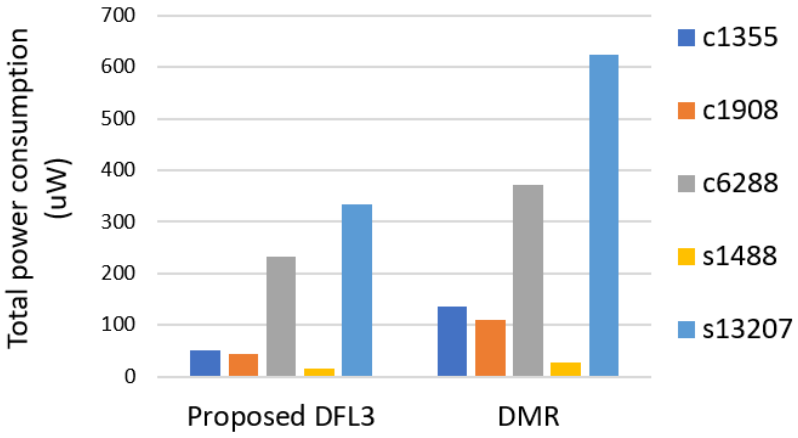


FIGURE 4.20: Comparison of power consumption between proposed DFL3 and DMR.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The study and usage of FPGA systems have reached a very high level. FPGAs are able to work as crucial components to build up large complicated electronic systems and they have played important roles in the applications of many fields including civil use, industry, military, etc. We should keep sober on the dramatic increasing trend of the popularity of FPGA systems because it can also attract attackers' attention for high improper benefits. To protect FPGAs from malicious attack and also maintain a safe operating environment for them, many defense schemes have been proposed. However, most of the existing technologies ignore one blind spot in the whole FPGA supply chain, which is the potential security threat from FPGA design software that is commonly called CAD tool. In this thesis, we fill the gap to reveal the CAD-based security threats and exploit the principles of moving target defense to propose a series of countermeasures against the potential attacks.

In Chapter 3, we validate the feasibility of pin grounding and further extends it to a runtime scheme. Our method not only grounds the FPGA I/O pins at the configuration

time, but it also checks the pin status during the operation time. To thwart the FPGA Trojan configured by an untrusted FPGA vendor or malicious CAD tools, we propose an HMTD method. In addition to replicating the functional module to replace the obsolete component for the legacy system, our method further specifies the relative physical distance between two replicas in the FPGA user constraint file and performs output comparison from two randomly selected replicas. Because the CAD tool cannot foresee the user constraint file, our method can effectively detect FPGA Trojans inserted by the CAD tool through implicit settings at the development time of that tool. Our experimental results show that the proposed HMTD method reduces the hardware Trojan bypass rate by 61% (on average) lower than the existing ATMR method. Our RPG scheme increases the FPGA utilization rate below 0.1%.

In Chapter 4, we introduce the proposed FOMTD, which is composed of three defense lines to defend three levels of attacks in terms of the level of attacker's knowledge to the design. Experiment results show that proposed FOMTD can effectively reduce the hardware Trojan hit rate with very reasonable overhead. More precisely, defense line one of FOMTD countermeasure reduces the hardware Trojan hit rate of level one attack to as low as 0.36% with only 0.33% more LUT consumption. Defense line two further reduces the Trojan hit rate up to 40% (35%) over the baseline for level two attack (level three attack) while consumes 14.68% more power. Defense line three reduces the Trojan hit rate up to 91% (72%) for level two attack (level three attack) with 15.51% more power cost. For the overhead of worst-case delay, the measurements vary because of the different slice designations. However, the highest delay increment for all the test cases is still under 23%. Comparing to DMR, our proposed defense two consumes less power and defense three has a better effect in controlling the hardware Trojan hit rate. Furthermore,

proposed enhancement of losing input gating of defense line three can achieve an even better security performance and the gate replacing enhancement can significantly mitigate the ability of attackers to locate identical locations for multiple-Trojan insertion. On average, our method makes the Trojan hit rate decrease by 62% and 88% for the attacker searches for exact matching and approximate matching configures, respectively. When advanced attacks is further made, our defense lines maintain the hit rate as low as 35%. Considering the significant improvement on the resilience on Trojan attacks, the overhead of our method is moderate and acceptable for security critical applications.

5.2 Future Work

5.2.1 Reduce the Delay Overhead Caused by Implementing HMTD

In Chapter 3, we introduced a hardware MTD method to thwart the attacks from malicious CAD tool. One limitation of the scheme is the delay overhead. In the future work, more attention can be paid to optimize the timing problem of HMTD while keeping the effect of it on limiting the hardware Trojan bypass rate.

5.2.2 Applying the information of LUT location to bitstream encryption

In both the countermeasures we introduced in Chapter 3 and 4, LUT location assignment is an important technique for achieving the effect of protection. In the future study, one possible defense of encrypting bitstream with physical location information can be tried to investigate. In this case, the bitstream encrypted by the location information key can

and only can be decrypted when running in the FPGA device which is constructed with the same location information. This “one-to-one" cryptography procedure can get rid of the transition process of the key and in this way, there is no need to worry about data leaking during the key propagation.

Appendix A

Source Codes for Defense Line 1 of FPGA-Oriented Moving Target Defense

A.1 Commands in the User Constraints File of Benchmark Circuit c432

```
INST "Mxor_N227_xo<0>1" BEL = C5LUT;
INST "Mxor_N227_xo<0>1" LOC = SLICE_X0Y61;
INST "Mxor_N239_xo<0>1" BEL = D5LUT;
INST "Mxor_N239_xo<0>1" LOC = SLICE_X0Y61;
INST "out3_SW0" BEL = B6LUT;
INST "out3_SW0" LOC = SLICE_X0Y61;
INST "out5_SW0" BEL = A6LUT;
INST "out5_SW0" LOC = SLICE_X0Y61;
INST "out181" BEL = D6LUT;
INST "out181" LOC = SLICE_X37Y61;
INST "out1811" BEL = C6LUT;
INST "out1811" LOC = SLICE_X37Y61;
INST "Mxor_N251_xo<0>1" BEL = A6LUT;
INST "Mxor_N251_xo<0>1" LOC = SLICE_X37Y61;
INST "N4211_SW0" BEL = B6LUT;
INST "N4211_SW0" LOC = SLICE_X37Y61;
INST "N3291" BEL = D6LUT;
INST "N3291" LOC = SLICE_X18Y61;
INST "out1814" BEL = C6LUT;
INST "out1814" LOC = SLICE_X18Y61;
INST "N432" BEL = A6LUT;
INST "N432" LOC = SLICE_X18Y61;
INST "Mxor_N247_xo<0>1" BEL = B5LUT;
INST "Mxor_N247_xo<0>1" LOC = SLICE_X18Y61;
```

A.2 Commands in the User Constraints File of Benchmark Circuit c6288

```
INST "n1143_SW0" BEL = B6LUT;
INST "n1143_SW0" LOC = SLICE_X0Y61;
INST "n1155_SW0" BEL = A6LUT;
INST "n1155_SW0" LOC = SLICE_X0Y61;
INST "n1267" BEL = C6LUT;
INST "n1267" LOC = SLICE_X0Y61;
INST "n1271" BEL = D5LUT;
INST "n1271" LOC = SLICE_X0Y61;
INST "n1145_SW0" BEL = A6LUT;
INST "n1145_SW0" LOC = SLICE_X37Y61;
INST "n1721" BEL = B6LUT;
INST "n1721" LOC = SLICE_X37Y61;
INST "n1273" BEL = C5LUT;
INST "n1273" LOC = SLICE_X37Y61;
INST "n1805" BEL = D5LUT;
INST "n1805" LOC = SLICE_X37Y61;
INST "n1139_SW0" BEL = D6LUT;
INST "n1139_SW0" LOC = SLICE_X18Y61;
INST "n1153_SW0" BEL = C6LUT;
INST "n1153_SW0" LOC = SLICE_X18Y61;
INST "n1265" BEL = B5LUT;
INST "n1265" LOC = SLICE_X18Y61;
INST "n1697" BEL = A6LUT;
INST "n1697" LOC = SLICE_X18Y61;
```

A.3 Commands in the User Constraints File of Benchmark Circuit s444

```
INST "G571" BEL = B5LUT;
INST "G571" LOC = SLICE_X0Y61;
INST "G801" BEL = C6LUT;
INST "G801" LOC = SLICE_X0Y61;
INST "G1111" BEL = D6LUT;
INST "G1111" LOC = SLICE_X0Y61;
INST "G371" BEL = C6LUT;
```

```
INST "G371" LOC = SLICE_X37Y61;
INST "G921" BEL = B6LUT;
INST "G921" LOC = SLICE_X37Y61;
INST "G451" BEL = A6LUT;
INST "G451" LOC = SLICE_X37Y61;
INST "G1101" BEL = C6LUT;
INST "G1101" LOC = SLICE_X18Y61;
INST "G411" BEL = D6LUT;
INST "G411" LOC = SLICE_X18Y61;
INST "G1011" BEL = A6LUT;
INST "G1011" LOC = SLICE_X18Y61;
```

A.4 Commands in the User Constraints File of Benchmark Circuit s13207

```
INST "g43741" BEL = D6LUT;
INST "g43741" LOC = SLICE_X0Y61;
INST "g55371" BEL = C6LUT;
INST "g55371" LOC = SLICE_X0Y61;
INST "g50961" BEL = A6LUT;
INST "g50961" LOC = SLICE_X0Y61;
INST "g56984_SW0" BEL = B6LUT;
INST "g56984_SW0" LOC = SLICE_X0Y61;
INST "g35151" BEL = A6LUT;
INST "g35151" LOC = SLICE_X37Y61;
INST "g35152" BEL = C6LUT;
INST "g35152" LOC = SLICE_X37Y61;
INST "g46761" BEL = B6LUT;
INST "g46761" LOC = SLICE_X37Y61;
INST "g48681" BEL = D6LUT;
INST "g48681" LOC = SLICE_X37Y61;
INST "g56721" BEL = A6LUT;
INST "g56721" LOC = SLICE_X18Y61;
INST "g61201" BEL = B6LUT;
INST "g61201" LOC = SLICE_X18Y61;
INST "g60031" BEL = C5LUT;
INST "g60031" LOC = SLICE_X18Y61;
INST "g64601" BEL = D5LUT;
INST "g64601" LOC = SLICE_X18Y61;
```

Appendix B

Source Codes for Defense Line 2 of FPGA-Oriented Moving Target Defense

B.1 Verilog Implementation on Benchmark Circuit c432

```
module c432_DMRrandom_gating(N1_1, N1_2, N4,N8,N11,N14,N17,N21,N24,N27,
N30,N34,N37,N40,N43,N47,N50,N53,N56,N60,N63,N66,N69,N73,N76,N79,N82,N86,
N89,N92,N95,N99,N102,N105,N108,N112,N115,N223,N329,N370,N421,N430,N431,N432);
input N1_1, N1_2, N4,N8,N11,N14,N17,N21,N24,N27,N30,N34,N37,N40,N43,N47,N50,
N53,N56,N60,N63,N66,N69,N73,N76,N79,N82,N86,N89,N92,N95,N99,N102,N105,N108,
N112,N115;
output N223,N329,N370,N421,N430,N431,N432;
wire sel;
c432_basic_gating copy1(sel & N1_1,sel & N4,sel & N8,sel & N11,sel & N14,sel & N17,sel
& N21,sel & N24,sel & N27,sel & N30,sel & N34,sel & N37,sel & N40,sel & N43,sel &
N47,sel & N50,sel & N53,sel & N56,sel & N60,sel & N63,sel & N66,sel & N69,sel & N73,sel
& N76,sel & N79,sel & N82,sel & N86,sel & N89,sel & N92,sel & N95, sel & N99,sel &
N102,sel & N105,sel & N108,sel & N112,sel & N115, N223_1, N329_1, N370_1, N421_1,
N430_1, N431_1, N432_1);
c432_basic_gating copy2(∼sel & N1_2,∼sel & N4,∼sel & N8,∼sel & N11,∼sel & N14,∼sel
& N17,∼sel & N21,∼sel & N24,∼sel & N27,∼sel & N30,∼sel & N34,∼sel & N37,∼sel
& N40,∼sel & N43,∼sel & N47,∼sel & N50,∼sel & N53,∼sel & N56,∼sel & N60,∼sel
& N63,∼sel & N66,∼sel & N69,∼sel & N73,∼sel & N76,∼sel & N79,∼sel & N82,∼sel
& N86,∼sel & N89,∼sel & N92,∼sel & N95,∼sel & N99,∼sel & N102,∼sel & N105,∼sel
& N108,∼sel & N112,∼sel & N115, N223_2, N329_2, N370_2, N421_2, N430_2, N431_2,
N432_2);
    assign sel = N8 ∧ N60;
    assign N223 = sel? N223_1 : N223_2;
    assign N329 = sel? N329_1 : N329_2;
    assign N370 = sel? N370_1 : N370_2;
```

```
    assign N421 = sel? N421_1 : N421_2;
    assign N430 = sel? N430_1 : N430_2;
    assign N431 = sel? N431_1 : N431_2;
    assign N432 = sel? N432_1 : N432_2;
endmodule
```

B.2 Verilog Implementation on Benchmark Circuit s444

```
module s444_DMRRandom_gating( input blif_clk_net,
    input blif_reset_net,
    input G0_1,
    input G0_2,
    input G1,
    input G2,
    output G118,
    output G167,
    output G107,
    output G119,
    output G168,
    output G108);
wire sel;
s444_bench_gating copy1( blif_clk_net, blif_reset_net, sel & G0_1, sel & G1, sel & G2,
G118_1, G167_1, G107_1, G119_1, G168_1, G108_1);
s444_bench_gating copy2( blif_clk_net, blif_reset_net, ∼sel & G0_2, ∼sel & G1, ∼sel &
G2, G118_2, G167_2, G107_2, G119_2, G168_2, G108_2);
reg SEL;
always @(posedge blif_clk_net)
begin
    if (blif_reset_net == 1'b1)
    SEL = 0;
    else
    SEL = G1 ∧ G2;
end
    assign sel = SEL;
    assign G118 = sel? G118_1 : G118_2;
    assign G167 = sel? G167_1 : G167_2;
    assign G107 = sel? G107_1 : G107_2;
    assign G119 = sel? G119_1 : G119_2;
    assign G168 = sel? G168_1 : G168_2;
```

```
    assign G108 = sel? G108_1 : G108_2;
endmodule
```

Appendix C

Source Codes for Defense Line 3 of FPGA-Oriented Moving Target Defense

C.1 Verilog Implementation on Benchmark Circuit c432

C.1.1 Top-Level Control Logic

```
module c432_AsseDMR_gating_4sec_all_input_gating(N1_1, N1_2, N4,N8,N11,N14,N17,
N21,N24,N27,N30,N34,N37,N40,N43,N47,N50,N53,N56,N60,N63,N66,N69,N73,N76,N79,
N82,N86,N89,N92,N95,N99,N102,N105,N108,N112,N115,N223,N329,N370,N421,N430,
N431,N432);
parameter NUMSUB = 4;
input N1_1, N1_2,N4,N8,N11,N14,N17,N21,N24,N27,N30,N34,N37,N40,N43,N47,N50,
N53,N56,N60,N63,N66,N69,N73,N76,N79,N82,N86,N89,N92,N95,N99,N102,N105,N108,
N112,N115;
output N223,N329,N370,N421,N430,N431,N432;
wire N223,N329,N370,N421,N430,N431,N432;
wire[NUMSUB-1:0] sel;
wire copysel;
wire[NUMSUB-1:0] NodeOut1, NodeOut2;
c432_submod_gating_4 copy1(copysel & N1_1,copysel & N4,copysel & N8,copysel & N11,
copysel & N14,copysel & N17,copysel & N21,copysel & N24,copysel & N27,copysel &
N30, copysel & N34,copysel & N37,copysel & N40,copysel & N43,copysel & N47,copysel
& N50,copysel & N53,copysel & N56,copysel & N60,copysel & N63, copysel & N66,copysel
& N69,copysel & N73,copysel & N76,copysel & N79,copysel & N82,copysel & N86,copysel
& N89,copysel & N92,copysel & N95, copysel & N99,copysel & N102,copysel & N105,
copysel & N108,copysel & N112,copysel & N115, N223_1, N329_1, N370_1, N421_1, N430_1,
N431_1, N432_1, sel, copysel, NodeOut2, NodeOut1);
c432_submod_gating_4 copy2(∼copysel & N1_2,∼copysel & N4,∼copysel & N8,∼copysel
& N11,∼copysel & N14,∼copysel & N17,∼copysel & N21,∼copysel & N24,∼copysel &
```

```
N27,∼copysel & N30, ∼copysel & N34,∼copysel & N37,∼copysel & N40,∼copysel &
N43,∼copysel & N47,∼copysel & N50,∼copysel & N53,∼copysel & N56,∼copysel & N60,
∼copysel & N63, ∼copysel & N66,∼copysel & N69,∼copysel & N73,∼copysel & N76,
∼copysel & N79,∼copysel & N82,∼copysel & N86,∼copysel & N89,∼copysel & N92,
∼copysel & N95, ∼copysel & N99,∼copysel & N102,∼copysel & N105,∼copysel & N108,
∼copysel & N112,∼copysel & N115, N223_2, N329_2, N370_2, N421_2, N430_2, N431_2,
N432_2, sel, ∼copysel, NodeOut1, NodeOut2);
    assign sel[0]= N34 & N37;
    assign sel[1]= N63 ∧ N66;
    assign sel[2]= N56 ∧ N60;
    assign sel[3]= N73 ∼∧ N76;
    assign copysel = N8 ∧ N60;
wire [6:0] copy1out, copy2out;
    assign copy1out = {N223_1, N329_1, N370_1, N421_1, N430_1, N431_1, N432_1};
    assign copy2out = {N223_2, N329_2, N370_2, N421_2, N430_2, N431_2, N432_2};
    assign {N223,N329,N370,N421,N430,N431,N432} = copysel? copy1out : copy2out;
endmodule
```

C.1.2 Modified Instance

```
‘define CASE1
//‘define CASE2
//‘define CASE3
//‘define CASE4
//‘define CASE5
module c432_submod_gating_4 (N1,N4,N8,N11,N14,N17,N21,N24,N27,N30, N34,N37,
N40,N43,N47,N50,N53,N56,N60,N63, N66,N69,N73,N76,N79,N82,N86,N89,N92,N95,
N99, N102,N105,N108,N112,N115,N223,N329,N370,N421, N430,N431,N432, sel, copysel,
NodeIn, NodeOut);
input N1,N4,N8,N11,N14,N17,N21,N24,N27,N30, N34,N37,N40,N43,N47,N50,N53,N56,
N60, N63, N66,N69,N73,N76,N79,N82,N86,N89,N92,N95, N99,N102,N105,N108,N112,
N115;
input copysel;
output N223,N329,N370,N421,N430,N431,N432;
wire N118,N119,N122,N123,N126,N127,N130,N131,N134,N135, N138,N139,N142,N143,
N146,N147,N150,N151,N154,N157, N158,N159,N162,N165,N168,N171,N174,N177,N180,
N183, N184,N185,N186,N187,N188,N189,N190,N191,N192,N193, N194,N195,N196,N197,
N198,N199,N203,N213,N224,N227, N230,N233,N236,N239,N242,N243,N246,N247,N250,
N251, N254,N255,N256,N257,N258,N259,N260,N263,N264,N267, N270,N273,N276,N279,
N282,N285,N288,N289,N290,N291, N292,N293,N294,N295,N296,N300,N301,N302,N303,
```

```
N304, N305,N306,N307,N308,N309,N319,N330,N331,N332,N333, N334,N335,N336,N337,
N338,N339,N340,N341,N342,N343, N344,N345,N346,N347,N348,N349,N350,N351,N352,
N353, N354,N355,N356,N357,N360,N371,N372,N373,N374,N375, N376,N377,N378,N379,
N380,N381,N386,N393,N399,N404, N407,N411,N414,N415,N416,N417,N418,N419,N420,
N422, N425,N428,N429;
parameter NUMSUB = 4;
input [NUMSUB-1:0] sel;
input [NUMSUB-1:0] NodeIn;
output [NUMSUB-1:0] NodeOut;
wire [NUMSUB-1:0] sel;
wire [NUMSUB-1:0] NodeIn;
wire [NUMSUB-1:0] NodeOut;
wire [NUMSUB-1:0] SelectedNode;
    ‘ifdef CASE1
    assign NodeOut = {N425_current, N344_current, N194_current, N118_current};
    ‘endif
    ‘ifdef CASE2
    assign NodeOut ={N428_current, N345_current, N195_current, N119_current};
    ‘endif
    ‘ifdef CASE3
    assign NodeOut = {N429_current, N346_current, N196_current, N122_current};
    ‘endif
    ‘ifdef CASE4
    assign NodeOut = {N430_current, N347_current, N197_current, N123_current};
    ‘endif
    ‘ifdef CASE5
    assign NodeOut = {N431_current, N348_current, N198_current, N126_current};
    ‘endif
    assign SelectedNode[0] = copysel & (sel[0]? NodeOut[0] : NodeIn[0]);
    assign SelectedNode[1] = copysel & (sel[1]? NodeOut[1] : NodeIn[1]);
    assign SelectedNode[2] = copysel & (sel[2]? NodeOut[2] : NodeIn[2]);
    assign SelectedNode[3] = copysel & (sel[3]? NodeOut[3] : NodeIn[3]);
    ‘ifdef CASE1
    assign N118 = SelectedNode[0];
    assign N194 = SelectedNode[1];
    assign N344 = SelectedNode[2];
    assign N425 = SelectedNode[3];
    ‘endif
    ‘ifdef CASE2
    assign N119 = SelectedNode[0];
```

```
assign N195 = SelectedNode[1];
assign N345 = SelectedNode[2];
assign N428 = SelectedNode[3];
‘endif
‘ifdef CASE3
assign N122 = SelectedNode[0];
assign N196 = SelectedNode[1];
assign N346 = SelectedNode[2];
assign N429 = SelectedNode[3];
‘endif
‘ifdef CASE4
assign N123 = SelectedNode[0];
assign N197 = SelectedNode[1];
assign N347 = SelectedNode[2];
assign N430 = SelectedNode[3];
‘endif
‘ifdef CASE5
assign N126 = SelectedNode[0];
assign N198 = SelectedNode[1];
assign N348 = SelectedNode[2];
assign N431 = SelectedNode[3];
‘endif
‘ifdef CASE1
not NOT1_1 (N118_current, N1);
‘else
not NOT1_1 (N118, N1);
‘endif
‘ifdef CASE2
not NOT1_2 (N119_current, N4);
‘else
not NOT1_2 (N119, N4);
‘endif
‘ifdef CASE3
not NOT1_3 (N122_current, N11);
‘else
not NOT1_3 (N122, N11);
‘endif
‘ifdef CASE4
not NOT1_4 (N123_current, N17);
‘else
```

```
not NOT1_4 (N123, N17);
‘endif
‘ifdef CASE5
not NOT1_5 (N126_current, N24);
‘else
not NOT1_5 (N126, N24);
‘endif
not NOT1_6 (N127, N30);
not NOT1_7 (N130, N37);
not NOT1_8 (N131, N43);
not NOT1_9 (N134, N50);
not NOT1_10 (N135, N56);
not NOT1_11 (N138, N63);
not NOT1_12 (N139, N69);
not NOT1_13 (N142, N76);
not NOT1_14 (N143, N82);
not NOT1_15 (N146, N89);
not NOT1_16 (N147, N95);
not NOT1_17 (N150, N102);
not NOT1_18 (N151, N108);
nand NAND2_19 (N154, N118, N4);
nor NOR2_20 (N157, N8, N119);
nor NOR2_21 (N158, N14, N119);
nand NAND2_22 (N159, N122, N17);
nand NAND2_23 (N162, N126, N30);
nand NAND2_24 (N165, N130, N43);
nand NAND2_25 (N168, N134, N56);
nand NAND2_26 (N171, N138, N69);
nand NAND2_27 (N174, N142, N82);
nand NAND2_28 (N177, N146, N95);
nand NAND2_29 (N180, N150, N108);
nor NOR2_30 (N183, N21, N123);
nor NOR2_31 (N184, N27, N123);
nor NOR2_32 (N185, N34, N127);
nor NOR2_33 (N186, N40, N127);
nor NOR2_34 (N187, N47, N131);
nor NOR2_35 (N188, N53, N131);
nor NOR2_36 (N189, N60, N135);
nor NOR2_37 (N190, N66, N135);
nor NOR2_38 (N191, N73, N139);
```

```
nor NOR2_39 (N192, N79, N139);
nor NOR2_40 (N193, N86, N143);
‘ifdef CASE1
nor NOR2_41 (N194_current, N92, N143);
‘else
nor NOR2_41 (N194, N92, N143);
‘endif
‘ifdef CASE2
nor NOR2_42 (N195_current, N99, N147);
‘else
nor NOR2_42 (N195, N99, N147);
‘endif
‘ifdef CASE3
nor NOR2_43 (N196_current, N105, N147);
‘else
nor NOR2_43 (N196, N105, N147);
‘endif
‘ifdef CASE4
nor NOR2_44 (N197_current, N112, N151);
‘else
nor NOR2_44 (N197, N112, N151);
‘endif
‘ifdef CASE5
nor NOR2_45 (N198_current, N115, N151);
‘else
nor NOR2_45 (N198, N115, N151);
‘endif
and AND9_46 (N199, N154, N159, N162, N165, N168, N171, N174, N177, N180);
not NOT1_47 (N203, N199);
not NOT1_48 (N213, N199);
not NOT1_49 (N223, N199);
xor XOR2_50 (N224, N203, N154);
xor XOR2_51 (N227, N203, N159);
xor XOR2_52 (N230, N203, N162);
xor XOR2_53 (N233, N203, N165);
xor XOR2_54 (N236, N203, N168);
xor XOR2_55 (N239, N203, N171);
nand NAND2_56 (N242, N1, N213);
xor XOR2_57 (N243, N203, N174);
nand NAND2_58 (N246, N213, N11);
```

```
xor XOR2_59 (N247, N203, N177);
nand NAND2_60 (N250, N213, N24);
xor XOR2_61 (N251, N203, N180);
nand NAND2_62 (N254, N213, N37);
nand NAND2_63 (N255, N213, N50);
nand NAND2_64 (N256, N213, N63);
nand NAND2_65 (N257, N213, N76);
nand NAND2_66 (N258, N213, N89);
nand NAND2_67 (N259, N213, N102);
nand NAND2_68 (N260, N224, N157);
nand NAND2_69 (N263, N224, N158);
nand NAND2_70 (N264, N227, N183);
nand NAND2_71 (N267, N230, N185);
nand NAND2_72 (N270, N233, N187);
nand NAND2_73 (N273, N236, N189);
nand NAND2_74 (N276, N239, N191);
nand NAND2_75 (N279, N243, N193);
nand NAND2_76 (N282, N247, N195);
nand NAND2_77 (N285, N251, N197);
nand NAND2_78 (N288, N227, N184);
nand NAND2_79 (N289, N230, N186);
nand NAND2_80 (N290, N233, N188);
nand NAND2_81 (N291, N236, N190);
nand NAND2_82 (N292, N239, N192);
nand NAND2_83 (N293, N243, N194);
nand NAND2_84 (N294, N247, N196);
nand NAND2_85 (N295, N251, N198);
and AND9_86 (N296, N260, N264, N267, N270, N273, N276, N279, N282, N285);
not NOT1_87 (N300, N263);
not NOT1_88 (N301, N288);
not NOT1_89 (N302, N289);
not NOT1_90 (N303, N290);
not NOT1_91 (N304, N291);
not NOT1_92 (N305, N292);
not NOT1_93 (N306, N293);
not NOT1_94 (N307, N294);
not NOT1_95 (N308, N295);
not NOT1_96 (N309, N296);
not NOT1_97 (N319, N296);
not NOT1_98 (N329, N296);
```

```
xor XOR2_99 (N330, N309, N260);
xor XOR2_100 (N331, N309, N264);
xor XOR2_101 (N332, N309, N267);
xor XOR2_102 (N333, N309, N270);
nand NAND2_103 (N334, N8, N319);
xor XOR2_104 (N335, N309, N273);
nand NAND2_105 (N336, N319, N21);
xor XOR2_106 (N337, N309, N276);
nand NAND2_107 (N338, N319, N34);
xor XOR2_108 (N339, N309, N279);
nand NAND2_109 (N340, N319, N47);
xor XOR2_110 (N341, N309, N282);
nand NAND2_111 (N342, N319, N60);
xor XOR2_112 (N343, N309, N285);
‘ifdef CASE1
nand NAND2_113 (N344_current, N319, N73);
‘else
nand NAND2_113 (N344, N319, N73);
‘endif
‘ifdef CASE2
nand NAND2_114 (N345_current, N319, N86);
‘else
nand NAND2_114 (N345, N319, N86);
‘endif
‘ifdef CASE3
nand NAND2_115 (N346_current, N319, N99);
‘else
nand NAND2_115 (N346, N319, N99);
‘endif
‘ifdef CASE4
nand NAND2_116 (N347_current, N319, N112);
‘else
nand NAND2_116 (N347, N319, N112);
‘endif
‘ifdef CASE5
nand NAND2_117 (N348_current, N330, N300);
‘else
nand NAND2_117 (N348, N330, N300);
‘endif
nand NAND2_118 (N349, N331, N301);
```

```
nand NAND2_119 (N350, N332, N302);
nand NAND2_120 (N351, N333, N303);
nand NAND2_121 (N352, N335, N304);
nand NAND2_122 (N353, N337, N305);
nand NAND2_123 (N354, N339, N306);
nand NAND2_124 (N355, N341, N307);
nand NAND2_125 (N356, N343, N308);
and AND9_126 (N357, N348, N349, N350, N351, N352, N353, N354, N355, N356);
not NOT1_127 (N360, N357);
not NOT1_128 (N370, N357);
nand NAND2_129 (N371, N14, N360);
nand NAND2_130 (N372, N360, N27);
nand NAND2_131 (N373, N360, N40);
nand NAND2_132 (N374, N360, N53);
nand NAND2_133 (N375, N360, N66);
nand NAND2_134 (N376, N360, N79);
nand NAND2_135 (N377, N360, N92);
nand NAND2_136 (N378, N360, N105);
nand NAND2_137 (N379, N360, N115);
nand NAND4_138 (N380, N4, N242, N334, N371);
nand NAND4_139 (N381, N246, N336, N372, N17);
nand NAND4_140 (N386, N250, N338, N373, N30);
nand NAND4_141 (N393, N254, N340, N374, N43);
nand NAND4_142 (N399, N255, N342, N375, N56);
nand NAND4_143 (N404, N256, N344, N376, N69);
nand NAND4_144 (N407, N257, N345, N377, N82);
nand NAND4_145 (N411, N258, N346, N378, N95);
nand NAND4_146 (N414, N259, N347, N379, N108);
not NOT1_147 (N415, N380);
and AND8_148 (N416, N381, N386, N393, N399, N404, N407, N411, N414);
not NOT1_149 (N417, N393);
not NOT1_150 (N418, N404);
not NOT1_151 (N419, N407);
not NOT1_152 (N420, N411);
nor NOR2_153 (N421, N415, N416);
nand NAND2_154 (N422, N386, N417);
‘ifdef CASE1
nand NAND4_155 (N425_current, N386, N393, N418, N399);
‘else
nand NAND4_155 (N425, N386, N393, N418, N399);
```

```
    ‘endif
    ‘ifdef CASE2
    nand NAND3_156 (N428_current, N399, N393, N419);
    ‘else
    nand NAND3_156 (N428, N399, N393, N419);
    ‘endif
    ‘ifdef CASE3
    nand NAND4_157 (N429_current, N386, N393, N407, N420);
    ‘else
    nand NAND4_157 (N429, N386, N393, N407, N420);
    ‘endif
    ‘ifdef CASE4
    nand NAND4_158 (N430_current, N381, N386, N422, N399);
    ‘else
    nand NAND4_158 (N430, N381, N386, N422, N399);
    ‘endif
    ‘ifdef CASE5
    nand NAND4_159 (N431_current, N381, N386, N425, N428);
    ‘else
    nand NAND4_159 (N431, N381, N386, N425, N428);
    ‘endif
    nand NAND4_160 (N432, N381, N422, N425, N429);
endmodule
```

C.2 Verilog Implementation on Benchmark Circuit S444

C.2.1 Top-Level Control Logic

```
module s444_AsseDMR_gating_4sec_all_input_gating( input blif_clk_net,
    input blif_reset_net,
    input G0_1,
    input G0_2,
    input G1,
    input G2,
    output G118,
    output G167,
    output G107,
    output G119,
    output G168,
```

```
    output G108);
wire [3:0] NodeIn, NodeOut, sel;
wire copysel;
reg [3:0] SEL;
reg COPYSEL;
s444_submod_gating_4 copy1( blif_clk_net, blif_reset_net, ∼copysel & G0_1, ∼copysel
& G1, ∼copysel & G2, G118_1, G167_1, G107_1, G119_1, G168_1, G108_1, ∼copysel,
NodeIn, NodeOut, sel);
s444_submod_gating_4 copy2( blif_clk_net, blif_reset_net, copysel & G0_2, copysel & G1,
copysel & G2, G118_2, G167_2, G107_2, G119_2, G168_2, G108_2, copysel, NodeOut,
NodeIn, sel);
always @(posedge blif_clk_net)
begin
    if (blif_reset_net == 1'b1)
    begin
    SEL[0] = 0;
    SEL[1] = 0;
    SEL[2] = 0;
    SEL[3] = 0;
    COPYSEL = 0;
    end
    else
    begin
    SEL[0] = G1;
    SEL[1] = G2;
    SEL[2] = G1 | G2;
    SEL[3] = G1 ∧ G2;
    COPYSEL = G1 ∧ G2;
    end
end
    assign sel = SEL;
    assign copysel = COPYSEL;
    assign G118 = copysel? G118_2 : G118_1;
    assign G167 = copysel? G167_2 : G167_1;
    assign G107 = copysel? G107_2 : G107_1;
    assign G119 = copysel? G119_2 : G119_1;
    assign G168 = copysel? G168_2 : G168_1;
    assign G108 = copysel? G108_2 : G108_1;
endmodule
```

C.2.2 Modified Instance

```
module s444_submod_gating_4( blif_clk_net, blif_reset_net, G0, G1, G2, G118, G167, G107,
G119, G168, G108, copysel, NodeIn, NodeOut, sel);
input blif_clk_net;
input blif_reset_net;
input G0;
input G1;
input G2;
input copysel;
output G118;
output G167;
output G107;
output G119;
output G168;
output G108;
input [3:0] NodeIn;
output [3:0] NodeOut;
input [3:0] sel;
reg G11;
reg G12;
reg G13;
reg G14;
reg G15;
reg G16;
reg G17;
reg G18;
reg G19;
reg G20;
reg G21;
reg G22;
reg G23;
reg G24;
reg G25;
reg G26;
reg G27;
reg G28;
reg G29;
reg G30;
reg G31;
```

```
wire IIII212;
wire G50;
wire G95;
wire IIII211;
wire G120;
wire IIII192;
wire IIII272;
wire G79;
wire G138;
wire G61;
wire G140;
wire G161;
wire IIII201;
wire G77;
wire IIII190;
wire IIII283;
wire G93;
wire G137;
wire G105;
wire G165;
wire G144;
wire G129;
wire IIII226;
wire IIII281;
wire IIII372;
wire G134;
wire IIII235;
wire G147;
wire G146;
wire G41;
wire G83;
wire G160;
wire G35;
wire G122;
wire G80;
wire G117;
wire G132;
wire G151;
wire G158;
wire G65;
```

```
wire G81;
wire G113;
wire G73;
wire G58;
wire IIII181;
wire G163;
wire G148;
wire G43;
wire IIII271;
wire IIII392;
wire G153;
wire G114;
wire G85;
wire G74;
wire G115;
wire G103;
wire IIII202;
wire G51;
wire IIII236;
wire G49;
wire IIII246;
wire G157;
wire G130;
wire G54;
wire G67;
wire G42;
wire IIII182;
wire G87;
wire G71;
wire G69;
wire G90;
wire IIII255;
wire G126;
wire IIII321;
wire G75;
wire G47;
wire G111;
wire G142;
wire IIII304;
wire G40;
```

```
wire IIII180;
wire IIII318;
wire G101;
wire G121;
wire G57;
wire IIII225;
wire G166;
wire G159;
wire G149;
wire IIII256;
wire G33;
wire IIII200;
wire G139;
wire G109;
wire IIII257;
wire G162;
wire G164;
wire G68;
wire IIII293;
wire IIII291;
wire G145;
wire G37;
wire G66;
wire G136;
wire IIII191;
wire G125;
wire G141;
wire G123;
wire IIII273;
wire IIII105;
wire IIII292;
wire G155;
wire G56;
wire G44;
wire G128;
wire G46;
wire G94;
wire G91;
wire G59;
wire G97;
```

```
wire IIII210;
wire G62;
wire G102;
wire G104;
wire G150;
wire IIII227;
wire G99;
wire G152;
wire G135;
wire G116;
wire G112;
wire G60;
wire G82;
wire G64;
wire IIII237;
wire G84;
wire G86;
wire IIII382;
wire G92;
wire G34;
wire IIII245;
wire IIII336;
wire G162BF;
wire G45;
wire G89;
wire IIII324;
wire G76;
wire G72;
wire G98;
wire G63;
wire G55;
wire G133;
wire G78;
wire G96;
wire G38;
wire G127;
wire IIII247;
wire G48;
wire G53;
wire G36;
```

```
wire G156;
wire G110;
wire G88;
wire G32;
wire IIII302;
wire G131;
wire IIII282;
wire G143;
wire G100;
wire G70;
wire IIII303;
wire G154;
wire G52;
wire G124;
wire G106;
    assign NodeOut = G104_current, IIII304_current, G73_current, G140_current;
    assign G140 = copysel & (sel[0]? NodeIn[0] : NodeOut[0]);
    assign G73 = copysel & (sel[1]? NodeIn[1] : NodeOut[1]);
    assign IIII304 = copysel & (sel[2]? NodeIn[2] : NodeOut[2]);
    assign G104 = copysel & (sel[3]? NodeIn[3] : NodeOut[3]);
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G11 <= 0;
    else
    G11 <= G37;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G12 <= 0;
    else
    G12 <= G41;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G13 <= 0;
    else
    G13 <= G45;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G14 <= 0;
    else
    G14 <= G49;
```

```
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G15 <= 0;
    else
    G15 <= G58;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G16 <= 0;
    else
    G16 <= G62;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G17 <= 0;
    else
    G17 <= G66;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G18 <= 0;
    else
    G18 <= G70;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G19 <= 0;
    else
    G19 <= G80;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G20 <= 0;
    else
    G20 <= G84;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G21 <= 0;
    else
    G21 <= G88;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G22 <= 0;
    else
    G22 <= G92;
```

```
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G23 <= 0;
    else
    G23 <= G101;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G24 <= 0;
    else
    G24 <= G162BF;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G25 <= 0;
    else
    G25 <= G109;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G26 <= 0;
    else
    G26 <= G110;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G27 <= 0;
    else
    G27 <= G111;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G28 <= 0;
    else
    G28 <= G112;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G29 <= 0;
    else
    G29 <= G113;
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G30 <= 0;
    else
    G30 <= G114;
```

```
always @(posedge blif_clk_net or posedge blif_reset_net)
    if(blif_reset_net == 1)
    G31 <= 0;
    else
    G31 <= G155;
    assign IIII212 = ((~G51)) | ((~IIII210));
    assign G50 = ((~G52));
    assign G95 = ((~G76)&(~G77)&(~G78)&(~G79));
    assign IIII211 = ((~G14)) | ((~IIII210));
    assign IIII192 = ((~G43)) | ((~IIII190));
    assign G120 = ((~G150)) | ((~G128));
    assign G167 = ((~G29));
    assign IIII272 = ((~G19)) | ((~IIII271));
    assign G79 = ((~G97));
    assign G138 = (G136) | (G142);
    assign G61 = ((~IIII226)) | ((~IIII227));
    //assign G140 = (G24) | (G21) | (G20) | (G150);
    assign G140_current = (G24) | (G21) | (G20) | (G150);
    assign G161 = ((~G17));
    assign IIII201 = ((~G13)) | ((~IIII200));
    assign G77 = ((~G20));
    assign G93 = ((~G74)&(~G79)&(~G75));
    assign IIII283 = ((~G86)) | ((~IIII281));
    assign IIII190 = ((~G12)) | ((~G43));
    assign G137 = (G136) | (G20) | (G19);
    assign G105 = (G102) | (G103);
    assign G165 = ((~G148)) | ((~G149));
    assign G144 = ((~G21));
    assign G129 = ((~G19)) | ((~G135));
    assign IIII226 = ((~G15)) | ((~IIII225));
    assign IIII281 = ((~G20)) | ((~G86));
    assign IIII372 = ((~G0));
    assign G134 = (G152) | (G142) | (G21);
    assign IIII235 = ((~G16)) | ((~G64));
    assign G147 = (G152) | (G144);
    assign G146 = (G152) | (G143);
    assign G41 = ((~G98)&(~G42)&(~G152));
    assign G83 = ((~IIII272)) | ((~IIII273));
    assign G160 = ((~IIII382));
    assign G35 = ((~G12));
```

```
assign G122 = (G24&G121);
assign G80 = ((~G93)&(~G81)&(~G152));
assign G117 = ((~G145)) | ((~G146)) | ((~G147));
assign G151 = ((~G20)) | ((~G144)) | ((~G143)) | ((~G139));
assign G132 = ((~G133)) | ((~G134));
assign G158 = (G31) | (G160);
assign G65 = ((~IIII236)) | ((~IIII237));
assign G81 = ((~G83));
assign G113 = ((~G163)&(~G164));
//assign G73 = ((~IIII256)) | ((~IIII257));
assign G73_current = ((~IIII256)) | ((~IIII257));
assign G58 = ((~G97)&(~G59)&(~G152));
assign IIII181 = ((~G11)) | ((~IIII180));
assign G163 = (G161&G165&G162);
assign G148 = ((~G150)) | ((~G135)) | ((~G132));
assign G43 = ((~G34));
assign IIII271 = ((~G19)) | ((~G82));
assign IIII392 = ((~G30));
assign G114 = ((~G150)&(~G151));
assign G153 = ((~G152));
assign G85 = ((~G87));
assign G115 = (G161&G117&G162);
assign G74 = ((~G22));
assign G103 = ((~G106));
assign IIII202 = ((~G47)) | ((~IIII200));
assign G51 = ((~G34)&(~G35)&(~G36));
assign IIII236 = ((~G16)) | ((~IIII235));
assign G49 = ((~G98)&(~G50)&(~G152));
assign IIII246 = ((~G17)) | ((~IIII245));
assign G118 = ((~IIII336));
assign G157 = ((~G160));
assign G130 = ((~G143)&(~G152));
assign G54 = ((~G15)&(~G16)&(~G17));
assign G67 = ((~G69));
assign G42 = ((~G44));
assign G107 = ((~IIII321));
assign IIII182 = ((~IIII180));
assign G87 = ((~IIII282)) | ((~IIII283));
assign G71 = ((~G73));
assign G69 = ((~IIII246)) | ((~IIII247));
```

```
assign G90 = ((~G76)&(~G77)&(~G79));
assign IIII255 = ((~G18)) | ((~G72));
assign G126 = (G139&G21);
assign IIII321 = ((~G25));
assign G75 = ((~G19)&(~G20)&(~G21));
assign G47 = ((~G34)&(~G35));
assign G111 = ((~G140)) | ((~G141)) | ((~G139));
assign G142 = ((~G22));
//assign IIII304 = ((~G95)) | ((~IIII302));
assign IIII304_current = ((~G95)) | ((~IIII302));
assign G40 = ((~IIII181)) | ((~IIII182));
assign IIII180 = ((~G11));
assign IIII318 = ((~G2));
assign G101 = (G100&G99);
assign G57 = ((~G31)&(~G98));
assign G121 = ((~G19)) | ((~G135)) | ((~G142)) | ((~G136));
assign IIII225 = ((~G15)) | ((~G60));
assign G166 = ((~G162));
assign G149 = ((~G131)) | ((~G130));
assign G159 = (G156) | (G157);
assign IIII256 = ((~G18)) | ((~IIII255));
assign G33 = ((~G11)&(~G12)&(~G13));
assign IIII200 = ((~G13)) | ((~G47));
assign G139 = ((~G152));
assign G109 = ((~G122)&(~G123));
assign IIII257 = ((~G72)) | ((~IIII255));
assign G162 = ((~G120)) | ((~G149));
assign G164 = (G165&G166);
assign IIII293 = ((~G90)) | ((~IIII291));
assign G68 = ((~G55)&(~G56)&(~G57));
assign IIII291 = ((~G21)) | ((~G90));
assign G37 = ((~G98)&(~G38)&(~G152));
assign G145 = (G152) | (G142) | (G20) | (G19);
assign G66 = ((~G97)&(~G67)&(~G152));
assign G136 = ((~G23));
assign IIII191 = ((~G12)) | ((~IIII190));
assign G125 = (G139&G20&G19);
assign G141 = (G24) | (G22) | (G21);
assign G123 = ((~G137)) | ((~G138)) | ((~G21)) | ((~G139));
assign G168 = ((~IIII392));
```

```
assign IIII273 = ((~G82)) | ((~IIII271));
assign IIII105 = ((~G162));
assign IIII292 = ((~G21)) | ((~IIII291));
assign G155 = (G154&G153);
assign G56 = ((~G16));
assign G44 = ((~IIII191)) | ((~IIII192));
assign G128 = ((~G20)&(~G144)&(~G136)&(~G152));
assign G46 = ((~G48));
assign G94 = ((~G96));
assign G91 = ((~IIII292)) | ((~IIII293));
assign G59 = ((~G61));
assign G97 = ((~G53)&(~G57)&(~G54));
assign IIII210 = ((~G14)) | ((~G51));
assign G62 = ((~G97)&(~G63)&(~G152));
assign G102 = ((~G23));
assign G150 = ((~G19));
//assign G104 = (G23) | (G106);
assign G104_current = (G23) | (G106);
assign IIII227 = ((~G60)) | ((~IIII225));
assign G99 = ((~G152));
assign G152 = ((~IIII372));
assign G135 = ((~G20));
assign G116 = (G117&G166);
assign G112 = ((~G115)&(~G116));
assign G60 = ((~G57));
assign G82 = ((~G79));
assign G64 = ((~G55)&(~G57));
assign IIII237 = ((~G64)) | ((~IIII235));
assign G84 = ((~G93)&(~G85)&(~G152));
assign G86 = ((~G76)&(~G79));
assign IIII382 = ((~G1));
assign G92 = ((~G93)&(~G94)&(~G152));
assign G34 = ((~G11));
assign IIII245 = ((~G17)) | ((~G68));
assign IIII336 = ((~G27));
assign G162BF = ((~IIII105));
assign G45 = ((~G98)&(~G46)&(~G152));
assign G89 = ((~G91));
assign IIII324 = ((~G26));
assign G76 = ((~G19));
```

```
    assign G98 = ((~G32)&(~G33));
    assign G72 = ((~G55)&(~G56)&(~G161)&(~G57));
    assign G63 = ((~G65));
    assign G55 = ((~G15));
    assign G133 = (G152) | (G136) | (G22) | (G144);
    assign G78 = ((~G21));
    assign G96 = ((~IIII303)) | ((~IIII304));
    assign G38 = ((~G40));
    assign G127 = (G139&G24);
    assign IIII247 = ((~G68)) | ((~IIII245));
    assign G53 = ((~G18));
    assign G48 = ((~IIII201)) | ((~IIII202));
    assign G36 = ((~G13));
    assign G156 = ((~G31));
    assign G108 = ((~IIII324));
    assign G110 = ((~G124)&(~G125)&(~G126)&(~G127));
    assign G88 = ((~G93)&(~G89)&(~G152));
    assign G32 = ((~G14));
    assign IIII302 = ((~G22)) | ((~G95));
    assign G119 = ((~G28));
    assign G131 = (G144) | (G22) | (G23) | (G129);
    assign IIII282 = ((~G20)) | ((~IIII281));
    assign G143 = ((~G24));
    assign G100 = (G104&G105);
    assign G70 = ((~G97)&(~G71)&(~G152));
    assign IIII303 = ((~G22)) | ((~IIII302));
    assign G154 = (G158&G159);
    assign G52 = ((~IIII211)) | ((~IIII212));
    assign G124 = (G139&G22&G150);
    assign G106 = ((~IIII318));
endmodule
```

References

[1] *Fpga market by technology (sram, antifuse, flash), node size (less than 28 nm, 28-90 nm, more than 90 nm), configuration (high-end fpga, mid-range fpga, low-end fpga), vertical (telecommunications, automotive), and geography - global forecast to 2023*. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/fpga-market-194123367.html.

[2] *Fpgas accelerate time to market for industrial designs*. [Online]. Available: https://www.eetimes.com/document.asp?doc_id=1150608.

[3] *Soc fpga hardware security requirements and roadmap*. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/education/events/northamerica/isdf/SoC-FPGA-Hardware-Security.pdf.

[4] R. S. Chakraborty, I. Saha, A. Palchaudhuri, and G. K. Naik, "Hardware Trojan Insertion by Direct Modification of FPGA Configuration Bitstream", *IEEE Design Test*, vol. 30, no. 2, pp. 45–54, 2013, ISSN: 2168-2356. DOI: 10.1109/MDT.2013.2247460.

[5] Z. Zhang, L. Njilla, C. Kamhoua, K. Kwiat, and Q. Yu, *Securing FPGA-based Obsolete Component Replacement for Legacy Systems*, to appear in Proc. ISQED'18.

[6] *Basic fpga architecture and its applications*. [Online]. Available: https://www.edgefx.in/fpga-architecture-applications/.

[7] *ISE In Depth Tutorial*.

[8] *Altera design flow for xilinx users*. [Online]. Available: http://home.engineering.iastate.edu/~zambreno/classes\\/cpre583/documents/altera/an307.pdf.

[9] V. Mirian and P. Chow, "Extracting designs of secure ips using fpga cad tools", in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, 2016, pp. 293–298. DOI: 10.1145/2902961.2903033.

[10] A. L. Oliveira, "Techniques for the creation of digital watermarks in sequential circuit designs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1101–1117, 2001, ISSN: 0278-0070. DOI: 10.1109/43.945306.

[11] J. Zhang, Y. Lin, Y. Lyu, and G. Qu, "A PUF-FSM Binding Scheme for FPGA IP Protection and Pay-Per-Device Licensing", *IEEE Trans. on Information Forensics and Security*, vol. 10, no. 6, pp. 1137–1150, 2015, ISSN: 1556-6013. DOI: 10.1109/TIFS.2015.2400413.

[12] R. Karam, T. Hoque, S. Ray, M. Tehranipoor, and S. Bhunia, "MUTARCH: Architectural diversity for FPGA device and IP security", in *Proc. 22nd Asia and South Pacific Design Automation Conf.*, 2017, pp. 611–616. DOI: 10.1109/ASPDAC.2017.7858391.

[13] S. Mal-Sarkar, R. Karam, S. Narasimhan, A. Ghosh, A. Krishna, and S. Bhunia, "Design and Validation for FPGA Trust under Hardware Trojan Attacks", *IEEE Transactions on Multi-Scale Computing Systems*, vol. 2, no. 3, pp. 186–198, 2016. DOI: 10.1109/TMSCS.2016.2584052.

[14] G. Bloom, B. Narahari, R. Simha, A. Namazi, and R. Levy, "Fpga soc architecture and runtime to prevent hardware trojans from leaking secrets", in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2015, pp. 48–51. DOI: 10.1109/HST.2015.7140235.

[15] B. Khaleghi, A. Ahari, H. Asadi, and S. Bayat-Sarmadi, "Fpga-based protection scheme against hardware trojan horse insertion using dummy logic", *IEEE Embedded Systems Letters*, vol. 7, no. 2, pp. 46–50, 2015, ISSN: 1943-0663. DOI: 10.1109/LES.2015.2406791.

[16] D. M. Shila and V. Venugopal, "Design, implementation and security analysis of Hardware Trojan Threats in FPGA", in *Proc. 2014 IEEE International Conference on Communications (ICC)*, 2014, pp. 719–724. DOI: 10.1109/ICC.2014.6883404.

[17] M. Majzoobi, F. Koushanfar, and M. Potkonjak, "FPGA-oriented Security", in *Springer, New York,*, 2011.

[18] R. Lumbiarres-Lopez, M. Lopez-Garcia, and E. Canto-Navarro, "Hardware architecture implemented on fpga for protecting cryptographic keys against side-channel attacks", *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, pp. 1–1, 2017, ISSN: 1545-5971. DOI: 10.1109/TDSC.2016.2610966.

[19] N. Kamoun, L. Bossuet, and A. Ghazel, "Correlated power noise generator as a low cost dpa countermeasures to secure hardware aes cipher", in *2009 3rd International Conference on Signals, Circuits and Systems (SCS)*, 2009, pp. 1–6. DOI: 10.1109/ICSCS.2009.5412604.

[20] A. Mokari, B. Ghavami, and H. Pedram, "Scar-fpga : A novel side-channel attack resistant fpga", in *2009 5th Southern Conference on Programmable Logic (SPL)*, 2009, pp. 177–182. DOI: 10.1109/SPL.2009.4914903.

[21] *Moving Target Defense*. [Online]. Available: https://www.dhs.gov/science-and-technology/csd-mtd.

[22] M. Dunlop, S. Groat, W. Urbanski, R. Marchany, and J. Tront, "Mt6d: A moving target ipv6 defense", in *2011 - MILCOM 2011 Military Communications Conference*, 2011, pp. 1321–1326. DOI: 10.1109/MILCOM.2011.6127486.

[23] S. Venkatesan, M. Albanese, K. Amin, S. Jajodia, and M. Wright, "A moving target defense approach to mitigate ddos attacks against proxy-based architectures", in *2016 IEEE Conference on Communications and Network Security (CNS)*, 2016, pp. 198–206. DOI: 10.1109/CNS.2016.7860486.

[24] M. Azab and M. Eltoweissy, "Migrate: Towards a lightweight moving-target defense against cloud side-channels", in *2016 IEEE Security and Privacy Workshops (SPW)*, 2016, pp. 96–103. DOI: 10.1109/SPW.2016.28.

[25] *Managing time and costs in legacy component upgrades*, http://www.militaryaerospace.com/articles/print/volume-14/issue-12/departments/viewpoint/managing-time-and-costs-in-legacy-component-upgrades.html.

[26] D. Hallmans, K. Sandström, T. Nolte, and S. Larsson, "A method and industrial case: Replacement of an FPGA component in a legacy control system", in *Proc. IEEE 13th Intl. Conf. on Industrial Informatics*, 2015, pp. 208–214. DOI: 10.1109/INDIN.2015.7281736.

[27] G. Bloom, B. Narahari, R. Simha, A. Namazi, and R. Levy, "FPGA SoC architecture and runtime to prevent hardware Trojans from leaking secrets", in *Proc. 2015 IEEE Intl. Symp. on Hardware Oriented Security and Trust*, 2015, pp. 48–51. DOI: 10.1109/HST.2015.7140235.

[28] R. Karam, T. Hoque, S. Ray, M. Tehranipoor, and S. Bhunia, "Robust bitstream protection in FPGA-based systems through low-overhead obfuscation", in *Proc. 2016 International Conference on ReConFigurable Computing and FPGAs*, 2016, pp. 1–8. DOI: 10.1109/ReConFig.2016.7857187.

[29] V. Jyothi, M. Thoonoli, R. Stern, and R. Karri, "FPGA Trust Zone: Incorporating trust and reliability into FPGA designs", in *Proc. IEEE 34th International Conference on Computer Design*, 2016, pp. 600–605. DOI: 10.1109/ICCD.2016.7753346.

[30] *Xilinx ise in-depth tutorial ug695 (v14.1)*, https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ise_tutorial_ug695.pdf.

[31] *Fpga market size set to exceed usd 9.98 billion by 2022, with over 8.4% cagr from 2015 to 2022: Global market insights inc.* [Online]. Available: https://goo.gl/uEmByo.

[32] *Security for volatile FPGAs*. [Online]. Available: \sloppyhttp://www.cl.cam.ac.uk/techreportsUCAM-CL-TR-763.pdf.

[33] M. Majzoobi and F. Koushanfar, "Time-Bounded Authentication of FPGAs", *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 3, pp. 1123–1135, 2011, ISSN: 1556-6013. DOI: 10.1109/TIFS.2011.2131133.

[34] I. Hadzic, S. Udani, and J. M. Smith, "FPGA Viruses", in *Proc. Intl. Workshop on Field-Programmable Logic and Applications*, 1999, pp. 291–300, ISBN: 3-540-66457-2. [Online]. Available: http://dl.acm.org/citation.cfm?id=647926.739074.

[35] S. Trimberger, "Trusted Design in FPGAs", in *Proc. 44th Annual Design Automation Conference*, San Diego, California, 2007, pp. 5–8, ISBN: 978-1-59593-627-1. DOI: 10.1145/1278480.1278483. [Online]. Available: http://doi.acm.org/10.1145/1278480.1278483.

[36] S. Skorobogatov and C. Woods, "Breakthrough silicon scanning discovers backdoor in military chip", in *Proc. CHES'12*, Leuven, Belgium, 2012, pp. 23–40, ISBN: 978-3-642-33026-1. DOI: 10.1007/978-3-642-33027-8_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33027-8_2.

[37] P. Swierczynski, A. Moradi, D. Oswald, and C. Paar, "Physical Security Evaluation of the Bitstream Encryption Mechanism of Altera Stratix II and Stratix III FPGAs", *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 4, 34:1–34:23, Dec. 2014, ISSN: 1936-7406. DOI: 10.1145/2629462. [Online]. Available: http://doi.acm.org/10.1145/2629462.

[38] Y. Pino, V. Jyothi, and M. French, "Intra-die process variation aware anomaly detection in FPGAs", in *Proc. 2014 ITC*, 2014, pp. 1–6. DOI: 10.1109/TEST.2014.7035343.

[39] S. Mal-sarkar, A. Krishna, A. Ghosh, and S. Bhunia, "Hardware Trojan Attacks in FPGA Devices: Threat Analysis and Effective Countermeasures", in *Proc. ACM Great Lakes Symposium on VLSI*, 2014, pp. 287–292.

[40] R. Druyer, L. Torres, P. Benoit, P. V. Bonzom, and P. Le-Quere, "A survey on security features in modern fpgas", in *2015 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2015, pp. 1–8. DOI: 10.1109/ReCoSoC.2015.7238102.