2018-12-01

Assuring Intellectual Property Through Physical and Functional Comparisons

Adam Kendall Hastings
Brigham Young University

Assuring Intellectual Property Through Physical and Functional Comparisons

Adam Kendall Hastings

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Brad L. Hutchings, Chair
Jeffrey B. Goeders
James K. Archibald

Department of Electrical and Computer Engineering

Brigham Young University

ABSTRACT

Assuring Intellectual Property Through
Physical and Functional Comparisons

Adam Kendall Hastings
Department of Electrical and Computer Engineering, BYU
Master of Science

Hardware trojans pose a serious threat to trusted computing. However, hardware trojan detection methods are both numerous and onerous, making hardware trojan detection a difficult and time-consuming procedure. This thesis introduces the IP Assurance Framework, which drastically improves the time it takes design teams to test for hardware trojans. The IP Assurance Framework is implemented in two ways: The first method, Physical Assurance, compares instantiated IP blocks to a golden model via physical-level comparisons, while the second method, Functional Assurance, compares IP to a golden model using logical-level comparisons. Both methods are demonstrated to distinguish between tampered and untampered IP blocks, with a tolerable effect on IP timing and area.

Keywords: hardware security, hardware trojans, intellectual property, IP, assurance

ACKNOWLEDGMENTS

This thesis represents a summation of my research work during my Masters program at BYU. Many people helped me along my way here; Some of these people deserve special recognition. First and foremost, this work would not have come about without the help of Sean Jensen. Sean and I did almost all of this work in tandem. Many of the ideas formulated, discoveries made, and even code written in this work are the product of neither one of us alone, but rather of our collaboration. I feel privileged to have worked with such a brilliant engineer and good friend. Our collaboration was a thing of beauty, indeed [1], [2].

I would also like to thank my committee members for their helpful comments and feedback on the thesis. Their experience and wisdom has been invaluable. In particular, my graduate advisor, Dr. Brad Hutchings, deserves special recognition. This work is indebted to his guidance and mentorship.

Last but certainly not least is my wife, Sam. Her patience and encouragement were instrumental in making this thesis a reality. She also deserves some credit for making Figure 4.1, as well as letting me practice my thesis presentation in front of her countless times. Thanks again, Sam.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

In today's world, technology is underpinned by digital systems. Science, engineering, medicine, business, and government (among many others) are built upon and supported by the billions of computers, servers, and digital devices that make virtually all modern technology possible. These digital systems serve as the backbone for countless important applications, such as telecommunications, transportation, digital infrastructure, scientific research, artificial intelligence, robotics, and the internet. Unfortunately, the important nature of such applications make them a desirable target for hackers, who exploit technology for personal gain and often at the harm of others. A common way for hackers to exploit technology is to exploit the underlying digital systems.

Hackers can attack digital systems in many ways. For example, distributed denial of service (DDOS) attacks disrupt digital systems by flooding them with bogus internet traffic, rendering them inoperable and unable to provide their intended service [3]. DDoS attacks, like many network-based attacks, leverage weaknesses in computer networks to cause harm. Other attacks forgo network weaknesses in favor of software exploits, which find and exploit bugs in programs' code. Buffer overflows are a classic example of this type of attack [4]. In general, most people tend to think of these kinds of attacks—software and network exploits—when they think of hacking. What many people aren't aware of is that systems can be attacked at the hardware level as well. Just like software and networks, the very hardware itself of digital systems can be compromised to degrade performance, steal information, or cause device failure.

1.1 Hardware Trojans

How can hardware-based attacks compromise the security of a system? One category of attacks, called *hardware trojans*, has attracted attention in recent years. Hardware trojans are generally defined as malicious modifications to a circuit, either during design or fabrication [5].

They can be anything from a secret killswitch that disables a device to a hidden backdoor that leaks private information.

A considerable amount of research has been dedicated to discovering methods for detecting hardware trojans in digital designs. Generally, these detection techniques require careful design analysis using specialized tools, and often require in-depth knowledge of the specific detection technique. The many different available detection techniques, combined with the skill needed to successfully run each detection, makes thorough hardware trojan detection a difficult and laborious process. This is a problem, because organizations may not be willing to invest the large amount of time and effort needed to successfully vet a digital design for hardware trojans. Conversely, organizations who are especially concerned with security may find themselves spending far too much time running hardware trojan detection techniques on each new design they create. In either case, too much time and effort is required to thoroughly check for hardware trojans.

1.2 The IP Assurance Framework

This thesis aims to reduce the time and effort required to detect hardware trojans by introducing the *IP Assurance Framework*. The core insight in this framework is that reusable hardware such as Intellectual Property (IP) will be similar across various usages, meaning that the time and effort used to secure one instance of an IP from hardware trojans can be re-used to secure other instances of the IP from hardware trojans. Using this insight, the IP Assurance Framework enables hardware engineers to quickly and easily detect hardware trojans in their IP.

1.2.1 Intellectual Property

The IP Assurance Framework is based on Intellectual Property, or IP (not to be confused with the other IP, *Internet Protocol*). In digital design, Intellectual Property is usually defined as a discrete, reusable piece of logic or hardware design. The reusability of IP makes them a practical choice in hardware design: Engineers can spend less time and effort redesigning things that already exist, saving design teams time and money. These benefits have made IP widespread—According to a 2013 survey, 68% of new design content is made of IP (See Figure 1.1). This means that the

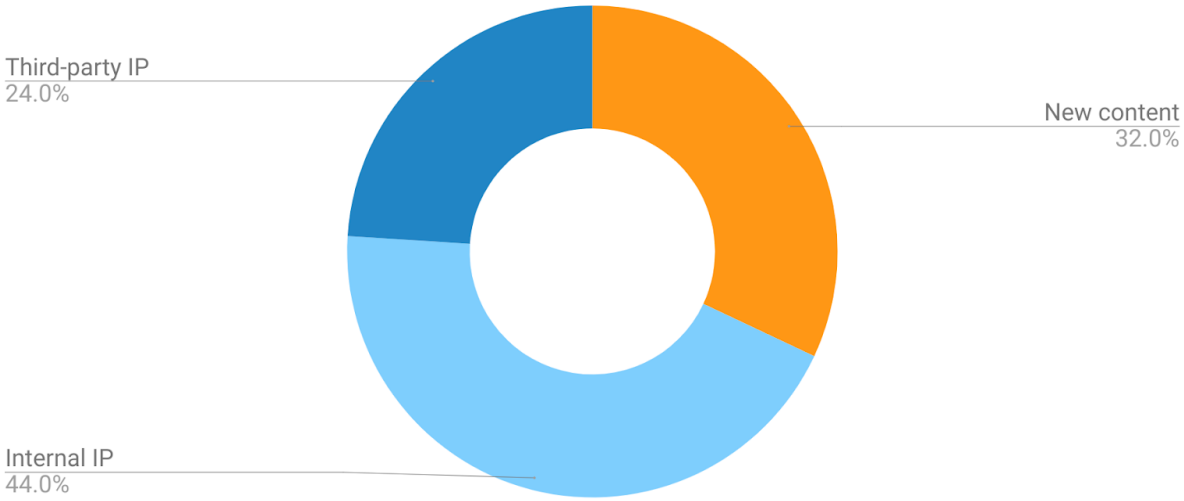

Figure 1.1: Pie chart showing the distribution of new design content, according to [6].

IP Assurance Framework can be used on an overwhelming majority of new design content, vastly improving engineers' ability to thoroughly check for hardware trojans.

1.2.2 Framework Overview

The principle benefit of the IP Assurance Framework is that it allows IP users to take advantage of state-of-the art hardware trojan detection, without requiring the IP users themselves to be experts in hardware trojan detection. Instead of placing the burden of hardware trojan detection on the IP user, this framework places the burden on the IP *creator*. This framework promotes the idea that the IP creator should create a gold standard model of the IP, a gold standard which has been thoroughly vetted using state of the art HT detection techniques, and is free of hardware trojans[1]. Once the gold standard has been established, IP users can compare an IP core in their design to the gold standard IP. If the user's IP is "equivalent" to the gold standard, then the user's IP is therefore also free of hardware trojans.

How exactly can a user's IP be compared to a gold standard IP? This is the central question this thesis aims to address. Due to the enormous complexity of modern CAD tools, and the way they modify, mix, and mangle IP blocks, determining equivalence between any two IP blocks is a highly non-trivial task. This thesis proposes two feasible methods of making the comparison

[1] insomuch as such a thing is possible

between a gold standard IP and a user's implementation of that IP. The first method compares physical-level details of the IPs, while the second method compares the IPs' functionality. Both methods are shown to be effective at distinguishing between tampered and untampered designs, demonstrating that they are both viable options for providing assurance to IP.

1.3 Thesis Outline

This thesis proceeds as follows: Chapter 2 first establishes a background on Field-Programmable Gate Arrays (FPGAs) and FPGA tools, which are used extensively throughout this work. A background on hardware trojans and hardware trojan detection is then established in Chapter 3.

Chapter 4 then fully introduces the IP Assurance Framework. Benchmarks to test the framework are presented in Chapter 5. Chapters 6 and 7 delve into the two methods used to compare IP. A discussion of the work is provided in Chapter 8. Finally, Chapter 9 concludes this thesis, discusses future work, and lists personal contributions made to this project.

CHAPTER 2. FPGAS AND FPGA TOOLS

2.1 Introduction to FPGAs

Field-programmable gate arrays (FPGAs) are integrated circuits that implement digital logic. FPGAs are unique from other integrated circuits in that they are *reprogrammable*, meaning that they can change their functionality after their date of manufacturing. For example, an FPGA can be configured to implement a given digital circuit, only to later be reconfigured to implement a completely different digital circuit. FPGAs can theoretically be reconfigured in this way an infinite number of times. This gives them a flexibility that has made them a popular choice for hardware designers.

2.1.1 FPGAs vs. ASICs vs. CPUs

In traditional digital hardware design, engineers had two choices when designing digital systems: Application-specific integrated circuits (ASICs) and central processing units (CPUs). ASICs can compute very fast and consume little space and area, but are very hard to design, and can't be changed once they are manufactured. This means that they are limited in what they can compute. CPUs overcome this problem by being Turing complete, meaning that they can compute anything that is computable; However, this generally comes at the cost of speed. FPGAs span the gap between ASICs and CPUs (see Table 2.1)—They are fast like ASICs, but also flexible like

Table 2.1: Summary of differences between ASICs, FPGAs, and CPUs

	ASIC	**FPGA**	**CPU**
Speed	very fast	fast	slow
Size	very small	small	large
Power Consumption	very low	medium	high
Non-Recurring Engineering Cost	extremely high	low	low
Cost per Device	very low	medium	low

CPUs. And while they are not as easy to use as CPUs, they are orders of magnitude easier to use than designing and building a custom ASIC.

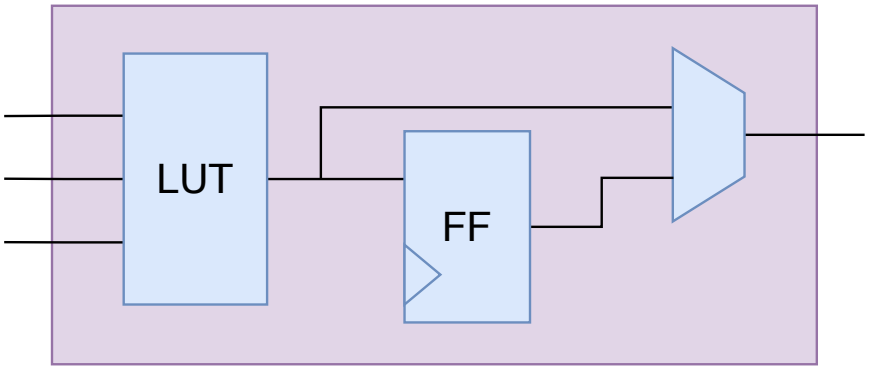

Figure 2.1: A simple Configurable Logic Block, or CLB. CLBs in modern FPGAs are typically much more complex, but generally follow the same pattern of LUTs, flip-flops, and routing multiplexers.

2.1.2 FPGA Architecture

Standard FPGA architecture can be thought of as a fabric, with thousands or even millions of reprogrammable blocks woven together by a myriad of interconnecting wires. These reprogrammable blocks are called Configurable Logic Blocks [1], or CLBs. In its simplest form, a CLB contains a lookup table (LUT), a flip-flop (FF), and a routing multiplexer, each of which is programmable by the user. The lookup table implements combinational logic, while the flip-flop implements sequential logic. The multiplexer then picks which signal (combinational or sequential) leaves the CLB. In modern FPGAs, CLBs are often subdivided into multiple *slices*, each of which may contain several LUTs, flip flops, and routing multiplexers (see Figure 2.5 for a detailed image of a slice in the Artix-7 architecture).

FPGAs typically arrange their CLBs in a 2d array (see Figure 2.2). In between the CLBs, there are wires and switchboxes that connect CLBs to each other or to external pins on the FPGA. A more detailed example of this 2d array of logic blocks and interconnect wires can be seen in Figure 2.3.

[1] Also called Logic Array Blocks, depending on vendor

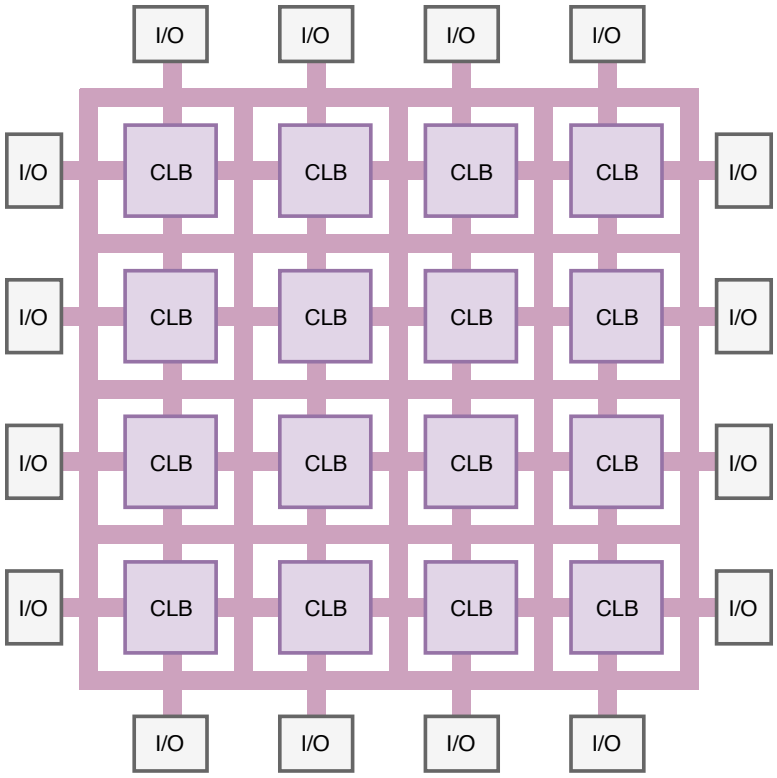

Figure 2.2: Overview of a simple FPGA architecture. Reproduced from [7].

2.1.3 Applications

FPGAs' unique qualities make them a popular choice in many engineering applications. One such application is systems with a low production volume: Many systems often need the speed of an FPGA or ASIC to meet timing constraints, yet are produced in small enough quantities that makes it hard to justify the enormous up-front cost of an ASIC. In these scenarios, it is simply more economical to use an FPGA. The decision to use FPGAs to save money can be seen across many industries, particularly in the aerospace, automotive, and defense industries.

FPGAs are also used as hardware accelerators. They can be reconfigured to accelerate many different kinds of algorithms, and are especially good at parallelizable algorithms. For example, in many problems where the solution is best found by brute force, it is possible (and usually advisable) to make multiple attempts at finding the solution concurrently. This can easily be done on an FPGA, where their customizability, concurrency, and reprogrammability allow them to implement essentially any parallel digital algorithm. This makes FPGAs an increasingly popular choice when accelerating parallel algorithms, especially in network servers and data centers.

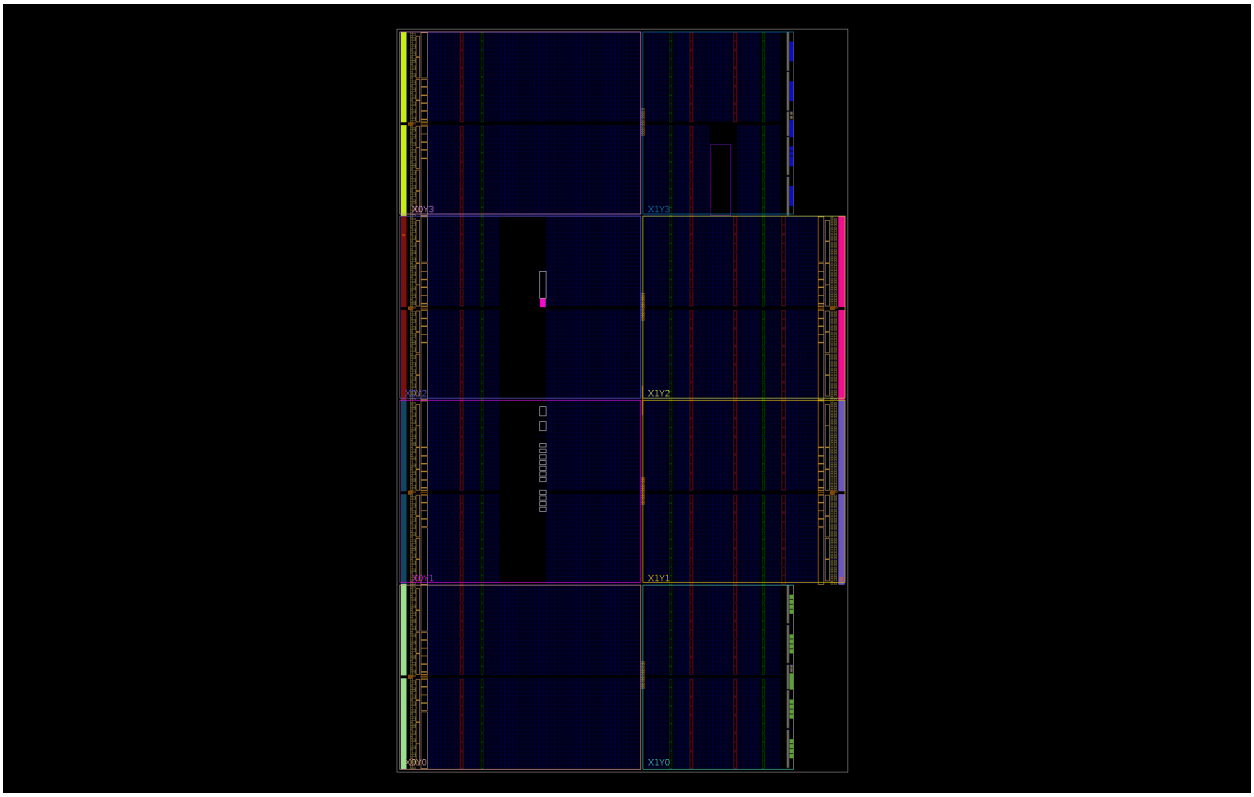

Figure 2.3: An example of an FPGA as seen in the Vivado device viewer. While FPGAs are often described as a regular and repeating array of logic blocks, in practice, they are slightly more complicated. In this FPGA, a Xilinx Artix-7, the architecture contains the expected array of logic blocks (shown in blue), but also contains BRAM (block random-access memory, shown in the red columns) and DSPs (digital signal processing blocks, shown in the green columns).

2.2 Why Was This Work Conducted on an FPGA?

The focus of this thesis is the IP Assurance Framework, which allows IP users to fully take advantage of state-of-the-art hardware trojan detection and mitigation. The ideas promoted in this thesis are not exclusive to FPGAs or FPGA design. Furthermore, I expect that this work should easily extend into non-FPGA digital design.

That being said, in order to demonstrate the feasibility of implementing this framework, it was necessary to actually build and test a working model of the framework in action. Building and testing an implementation of the framework required a sufficiently advanced and usable digital design tool suite, as well as knowledge on how to use and manipulate the tools. My research team found the Xilinx Vivado Design Suite to fit our needs—it's powerful enough to implement the framework, and can interface with TCL (Tool Command Language), which allowed us to heavily

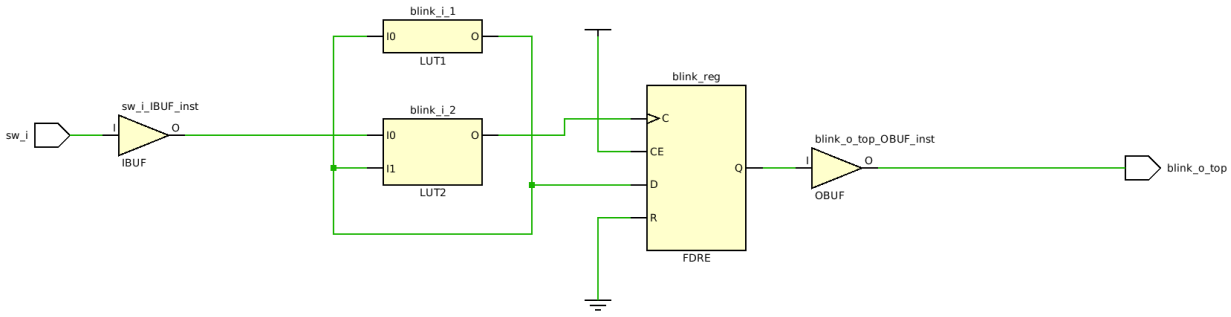

Figure 2.4: A schematic of a simple circuit. This circuit was synthesized from Verilog code into a netlist using the Vivado synthesis tool. This schematic shows how the Vivado synthesis tool represents a design in terms of Xilinx primitives such as LUTs, flip-flops, and buffers, rather than in terms of pure logic gates.

script and automate our testing. And perhaps above all else, the Vivado Design Suite was a tool that we were all familiar with. Thus while this work could have been conducted on a number of different tools and platforms, it exclusively used Vivado for creating digital designs. Therefore, the details of this thesis are very much intertwined with the Vivado Design Suite. As such, this thesis makes many references to the tool and its functions and features.

Because the details of this work are heavily intertwined with Vivado, it is necessary to first establish the basic features of the tool. Vivado is created by Xilinx, an FPGA vendor, and is primarily used to make designs for FPGAs [2]. In simple terms, Vivado is a tool that transforms a user's digital circuit design into a binary file that can be used to program the user's digital circuit onto a target FPGA. The process of converting a user's design into this binary file (called a bitstream) is done in three major steps: synthesis, implementation, and bitstream generation.

Synthesis

The first step in the FPGA design process is *synthesis*. This step begins with a user's digital circuit design, usually in the form of a hardware description language (HDL) such as Verilog or

[2]Readers familiar with other FPGA vendors' design tools (such as the Intel/Altera Quartus design tools) will find that Vivado is similar to these other tools. As a side note, it is expected that these other tool suites should also be able to implement the IP Assurance Framework in ways similar to those described in this thesis.

VHDL. The synthesis tool parses the text-based HDL code and transforms it into a logic gate-level representation of the user's design [8]. This gate-level representation is a complete representation of the user's design at the logical level.

This process of turning a user's HDL design into a gate-level description is standard across digital design, and is unique neither to Vivado nor FPGA design tools in general. However, unlike other synthesis tools (particularly non-FPGA synthesis tools), Vivado does not produce a gate-level representation in the strict sense. Instead, the Vivado synthesis tool produces a design representation in terms of Xilinx primitives, *e.g.* LUTs, flip-flops, and buffers. This representation is logically and functionally equivalent to a gate-level representation, but is at a level of abstraction that better models the way an FPGA behaves. This type of representation, in terms of primitives (or cells) rather than gates, is often referred to as a *netlist*. An example of a netlist displayed as a schematic can be found in Figure 2.4.

Implementation

The second step in the Xilinx FPGA design process is *implementation*. This step uses the netlist produced during synthesis to create a placed and routed design ready for bitstream generation. This is accomplished in three sub-processes: optimization, place, and route:

1. Optimization: This step optimizes the netlist to make it smaller, faster, and consume less power. For example, consider a 3-input LUT acting as an AND gate where one of the inputs is a constant 1. The constant input can be removed, and the 3-input LUT can be reduced to a 2-input LUT. Optimizations such as this can improve the FPGA's performance without changing its functionality.

2. Place: This step takes the Xilinx primitives in the optimized netlist and maps them to specific primitives on the FPGA. In Xilinx terminology, this step is the process of taking cells in the design, and mapping them to specific BELs (Basic ELements) on the device.

3. Route: Once the cells have been placed, they are routed together. This steps involves finding available wires in the FPGA to make all necessary connections between cells. An example of a placed and routed circuit can be found in Figures 2.5 and 2.6.

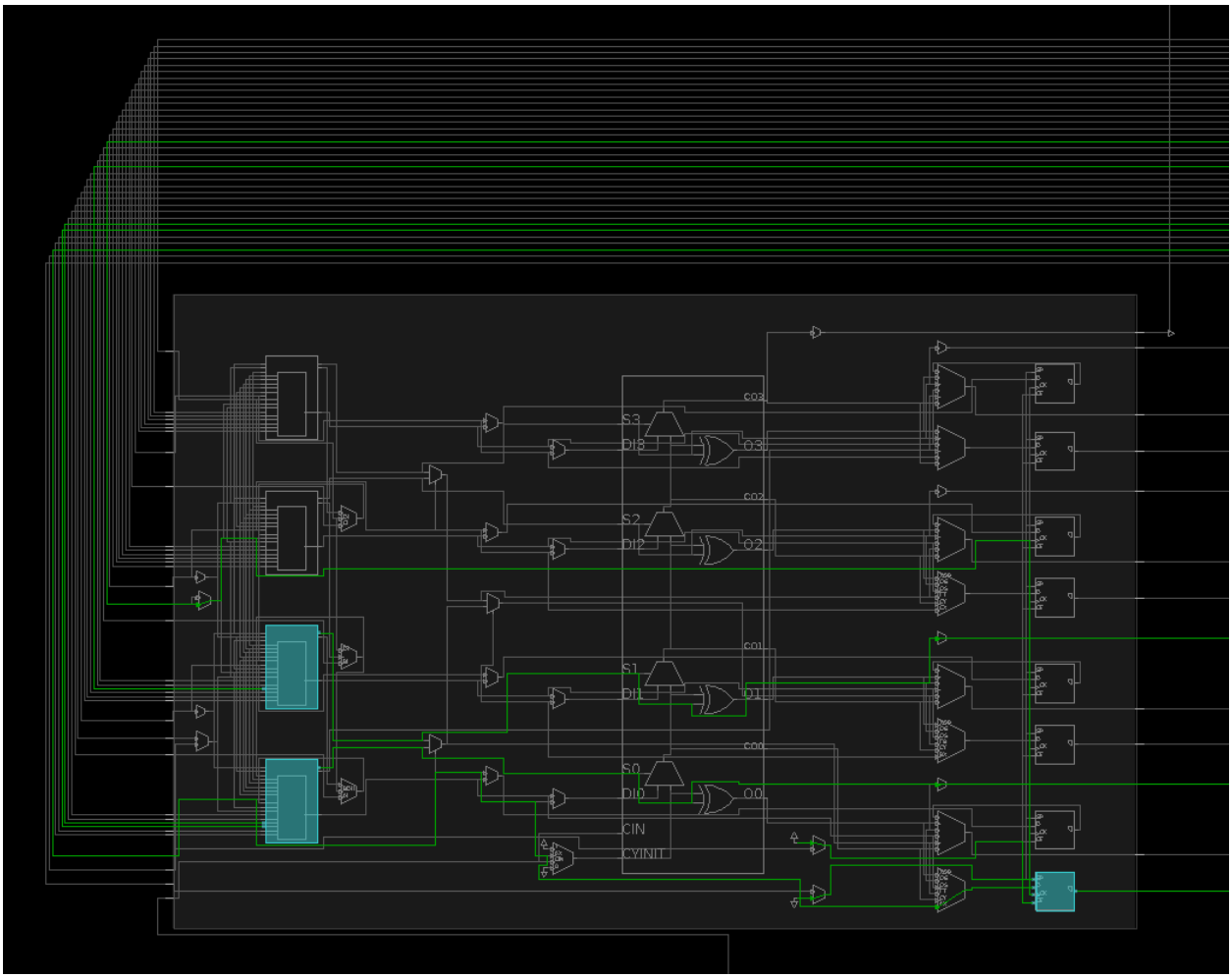

Figure 2.5: An example of a placed and routed design, as seen in the Vivado device viewer. This design implements the netlist shown in Figure 2.4 onto a Xilinx Artix-7 FPGA. This design was so small that its logic fit entirely into a single slice. The cells used in this implemented design are highlighted in teal, while the nets connecting them together are highlighted in green. Note that the cells in this image correspond to the two LUTs and one flip-flop found in Figure 2.4.

Bitstream Generation

Implementation turns a netlist into a model of the design as it will be implemented on a target FPGA. *Bitstream generation* then takes this model of the design and translates it into a large binary file called a bitstream. This bitstream contains all the configuration data the FPGA needs to program itself into the user's design. This configuration data can be anything from memory initializations to routing information to LUT equations. Once Vivado has generated a bitstream,

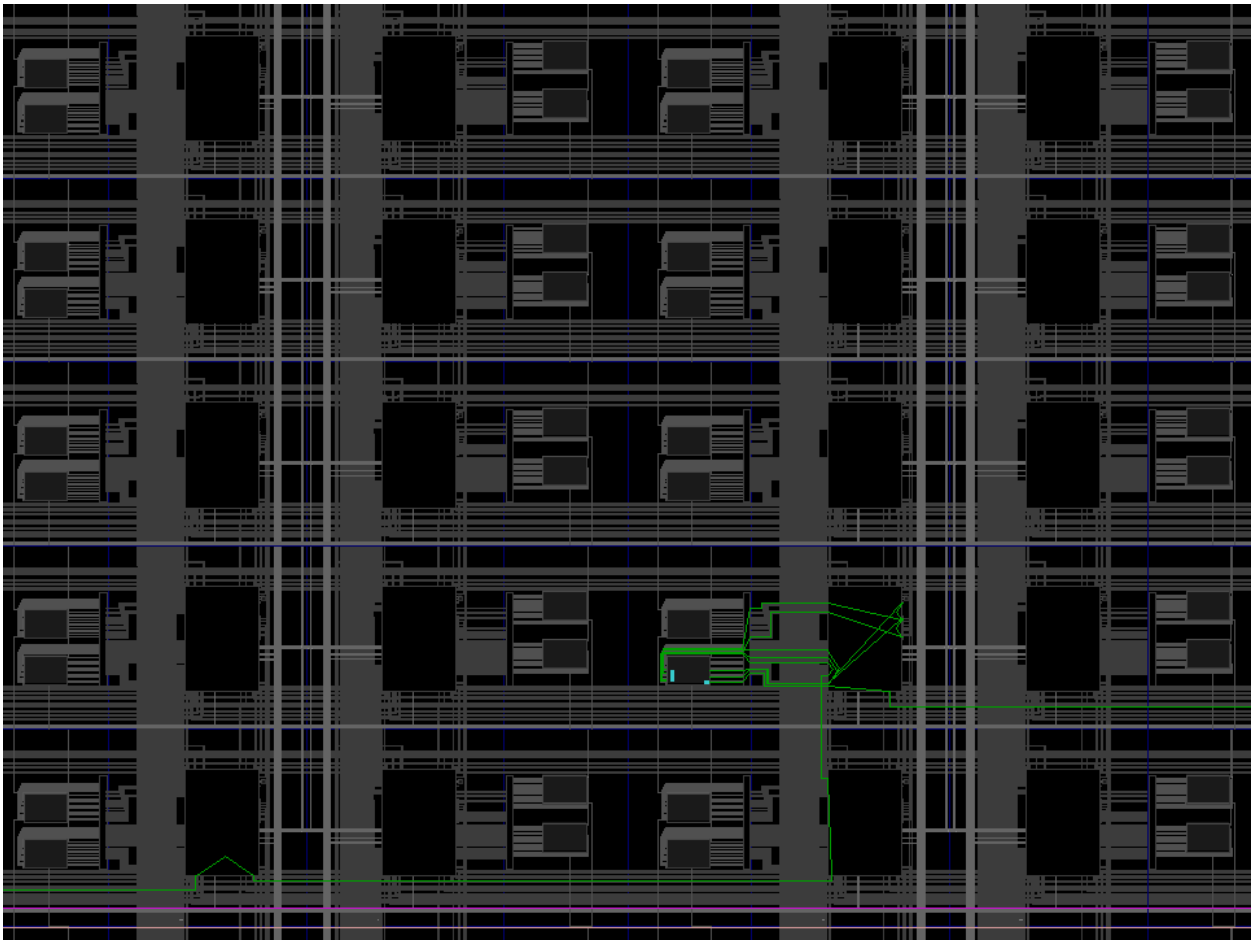

Figure 2.6: A zoomed-out view of the design in Figure 2.5. The nets leave the slice and enter a switchbox, which routes the signal to other places within the FPGA. In this case, the switchbox routes the signals to input and output buffers (not shown in this image), and eventually to the package pins on the FPGA.

the bitstream can be downloaded onto the target FPGA device. The FPGA then uses the bitstream to configure itself into the user-specified design.

CHAPTER 3. HARDWARE TROJANS

This thesis is based upon foundational work on hardware trojans. As such, this chapter introduces this foundational work in three sections. The first section gives a taxonomy of hardware trojans. The second details various hardware trojan detection methods. With a background in the subject established, this chapter then presents this thesis's threat model.

3.1 Taxonomy of Hardware Trojans

Hardware trojans are defined as "modifications to original circuitry inserted by adversaries to exploit hardware or to use hardware mechanisms to gain access to data or software running on the chips" [5]. Using this definition, Tehranipoor and Koushanfar decompose hardware trojans into three main categories: Physical characteristics, activation characteristics, and action characteristics [5] (See Figure 3.1).

3.1.1 Physical Characteristics

Tehranipoor and Koushanfar define the physical characteristics of a hardware trojan to be various hardware manifestations of trojans. This category is subpartitioned into *distribution*, *structure*, *size*, and *type*. The distribution category describes the location of the trojan within a chip's physical layout. The structure category classifies trojans based on whether they change a chip's physical form factor or not. The size category accounts for the number of components within a chip that have been added, removed, or compromised. Finally, the type category partitions hardware trojans into *functional* and *parametric* classes, where functional hardware trojans are realized by adding or removing gates or transistors, whereas parametric hardware trojans are realized through modifications to existing wires and logic.

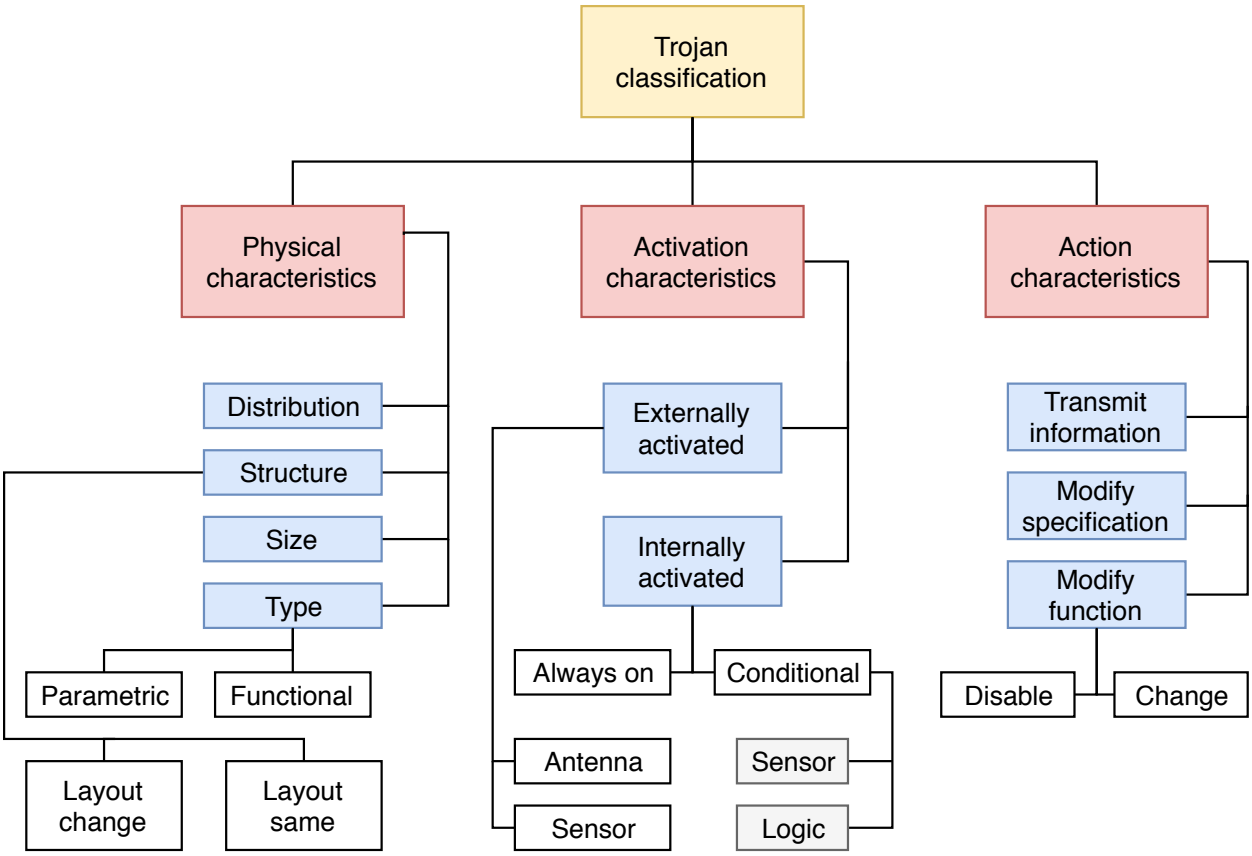

Figure 3.1: Taxonomy of hardware trojans. Reproduced from [9] via [5].

3.1.2 Activation Characteristics

The activation characteristics are the criteria that causes a hardware trojan to carry out its disruptive function [5]. This can be broken into two categories: Externally-activated and internally-activated hardware trojans. Externally-activated hardware trojans rely on some signal from the outside world (e.g. from an antenna or sensor) to trigger the trojan's payload. In contrast, internally-activated hardware trojans rely on some internal state or event to activate the payload. This is broken down further into *always on* trojans (meaning the trojan is always active and can cause disruptions at any point), and *conditional* trojans, which wait for some sensor reading to occur, or for some logic state to be reached or secret value to be encountered. In conditional hardware trojans, the trojan lies dormant until the condition is met.

3.1.3 Action Characteristics

Tehranipoor and Koushanfar's taxonomy also includes a category for action characteristics, which describe the trojan's payload. For example, trojans in the *transmit information* class leak information to an adversary. *Modify specification* trojans change a device's parametric properties, such as circuit delay. Finally, trojans in the *modify function* alter the chip's behavior by adding or modifying logic, or by bypassing or disabling it altogether.

3.1.4 Examples

To illustrate how actual hardware trojans fit into Tehranipoor & Koushanfar's taxonomy, let's look at some real-world examples. Note that these examples represent only a very small subset of known hardware trojan attacks.

Example #1: Stealing Encryption Keys

Jin et al. implement a rather simple hardware trojan in [10]. In this example, a hardware trojan was placed in an FPGA design by modifying the Register Transfer-Level code to leak a device's encryption key when triggered. In this hardware trojan, the trigger is a secret string input to the device via a keyboard. When the device encounters this secret string, it leaks the encryption key via a serial RS232 port. This hardware trojan added 0.8% more flip-flops and 6.8% more LUTs than the original, untampered design.

How does this attack fit into Tehranipoor & Koushanfar's taxonomy of hardware trojans? Let's first look at the physical characteristics: Because this hardware trojan was implemented on an FPGA, any change to the RTL code (and any re-synthesis and re-implementation in general) changes the distribution and the structure (layout) of the original design. The trojan has an effect on size by increasing the number of flip-flops and LUTs used. This is a functional hardware trojan, because it adds logic to the original design. What about activation characteristics? This is an externally activated attack, since the modified design waits for a signal from an outside sensor (keyboard). The action characteristic of this hardware trojan is to transmit information.

Example #2: Denial of Service

Baumgarten et al. implement a Denial-of-Service (DoS) attack on an FPGA in [11] by inserting malicious RTL into the original design. In this attack, the device's encryption key is corrupted, rendering the ciphertext undecipherable. The hardware trojan is triggered by an internal counter. This hardware trojan lies dormant until the counter's most significant bit is high, at which point it begins corrupting the encryption key. By making the counter sufficiently large, the authors argue that such a trojan could be present yet still pass validation tests.

This hardware trojan is inserted by modifying a design's RTL, making this a functional attack. Because this attack was implemented on an FPGA, this means that the distribution, structure, and size of the original design will change. The hardware trojan is internally activated by the counter. The trigger is conditional, because it requires the counter's highest bit to be set before activating. This trigger is logic-based, since it originates from within the logic of the tampered design. Finally, the action of the hardware trojan is to modify the original design's function by making its output unusable. Under Tehranipoor & Koushanfar's taxonomy, this attack modifies functionality by disabling the device's ability to properly communicate securely.

3.2 Hardware Trojan Detection

To address the threat of hardware trojans, researchers have developed methods to detect and mitigate compromised hardware. This section selects a few of these methods and describes how they work. Rather than a comprehensive guide to hardware trojan detection, this section serves to inform the reader of the huge variety of methods available, as well as to highlight the inherent difficulty in hardware trojan detection .

The goal of hardware trojan detection is to find and remove hardware trojans before they ever make it into a final design in the field. This can be done at several steps of the digital design process; This section highlights a few methods that can detect trojans at the design step and post-fabrication steps, respectively. For more on hardware trojan detection, see [5], [9], [12], [12]–[21], [21]–[26].

Example #1: FANCI

In [27], Waksman *et al.* introduce the hardware trojan detection tool FANCI, or Functional Analysis for Nearly-unused Circuit Identification. FANCI relies on the assumption that hardware trojans (referred to here as backdoors) are nearly always dormant, meaning that they rarely influence a device's output. By using a static boolean functional analysis on an RTL-level design, FANCI identifies nearly-unused parts of the design and flags them as being suspicious. Once the suspicious portions of a design have been flagged, the engineers can review each case one-by-one and determine if the flagged portion was a backdoor or simply a false positive. The authors then demonstrate that this technique, when applied to the TrustHub benchmark suite [28], detects each of the backdoors while maintaining a permissible number of false positives.

Example #2: IC Fingerprinting

Many hardware trojan detection methods use side-channel analysis (typically on power consumption or timing) to flag potentially suspicious behavior. According to [5], Agrawal et al. were the first to employ such a technique in [29]. In this paper, Agrawal et al. introduce a technique to detect hardware trojans by building a "fingerprint" from fabricated Integrated Circuits (ICs) using various side-channel signals, such as power, electromagnetic radiation, and thermal emissions. This fingerprint is constructed for several selected ICs within an IC family (i.e. ICs manufactured using the same process at the same foundry). Most importantly, these selected ICs are reverse-engineered and validated to be free of hardware trojans using techniques such as demasking and layer-by-layer examination of X-ray scans[1]. Once a verified set of ICs has been established, all other ICs from the same family can compare fingerprints to the trusted set of ICs to verify that no hardware trojans have been included.

3.3 Threat Model

How does a hardware trojan secretly make its way into a design? Let's review the digital design process discussed in Chapter 2 and note potential vulnerabilities. The most obvious step for hardware trojan insertion is at the RTL design level. In this step, a malicious agent could add

[1] Note that this likely will be an extremely expensive endeavor

a hardware trojan to IP by tampering with the IP's HDL. This malicious agent could be attacking remotely via a network or could even be a planted malicious agent within a design team. In either case, the hardware trojan is inserted by tampering with a design's HDL code.

A less obvious attack vector is through CAD tools themselves. It is possible that a user's CAD tools (such as the Xilinx synthesis and implementation tools) could be compromised. In this situation, the tools themselves secretly add hardware trojans at the synthesis or implementation steps. To create truly assured designs, IP users shouldn't assume that their third-party CAD tools are completely safe.

CHAPTER 4. IP ASSURANCE FRAMEWORK

As discussed in Chapter 3, hardware trojans exist in many forms and pose a serious threat to trusted computing. To mitigate the problem, hardware trojan detection techniques have been developed. However, these techniques are difficult and time-consuming to apply, and often require expert or specialist knowledge. The hardware security community has developed countless ways to detect hardware trojans, but nothing on how to manage and apply all these techniques in a practical manner.

The ability to easily and effectively apply the plethora of available hardware trojan detection techniques is a serious need in the hardware security community. At the present, too much time and effort is required to apply hardware trojan detection to each new design, especially with regards to IP. When considering that IP is, by design, meant to be used and re-used, it becomes obvious that applying hardware trojan prevention techniques to each new instance of an IP is largely redundant work. What if it didn't have to be? What if the tedious step of detecting hardware trojans in IP only had to be done once, henceforth sparing future users of the IP from going through the same pain and effort? This is the central issue this thesis seeks to address, and was the motivation behind developing the IP Assurance Framework.

4.1 Framework Architecture

The IP Assurance Framework aims to reduce the amount of time and effort needed to detect hardware trojans in IP. It consists of three parts: IP Creation, IP Instantiation, and IP Comparison (See Figure 4.1).

IP Creation

The IP Creation begins with a trusted IP vendor. After the IP has been made to specification, the trusted vendor thoroughly combs through the IP using the hardware trojans detection

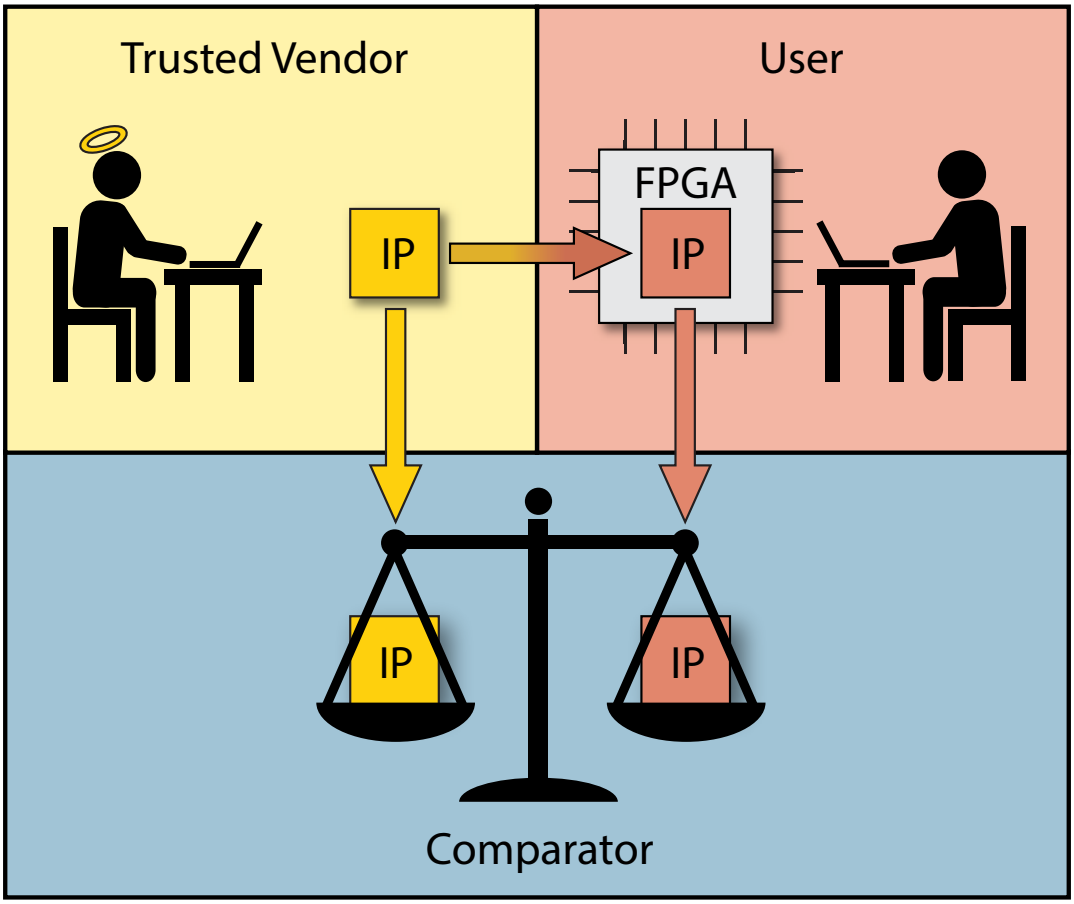

Figure 4.1: The IP Assurance Framework

techniques discussed in Section 3.2. Once the IP has been thoroughly examined, it is deemed to be safe to use. This vetted, secure IP then becomes the gold standard for all future copies and instantiations of the IP. In this thesis, gold standard IP is often referred to as the *trusted IP*.

IP Instantiation

Once the trusted vendor has created and vetted the IP, it is ready for distribution. The IP user can then take the distributed IP and instantiate it into their design. It is at this step that the user is vulnerable to the threats discussed in the threat model in Chapter 3. Because of this, the *instantiated* IP must be treated as if it were infected with hardware trojans, and may be unsafe to use.

IP Comparison

There are now two copies of the IP: the vetted, secure *trusted IP*, and the potentially infected *instantiated IP*. If the user compares the two IPs and can demonstrate that their instantiated IP is identical (more on this later) to the trusted IP, then the user can infer that their instantiated IP has not been tampered with. The key to the IP Assurance Framework is that this comparison between IPs can be done quickly, easily, and thoroughly. If such a comparison is possible, then the user has an easy way of verifying that their IP is safe to use. The principle benefit of this approach is that the arduous hardware trojan detection process only has to occur *once*, and can henceforth be used as a gold standard reference for future copies of the IP.

4.2 Implementing the Framework

What does it mean for two IPs to be "identical"? The notion of identicality can mean different things, depending on the user's needs. The definition of "identical", as well as the corresponding IP comparison methods, will depend on the way the IP Assurance Framework is implemented. This thesis describes two implementations of the framework, each of which have their own notions of identicality, and likewise their own methods of comparing IP.

Physical Assurance

The first implementation of the IP Assurance Framework is *Physical Assurance*. In this implementation, the IP user is concerned with verifying that the *physical structure* of the instantiated IP is identical to the trusted IP. If the user can demonstrate that their placed and routed IP is an exact replica of the trusted IP, then the user can be assured that their IP has not been tampered with *at the physical level*. In this implementation, the IP comparison step examines the physical details of the IPs—such as placement and routing—and verifies that the trusted and instantiated IPs match each other. This implementation approach is discussed in detail in Chapter 6.

Functional Assurance

The second implementation of the IP Assurance Framework is *Functional Assurance*. In this implementation, the IP user is concerned with verifying that the *functionality* of the instantiated IP is identical to the trusted IP. If the user can demonstrate that their instantiated IP behaves exactly like the trusted IP under all conditions, then the user can be assured that their IP has not been tampered with *at the functional level*. In this implementation, the IP comparison step examines the logic of the IPs, and verifies that, given the same inputs, the trusted IP and instantiated IPs always produce the same output. This implementation approach is discussed in detail in Chapter 7.

4.3 Analogy of Design Reuse

In many ways, the IP Assurance Framework is analogous to the principles of design reuse. In digital design, hardware components are often designed to be used and reused in multiple places across multiple designs and devices[1]. This reuse of design material has several advantages: First, reusable components can accelerate the design process by having engineers spend less time re-designing components from scratch [30]. Second, reusable components are already tested and proven, leading to better design performance and reliability [31]. And third, specialized reusable components can be designed by experts and optimized before being disseminated for reuse, resulting in better overall performance [31]. In general, the goal of design reuse is to produce higher-quality results in less time by reusing previous work.

The IP Assurance Framework has similar goals and methods as design reuse. For example, the specific advantages of the IP Assurance Framework mirror those of design reuse very closely: First, the design process can be accelerated by making the hardware trojan detection step quicker and easier. Second, a gold standard IP, which has been implemented across a multitude of designs and without incident, gives credence to the safety of the IP. And third, the hardware trojan detection process can be conducted by experts, resulting in a much more thorough vetting process than if done by non-experts. In general, by reusing the laborious work that is hardware trojan detection, engineers can produce safer, more assured designs without having to go through the effort themselves.

[1] For example, the use and prevalence of IP itself perfectly illustrates how engineers reuse previously-made components in digital design.

CHAPTER 5. BENCHMARKS

With the framework developed, it was time to test if it could actually be successfully implemented. Was it possible to determine equivalency between trusted and instantiated IP? Could our equivalence checker also detect malicious tampering? In order to confidently find out either way, we needed to conduct tests and collect data.

We assembled a large suite of benchmark circuits from which we could create tests. Because of the complexity of FPGA design tools, and the many ways they modify designs in unexpected ways, we decided that we needed a sufficiently large suite of benchmarks. In total, we created 23 benchmark designs, each of which contained at least one IP core. Using this benchmark suite, we were able to run tests on 53 independent IP cores. This benchmark suite, along with additional tools created during this work, can be found at `https://github.com/byuccl/ipassurance`.

5.1 Creating the Benchmarks

To rigorously test our Physical and Functional Assurance methods, we needed a benchmark suite of designs containing IP cores. The IP cores in this benchmark suite would ideally represent IP as it is actually used in real-life scenarios. Creating realistic scenarios meant creating a benchmark suite that is representative of IP as it is used "in the wild". This meant creating a benchmark suite that both contained different types of IP, and used IP in a variety of different ways.

Assembling a large, diverse suite of benchmarks by hand, each containing unique IP blocks, is hard and time-consuming, and we couldn't dedicate the time or effort needed to make the designs ourselves. So instead, we used two techniques to quickly assemble a large suite of benchmarks. The first technique was to take publicly available IP cores from OpenCores [1] and stitch them together without regard to functionality. Using this technique, we created 43 benchmarks, each

[1] OpenCores (`www.opencores.org`) is a website that serves the digital design community. OpenCores features a large library of free and open source IP cores which can be synthesized into designs by Vivado.

with a unique IP block. We called benchmarks created this way "synthetic benchmarks", because of the way they were artificially created. The second technique used to assemble our benchmark suite was to simply take an existing design and extract IP cores from it. Both techniques are discussed in further detail in the following subsections.

5.1.1 Synthetic Benchmarks

As previously stated, the synthetic benchmarks were made by randomly stitching together IP blocks found on OpenCores by hand. As such, these benchmarks were not designed to be usable, functional circuits. However, using this technique, I was able to quickly assemble a large suite of benchmarks that were sufficiently realistic enough for the purposes of this work.

To simulate the diversity of IP in the wild, I constructed each of the synthetic benchmarks under deliberately dissimilar conditions. The following list explains some of techniques I used to create variation among synthetic benchmarks:

- Designs were made to be a wide range of sizes. On the smallest end of the spectrum was the `counter` design, which consisted of 13 slices. By comparison, the largest design, `jpegencode`, used 13,103 slices.
- Some IP cores had all inputs and outputs used, whereas other IP cores had dangling inputs, dangling outputs, or both.
- IP cores were stitched together in different orders. For example, some benchmarks were stitched together in a completely feedforward manner, meaning that outputs from a given IP block were never fed back into any of the IP block's inputs. By comparison, some IP blocks had feedback loops from output directly back into an input, or even through other IP blocks before looping back into an input.
- Some benchmarks kept their IP blocks completely independent from each other, *i.e.* there were no wires connecting any IP blocks together, and therefore the IP blocks were isolated from each other.
- Some designs contained only a single IP block, while other designs contained several IP blocks.

- Designs were made at varying degrees of hierarchy. Some designs were made using Vivado's default hierarchy, where a design is represented as a hierarchical tree, in which the top module or component has submodules or subcomponents, each of which may in turn have their own submodules or subcomponents, and so on. In comparison, some designs were *fully flattened*, meaning that Vivado represents the design with no hierarchy.

- Some IP cores were given hard-coded inputs, such as a logical 1 or 0.

By varying the conditions under which the designs were created, it was hoped that the resulting synthetic benchmarks would be more representative of the complexity and diversity in real-world designs.

Are These Synthetic Benchmarks Any Good?

A possible criticism of our method of creating benchmarks is that it creates useless designs that do not resemble "real-world" designs. A plausible corollary to this criticism is that testing our framework with these synthetic designs might invalidate our findings, *i.e.* that demonstrating an effective implementation of the framework on synthetic designs does not imply that our implementation will work on functional, "real-world" designs.

Let's start with the basis of the criticism—that our synthetic benchmarks do not resemble real designs. This claim is largely true. Our synthetic designs likely do not resemble any functional design in use today. However, the corollary to the criticism—that the synthetic nature of the benchmarks may invalidate findings—is doubtful. This is because our implementations of the IP Assurance Framework—both Physical and Functional—only care about the circuit-level details of a design, and have no regard for design functionality. As long as these synthetic benchmarks simulate the way Vivado interacts with real-world design, then the synthetic benchmarks should be good enough for the purposes of this thesis.

5.1.2 LEON3 Benchmark

To address any remaining concerns regarding the quality of the synthetic benchmarks, we decided to also include real, industry-proven designs into our test suite. We found the LEON3

soft processor core (`www.gaisler.com/leon3`) to be a good test design—it's a large real-world design, widely used, free and open source under the GNU General Public License. The LEON3, like other soft processors, is a synthesizable design, written in VHDL, rather than a hardened silicon design. This means that the LEON3 can be synthesized and implemented onto an FPGA. After configuring our own LEON3 processor (synthesizing a LEON3 from scratch requires setting some parameters), we selected 10 VHDL components to act as target IP blocks.

5.2 Baseline Designs

This thesis describes two methods of implementing the IP Assurance Framework. What effects do these methods have on the performance of a circuit? To answer this question, we synthesized and implemented the above 53 benchmarks using the standard Vivado flow, and without any special constraints applied. This was done to demonstrate what would have happened if neither the Physical nor Functional Assurance methods were applied. These designs serve as a baseline comparison for the Physical and Functional Assurance methods. By comparing the Physical and Functional designs to their corresponding baseline designs, we can observe what kind of effects the assurance approaches have on circuit performance, specifically circuit speed and size.

CHAPTER 6. PHYSICAL ASSURANCE

With the theoretical framework established and a large suite of test benchmarks ready to use, the next step was to find out, *is it actually feasible to implement the IP Assurance Framework?* We had two implementation approaches we wanted to try; the first approach we called *Physical Assurance*.

6.1 Implementing the IP Assurance Framework

The idea behind Physical Assurance is simple—if you can demonstrate that an instantiated IP is physically identical to a trusted, trojan-free IP, then you have assurance that your instantiated IP is also trojan-free. To actually implement the Physical Assurance approach, we followed the three steps of the IP Assurance Framework: IP creation, IP instantiation, and IP comparison. In this section, each of these three steps is described in detail.

6.1.1 Creating the IP

To implement Physical Assurance, we needed a way of creating and distributing a piece of hardware IP that could be identically reconstructed across different designs. We found that Vivado's Hierarchical Design tool, and in particular its focus on module reuse, to be a natural fit for this problem.

Vivado HD

Vivado Hierarchical Design, or Vivado HD, is a special design flow provided by Xilinx that enables users to "partition a design into smaller, more manageable modules to be processed independently" [32]. Vivado HD allows users to create and use design modules *out-of-context*, *i.e.* removed from the context of a top-level design. This enables independent module analysis

and reuse by allowing users to create and examine modules in an environment removed from the influences of an external or top-level design.

Vivado HD is largely centered around the use of *physical blocks*, or *pblocks*. A pblock is a Xilinx design construct represented as a physical area within the FPGA, typically specified as a rectangular region within the FPGA fabric, and is defined by the range of CLBs it encompasses. Pblocks are typically used in FPGA floorplanning, and can be used to constrain placement and routing within an FPGA. However, pblocks, when used with Vivado HD, can also be used to build and design modules independently of one another. Physical Assurance relies on this second usage of pblocks: IP is developed and distributed as pblocks using Hierarchical Design.

To create pblock-based IP, the trusted vendor follows the steps outlined in [33]. This process produces a pblock saved as a Design Checkpoint File (.dcp), or DCP file. DCP files are Vivado's way of storing the results of various steps of the design flow, such as synthesis or implementation (and are not unique to the Vivado HD flow). In Physical Assurance, the generated DCP file contains all of the information needed to recreate the pblock with the IP core inside, such as the configuration, placement, and routing of all of the IP's cells.

Creating a Gold Standard

Using Vivado HD, the trusted vendor creates a DCP file containing a fully synthesized and implemented IP core, created independently from the influence of any top-level design. It is at this point that the trusted vendor uses the hardware trojan detection techniques described in Section 3.2 to verify, as best as possible, that the created IP is free of hardware trojans. This thoroughly vetted piece of IP becomes the gold standard version of this IP, and will be the version of the IP that all future instantiations of the IP will compare themselves to. To be precise, the gold standard is the vetted DCP file itself. An example of a pblock-based IP can be found in Figure 6.1.

One Extra Detail...

Once the trusted vendor creates a trusted IP, the user can use Vivado HD to integrate the IP into a custom design. Technically, this DCP file is all the user needs to instantiate the IP. However, once we started running our experiments, we noticed an unexpected peculiarity of Vivado HD.

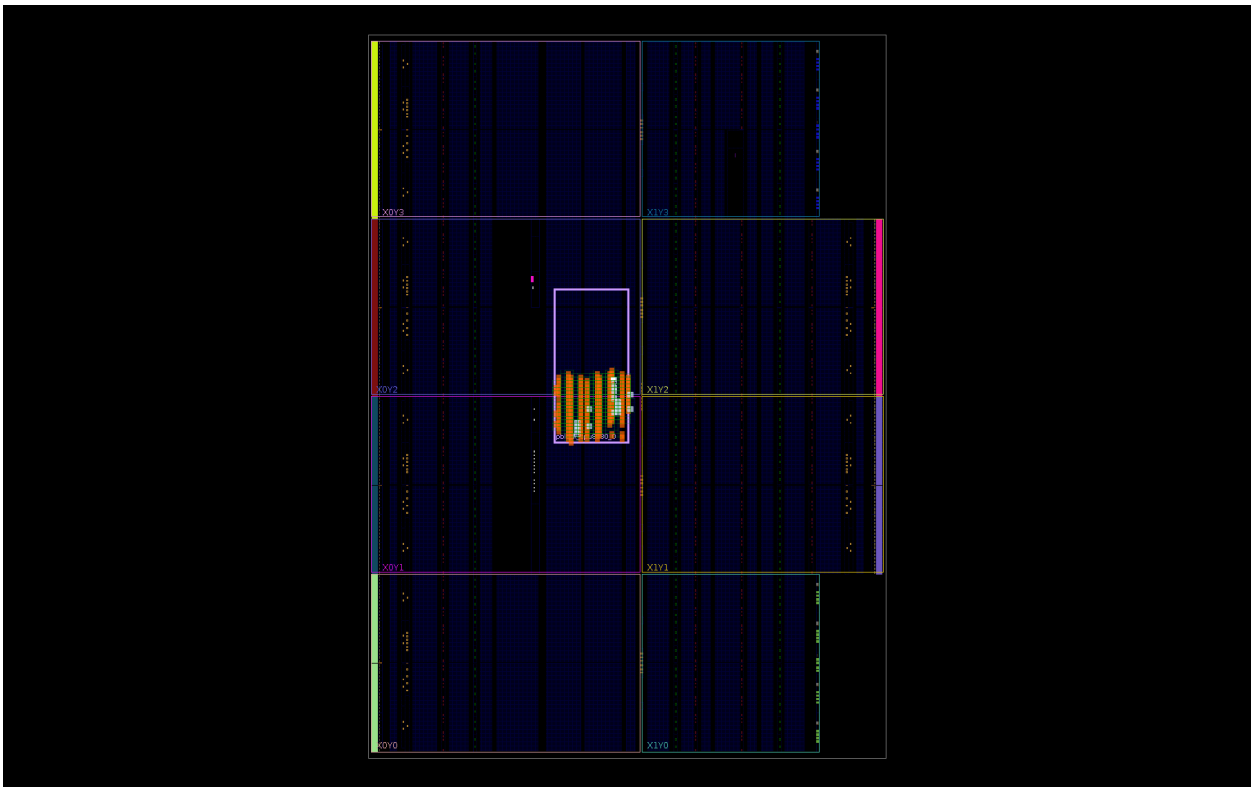

Figure 6.1: An IP core inside a pblock. The pblock is the magenta box in the center of the FPGA, while the IP itself is highlighted in orange. The small, white boxes inside the pblock are *partition pins*. These partition pins do not correspond to anything physical on the FPGA, but instead are abstract "hooks" that Vivado uses when connecting the pblock to an external design (note that in Figure 6.2, where the pblock has been integrated into a user's design, the partition pins disappear).

In short, Vivado was occasionally scrambling the names of the IP's inputs. This was causing our equivalence comparisons described in Section 6.1.3 to fail. To solve this problem, we provided Vivado HD with an extra set of constraints to use when integrating the IP into the user's design. These constraints were stored in a Xilinx Design Constraints (XDC) file (details of this XDC file, what it contains, and how it is created is described in Section 6.4.1). Once both the DCP *and* XDC files have been created, the trusted vendor passes both files over to the user. The IP is now ready to be instantiated.

6.1.2 Instantiating the IP

Once the user has both the DCP and XDC files, they can instantiate the IP into their custom design. This can be done using the process described in [33]. This process takes the gold standard

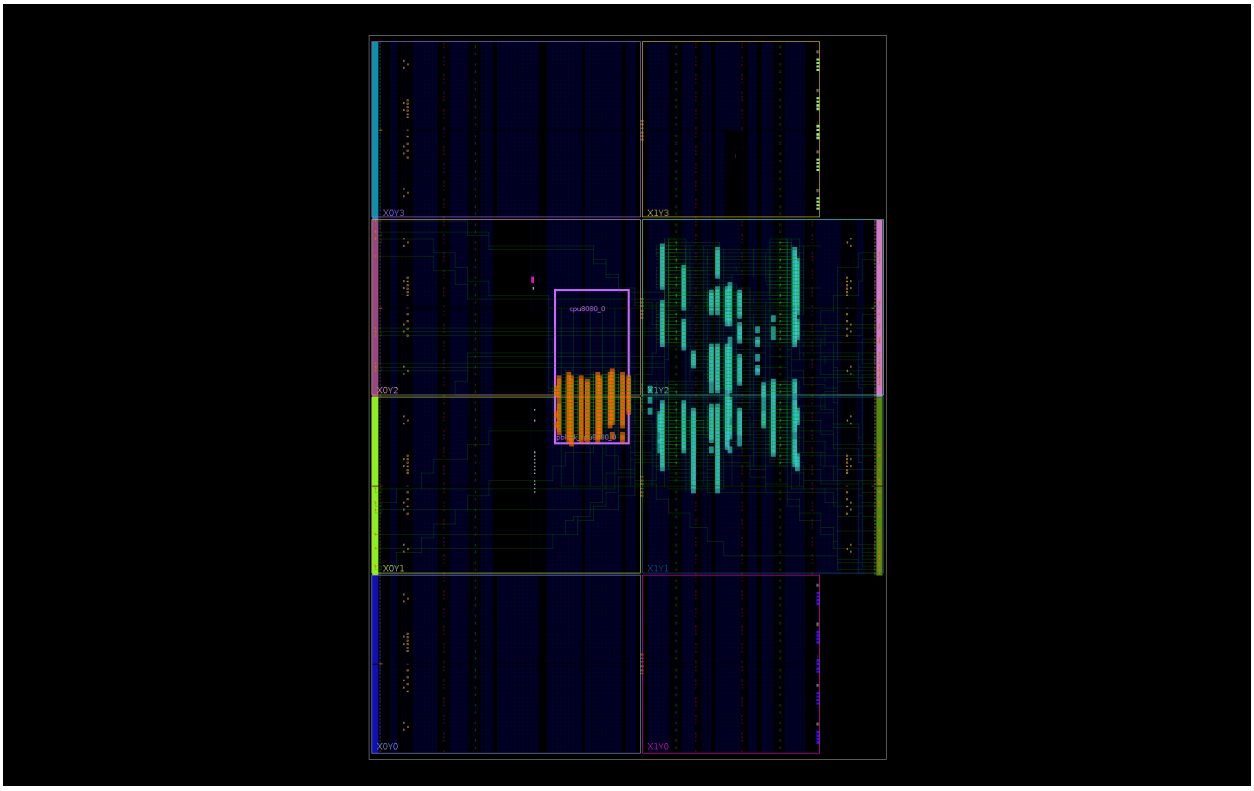

Figure 6.2: A pblock integrated into a user's design. This pblock is the same pblock as seen in Figure 6.1. Note that the IP in the original pblock and IP in this instantiated pblock look the same. This is because both IPs have the exact same placement and routing.

IP and places an identical copy of the IP into the user's design, and at the exact same location within the chip as in the original gold standard IP. It is worth noting that we had to slightly modify the provided scripts to include the additional XDC constraints file in the instantiation process. An example of a pblock-based IP instantiated into a user design can be seen in Figure 6.2.

6.1.3 Comparing the IP

At this point, there are now two versions of the IP: The first is the trusted, gold standard model of the IP created by the trusted vendor. The second version is the version that now resides in the user's design. This instantiated copy of the IP is the version that the user doesn't trust—it could have been tampered by a malicious attacker or by malicious CAD tools, for example. To verify that the instantiated IP is as safe to use as the trusted IP, the user must compare the two and determine some kind of equivalency.

In the Physical Assurance approach, the goal is to determine if the instantiated IP is an exact physical replica of the trusted IP. In other words, Physical Assurance asks, Are all the IP's cells in the exact same location? Are they configured the same way? Are all the nets routed exactly the same way? To answer these questions, we wrote a script which extracts all relevant design information from each design [1]. Specifically, the script iterates through the pblock's CLBs and extracts the following:

- *Sites*: All sites contained in the pblock
- *Cells*: All cells of the sites, as well as all cellpins, nets, and properties of the cells
- *Nets*: All nets of the cells, what type of net (boundary vs. non-boundary) they are, as well as all wires and PIPs (Programmable Interconnect Points) contained in the nets, and all net properties
- *BELs*: All BELs (Basic Elements) of all the pblock's cells, including their properties

As the script iterates through the CLBs, it writes the physical data listed above to a file. This process is done twice—once for the trusted IP's pblock, and once for the instantiated IP's pblock. With all of the IP pblocks' physical information bijectively mapped to a file, it becomes easy to compare physical information: Determining equivalence between pblocks is as simple as comparing the two generated files. We used the program `diff` to do all of our file comparisons. Another possible method is to hash the contents of the two files and compare hash values.

6.2 Experiments

With all the steps of the IP Assurance Framework implemented, it was time to test Physical Assurance. Experiments were conducted to answer two questions: 1) Can Physical Assurance determine equivalence between a trusted IP and an untampered, instantiated IP? And 2) Can Physical Assurance detect when an IP has been tampered with? These experiments are discussed in the following subsections.

[1] This script, `extractDesignInfo.tcl`, can be found at `https://github.com/byuccl/ipassurance`

6.2.1 Determining Equivalence

Using the program `diff`, we verified that all 53 of the instantiated IP blocks were identical to their respective trusted IP blocks. This implies that our Physical Assurance method does not readily yield false negatives, *i.e.*, it does not falsely report nonequivalent IP blocks.

6.2.2 Detecting Tampered Designs

To successfully implement the IP Assurance Framework, the approach also needs to successfully detect unwanted modifications to the user's IP. To test this, we made the following modifications to our IP blocks:

- *Intruding cells*: This modification takes a random cell from the IP's surrounding circuit, outside of the pblock, and places it inside the pblock.
- *Changing cell locations*: In this modification, a random cell belonging to the IP and residing within the pblock is moved to a different location in the pblock.
- *Changing the logic*: This modification randomly selects a LUT in the IP and inverts its `INIT` property, which represents the LUTs initialized logic equation.
- *Changing the routing*: This modification selects a random net within the IP and changes its path through the pblock. In this modification, the source and sink of the net remain the same; only the route of the net changes.

Each of these four modifications was independently applied to each of the 53 designs, for a total of 212 "tampered" designs. All 212 of these modifications were caught by our Physical Assurance approach. Based on the observed sensitivity of Physical Assurance, it is clear that *any* modifications to the original, trusted IP will be noticed. Thus we have demonstrated that the Physical Assurance approach can successfully distinguish between tampered and untampered designs, making it a powerful and proven tool in hardware trojan mitigation.

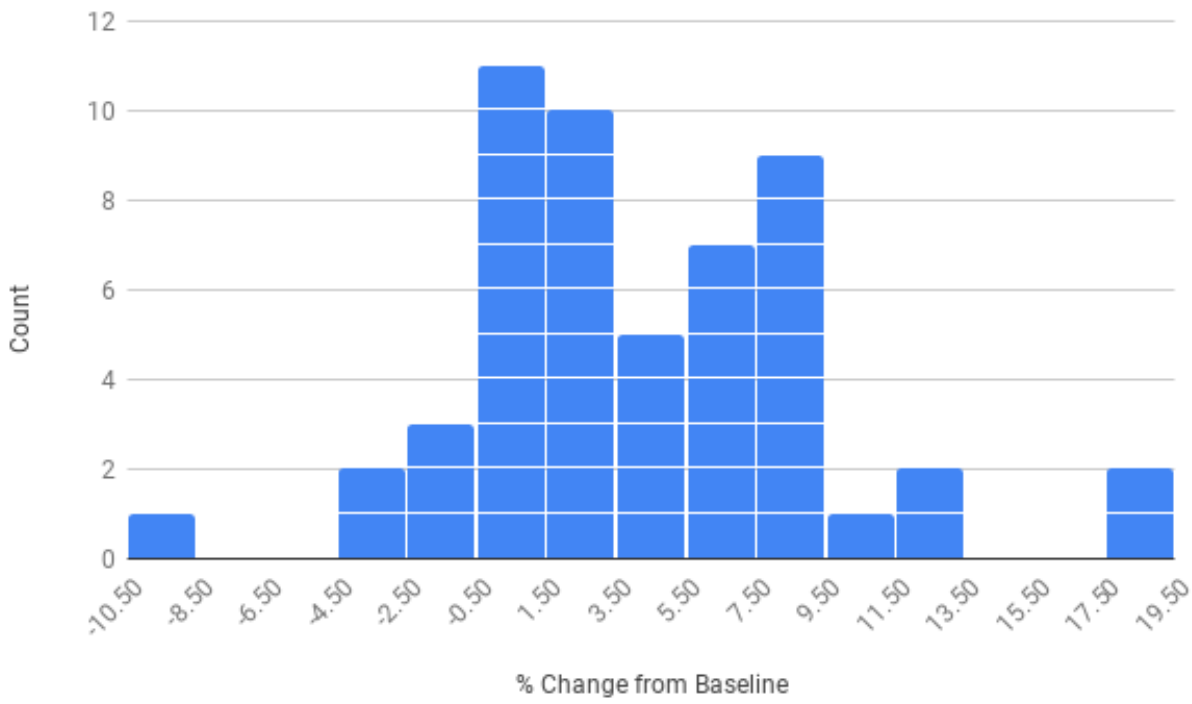

Figure 6.3: Histogram showing the percent increase or decrease in slices used between baseline designs and corresponding post-Physical Assurance designs.

6.3 Performance Impacts

What does a user have to "pay", in terms of performance, to implement Physical Assurance? To answer this question, we looked at two common performance metrics, area and maximum operating frequency.

6.3.1 Area

To measure Physical Assurance's effect of circuit area, we compared the number of slices used in the baseline design to the number of slices used in the Physical Assurance-based design. Results from this comparison can be seen in Figure 6.3. On average, designs that used the Physical Assurance method used 4.73% more slices than the corresponding baseline design. Likewise, the median design was 3.39% larger than the baseline design. This is not unexpected; by constraining an IP to a pblock, Vivado has less freedom to optimize the design, resulting in, on average, a larger design.

Some of our datapoints, however, show that Physical Assurance doesn't always result in a larger circuit. Of the 53 benchmarks, 6 were actually *smaller* than their corresponding baseline designs, with the smallest being the `basicrsa` design, which used 9.43% *fewer* slices than its corresponding baseline design. Why would some Physical Assurance-based designs be smaller than their corresponding baseline designs, despite the additional constraints placed on the Vivado Implementation tool? The answer is likely due to the complexity and stochasticity inherent in Vivado[2]. For example, Vivado's Implementation tool relies on randomized algorithms to place and route designs. Changing these algorithms' random seed will likely result in designs of different shapes and sizes across different runs, even when design sources and Vivado configurations remain the same. Thus in our dataset, it is not too surprising that some Physical Assurance-based designs ended up smaller than their baseline comparisons; the peculiarity is likely a result of the stochastic nature of Vivado's tools.

6.3.2 Operating Frequency

Another potential metric for measuring Physical Assurance's impact on circuit performance is maximum operating frequency, or f_{max}. To measure Physical Assurance's effect on speed, we could compare the f_{max} of a baseline design to f_{max} of its corresponding pblock-based design, similar to how differences in area were measured in Section 6.3.2. However, we decided that such a comparison would be misleading, largely due to the way Vivado optimizes its designs.

The Problem with Comparing Frequency

When Vivado runs place & route, one of the constraints it works with is the user-specified period. This period tells Vivado "how fast" the user would like the circuit to be, or more precisely, the user's shortest tolerable time period between clock ticks. In the synthetic benchmarks, this time period was set to 10 ns, while in the LEON3 benchmark, it was set to 20 ns. From the period, it is inferred that the minimal tolerable frequency is 100 MHz for synthetic designs, and 50 MHz for the LEON3 designs.

[2]Note this is not unique to Vivado; Other FPGA CAD tools have the same degree of complexity and stochasticity.

What happens if Vivado creates a circuit that can easily run as fast as this minimal tolerable frequency? In many cases, Vivado *could* continue to optimize the circuit to try to achieve a higher maximum operating frequency. However, we believe that once Vivado achieves the user's defined time constraint, it doesn't work any harder to further optimize the frequency; it has already reached the finish line, so to speak. This means that in a circuit that meets the user's timing constraints, f_{max} does not represent the *theoretical* maximum operating frequency of the design, but rather the maximum operating frequency of *that particular implementation of the design*.

This causes trouble when comparing different values of f_{max} between baseline and Physical Assurance-based designs. For example, the baseline `aes128` design has a maximum operating frequency of 143 MHz, while the corresponding Physical Assurance-based `aes128` design has a maximum operating frequency of only 135 MHz. Does this mean that using Physical Assurance caused the `aes128` design to slow down? Not necessarily. Both designs are well above the minimal tolerable frequency of 100 MHz, implying that Vivado likely *could* have made both designs faster. Then the 143 MHz and 135 MHz do not represent the theoretical fastest possible implementation of each respective design. They simply represent the frequency that Vivado achieved before "giving up", because it had already reached its goal. Therefore, it makes no sense to compare frequencies that are higher than the minimal tolerable frequency as a means to measure Physical Assurance's effects on timing.

Sweeping the Target Period

If we could compare circuits' theoretical f_{max}, we could then measure Physical Assurance effects of timing by comparing frequencies. How can a circuit's optimal f_{max} be found? One way to get a good approximation of the theoretical f_{max} is to sweep the target period. In a sweep, the timing constraint is repeatedly made smaller and smaller until Vivado fails to meet timing. The point at which Vivado fails to meet timing is a good measure of the fastest we can reasonably expect a design to be, and can give us a good estimate of the highest theoretical f_{max}. We could then compare these highest theoretical f_{max} values, and make informed conclusions on Physical Assurance's effect on timing. However, conducting this sweeping method is very time-consuming, and is outside the scope of this thesis. So for now, we must make informed decisions based on the

data we have, i.e. the f_{max} of our designs when the target period was set to 10 or 20 ns (depending on the type of design).

Passing vs. Failing Timing

What are useful ways to interpret the data? One way is to look for cases where a baseline design passed timing, and the corresponding Physical Assurance design failed timing. Such cases would suggest a relationship between Physical Assurance and timing—this would mean that with all variables held constant[3], the design failed timing only when Physical Assurance was applied.

In our 53 benchmark designs, only one design (`dfadd`) failed to meet timing when implemented using Physical Assurance. This suggests that Physical Assurance may indeed have a negative effect on timing. Conversely, no baseline designs that failed to meet timing subsequently succeeded in meeting timing when implemented using Physical Assurance.

Comparing Failing Designs

Another way to measure Physical Assurance's effect on timing is to compare designs where both the baseline and Physical Assurance-based designs failed to meet timing. In these situations, it's reasonable to suggest that the f_{max} for each design represents something close to the best Vivado could have reasonably achieved. If this is the case, it is fair to compare baseline and Physical Assurance f_{max} values.

Of the 53 benchmarks, 23 failed to meet timing across both baseline and Physical Assurance designs. The average failed baseline design was 1.72 times slower when implemented using Physical Assurance. However, this data is skewed by two large outliers, `amber` and `des3_area`. With these two outliers removed, the average design was only 1.11 times slower when implemented using Physical Assurance. Both averages suggest that, if the f_{max} of designs that failed to meet timing is reasonably close to the theoretical highest f_{max}, then Physical Assurance has a mildly negative effect on the timing of a circuit, with some circuits experiencing severe negative effects on timing.

[3] except for perhaps the random seed

6.4 Challenges and Solutions

In theory, Vivado HD (Hierarchical Design) should be able to instantiate identical copies of a pblock. In practice, getting Vivado HD to reproduce an identical copy of a pblock wasn't as straightforward as we expected, and sometimes required special tricks or workarounds. While these challenges are often minor in their nature, they are important to note, as they highlight that implementing Physical Assurance sometimes requires special tricks to get it to work. This section discusses these various challenges we encountered while testing the Physical Assurance method, as well the possible solutions we developed.

6.4.1 Cellpin to Belpin Mapping

In Hierarchical Design, Vivado preserves internal routes, but not interface nets [32]. This means that when boundary LUTs have multiple interface net inputs, Vivado will sometimes permute the inputs, resulting in a permuted LUT equation, and therefore physically inequivalent IPs. To address this issue, we wrote a script that would extract the mapping between the logical-level cell pins (cellpins) and physical-level BEL pins (belpins) of boundary LUTs in the trusted IP, and then constrain the instantiated IP to preserve this cellpin to belpin mapping. This process allowed us to instantiate physically identical boundary LUTs, and therefore instantiate physically identical IPs.

6.4.2 BUFG Issues

One of our trusted IP cores, `ahbjtag`, had a global buffer (BUFG) in it, which caused problems when we tried to instantiate it into a user design. While the Vivado Hierarchical Design Guide states that global buffers are supported in out-of-context modules [32], we were unable to make this happen [34]. We found that we could avoid this problem by moving the BUFG outside of the trusted IP, and connecting it to the trusted IP via interface nets [1].

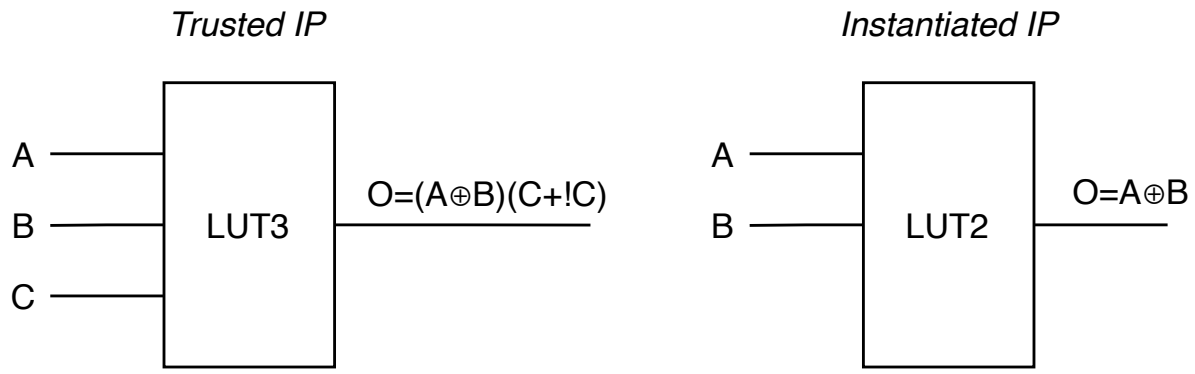

Figure 6.4: A peculiarity encountered when comparing the instantiated and trusted `md5` IPs. Note the extra term on the trusted IP—it has no effect on the LUT's functional output, yet its existence causes Physical Assurance to fail. Reproduced from [1].

6.4.3 Useless Inputs

The `md5` benchmark uncovered a peculiarity of Vivado, shown in Figure 6.4. In this scenario, the trusted IP feeds the input `const1` into a LUT, where its value is rendered useless. In the corresponding instantiated IP, this useless input is removed. Whether the useless input was intentionally included for some unknown reason or whether it represents a bug in Vivado is unclear. Regardless, it caused issues when we tried to compare trusted and instantiated IPs: The removed input meant that the instantiated IP was simply not physically identical to the trusted IP. It might be possible to tweak Vivado in such a way that precludes this irregularity from happening; However, we found an easier fix: By simply moving the location of the pblock, this problem went away on its own.

6.4.4 A Bug in Vivado 2016.2

We encountered an bug in Vivado HD 2016.2 that caused our comparison step to fail. In the `potato` benchmark, the trusted IP contained an extra cell that the instantiated IP did not. This cell was a 1-input LUT programmed to be a simple buffer. This buffer had no effect on the functionality of the design, but it *did* cause the comparison step to fail—the extra cell in the trusted IP meant that the two IPs were not physically identical. Why did the trusted IP have this extra cell in the first place? We found the extra cell was a result of a bug in Vivado 2016.2 [35]. We were able to avoid this bug by upgrading our tools to Vivado 2017.4 [1].

6.4.5 Routing Congestion

In some designs, the location or size of the pblock within the user's design would cause Vivado to fail routing. This is likely due to congested routing in the user's design's surrounding circuitry, caused by the additional constraints placed on the design by the pblock. We found that we could fix this problem by changing the size or location of the pblock in both the trusted and instantiated IPs.

6.5 Limitations

The tests described in Section 6.2 demonstrate that Physical Assurance is a working implementation of the IP Assurance Framework. Using Physical Assurance, IP users can quickly verify that their IPs are physically identical to a trusted, gold standard IP. This alone makes Physical Assurance a useful tool in hardware trojan mitigation. However, this approach, while successful, is not without drawbacks. Most significantly, it forces users to instantiate their IP inside of a pblock. This has two main effects: First, the user must instantiate their IP in the exact same location as the trusted IP. This may cause trouble for the user, especially if their design already has tight constraints on placement, timing, or routing. Second, the user can't make any optimizations to IP. Again, this may be a burden for a user working under tight design constraints.

CHAPTER 7. FUNCTIONAL ASSURANCE

The Physical Assurance method is a successful implementation of the IP Assurance Framework—it provides users with a method of verifying that their IP has not been tampered with. However, it also comes at the high cost of requiring users to implement IP as a pblock. It was this rigidity of Physical Assurance that led us to create a second implementation of the IP Assurance Framework, an implementation without such steep trade-offs. This second implementation we called *Functional Assurance*.

7.1 Implementing the IP Assurance Framework

Functional Assurance aims to provide the same assurance as Physical Assurance, but without the demanding constraints on placement and routing. The goal with Functional Assurance was to allow the user to place and route their IP however they wish. This meant that in order to determine equivalency, Functional Assurance would have to compare something other than physical placement and routing. Instead, Functional Assurance compares the *behavior* of the IP, *i.e.* its functionality.

As with Physical Assurance, the Functional Assurance approach consists of three parts—IP creation, IP instantiation, and IP comparison. Each of these is discussed in detail in the following sections.

7.1.1 Creating the IP

In Functional Assurance, IP is created in the same way as in Physical Assurance. That is, Functional Assurance uses Vivado HD to create IP as a pblock in an out-of-context manner. This allows the IP to be created without the influence of any top-level design, and therefore gives the IP maximum re-usability. An example of IP created inside a pblock can be seen in Figure 7.1.

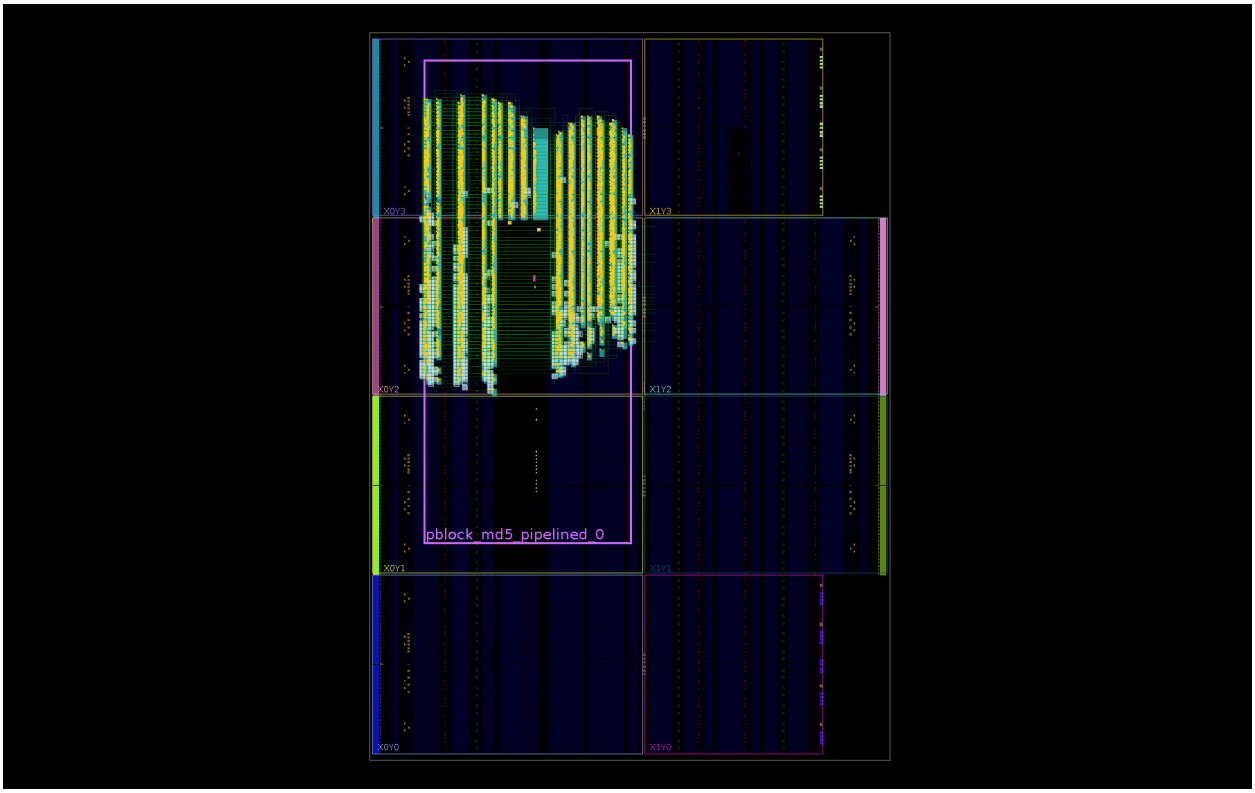

Figure 7.1: A IP block (`md5_pipelined_0`) built inside a pblock. The contents of the IP are highlighted in gold. This IP represents the trusted, gold standard version of the IP.

Functional Assurance differs from Physical Assurance in what it actually provides to the user. Unlike Physical Assurance, Functional Assurance needs no information regarding placement or routing of the IP. This means that the trusted vendor can simply provide the IP to the user in the form of a netlist [1]. This netlist can be exported as a Verilog file or as an EDIF file; For this work, we decided to use Verilog netlists.

Once the netlist has been generated, the trusted vendor then must verify, to the best of their abilities, that the generated netlist 1) works as specified, and 2) is free of hardware trojans. Hardware trojan detection will use the techniques mentioned in Chapter 3, among others. Once the trusted vendor has verified that the IP works properly and is safe to use, the vendor can then provide the IP to the user.

[1] Creating a netlist from a pblock-based IP is straightforward in Vivado. Just use the `export_netlist` command, or export a netlist using the Vivado GUI.

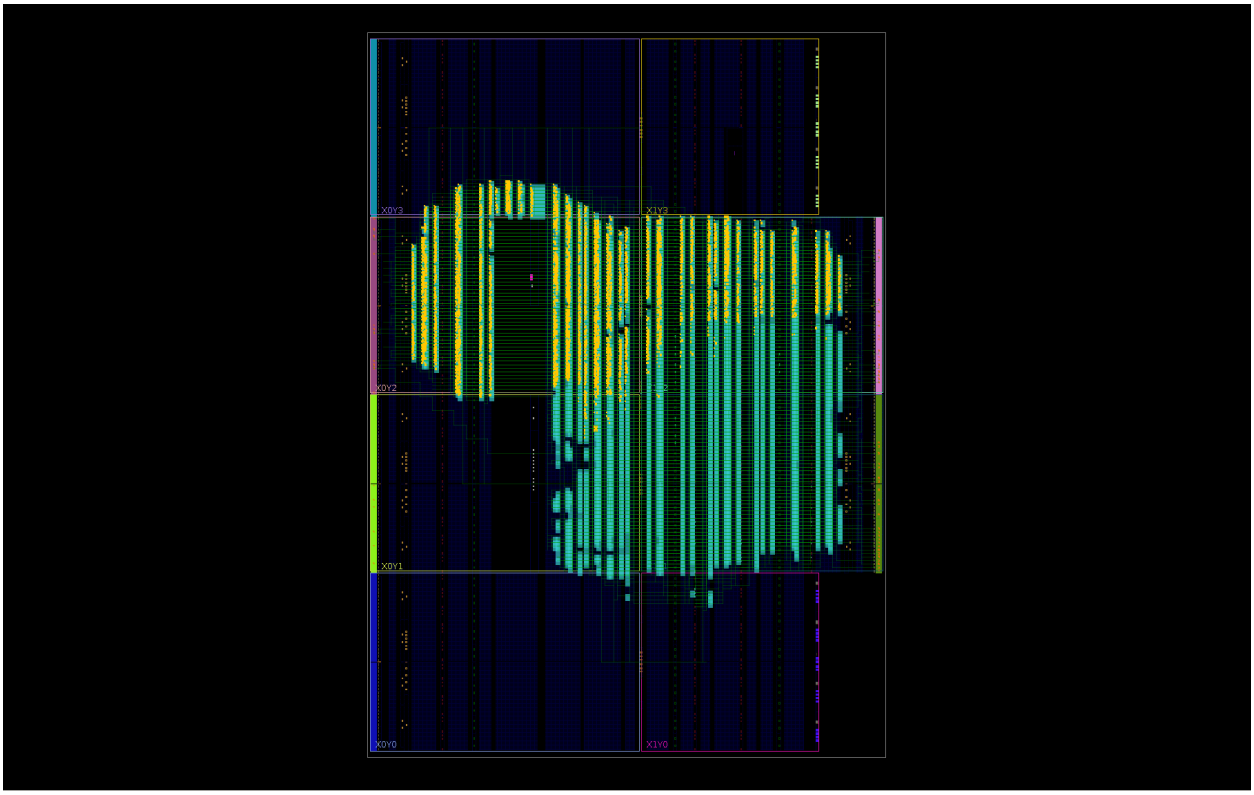

Figure 7.2: An IP block instantiated into a user's design. The contents of the IP are highlighted in gold. Unlike Physical Assurance, the cells of the IP can be placed wherever the Vivado Implementation tool deems best. Likewise, the Vivado Implementation tool is also free to make decisions regarding the IP's routing. This can result in smaller, faster designs. Custom placing and routing is also permitted in the Functional Assurance approach.

7.1.2 Instantiating the IP

Using Verilog netlists (instead of EDIF netlists) makes the IP instantiation process very easy: The netlist describes the IP as a Verilog module, meaning that the entire IP can be instantiated the exact same way as any other Verilog module.

While instantiating the IP is straightforward and easy, the Functional Assurance approach requires the user to complete one additional step. When we were conducting this work, we ran into problems with Vivado's optimizer tool. The optimizer would tweak and transform the instantiated IP in such a way that made it difficult to demonstrate functional equivalency between the trusted and instantiated IPs.

One way to address this problem is to turn off all optimizations. This would certainly prevent the IP from being optimized, although at a very high cost! Removing all optimizations

from an FPGA design would cause the design to be much larger and run much slower. I decided that in order to make Functional Assurance a reasonable solution to the IP assurance problem, I needed to be able to determine equivalency between trusted and instantiated IPs without turning off the optimizer. I found a solution to this problem in the Xilinx Vivado attribute, DONT_TOUCH.

DONT_TOUCH

DONT_TOUCH is a property of cells and nets in the Vivado tool set. The DONT_TOUCH property prevents the Vivado synthesis and implementation tools from performing optimizations on the selected cells or nets. By applying DONT_TOUCH to the cells and nets of the IP, Vivado can run synthesis and implementation with the optimizer on, performing full optimizations on the full FPGA design *except* for the cells and nets belonging to the target IP. The resulting design is optimized, except for the IP block. Note that the IP block will likely have already been optimized by the IP vendor; The DONT_TOUCH constraint only prohibits *further*[2] optimizations to the IP block.

The DONT_TOUCH property can easily be applied to an IP block in a design's XDC constraints file. For example, in a design with an IP block named aes128_0, the DONT_TOUCH property can be applied to the IP block by adding the following TCL lines to the design's XDC file:

```
1 set_property DONT_TOUCH true [get_cells aes128_0]
2 set_property DONT_TOUCH true [get_cells aes128_0/*]
3 set_property DONT_TOUCH true [get_nets aes128_0/*]
4 ...
```

Listing 7.1: Applying DONT_TOUCH to an IP block named aes128_0

7.1.3 Comparing the IP

The last step in the Functional Assurance approach is the comparison between the trusted and instantiated IPs. Unlike Physical Assurance, which relies on a physical-level comparison of IPs, Functional Assurance relies on a behavioral-level comparison of IPs. I found that I could achieve such comparisons using the Cadence Conformal tool suite.

[2]These further optimizations are almost exclusively cross-boundary optimizations.

Cadence Conformal [36] contains a Logic Equivalence Checker, or LEC. It is used primarily in the EDA (Electronic Design Automation) industry to help test and verify digital hardware designs. Its primary feature is the ability to compare gate-level "golden" designs to "revised" designs. Typically, this feature is used by digital designers, who want to verify that a design remains the same after it has been modified to add functionality, meet timing, *et cetera*. However, I soon recognized that Conformal could also be used to compare "trusted" IPs to "instantiated" IPs.

Conformal operates by comparing netlists. Thus to compare a trusted IP to an instantiated IP, we had to provide Conformal with a trusted IP's netlist and an instantiated IP's netlist [3]. We also provided Conformal with a library of Xilinx primitives[4] as well as the names of the inputs and outputs of the IP. Because doing this manually for each test design would be tedious, we scripted the process—Instead of relying on the Conformal GUI, we wrote TCL scripts (called dofiles) to do the work for us. Additionally, we were able to automate the creation of the scripts as well. The script-generating script can be found on GitHub (`www.github.com/byuccl/ipassurance`).

7.2 Experiments

Could Functional Assurance distinguish between tampered and untampered designs? This section answers this question in two parts: 1) Determining equivalence between untampered designs, and 2) Detecting modifications in tampered designs. This section addresses these two questions in the following subsections.

7.2.1 Determining Equivalence

On all 53 designs, Conformal reported equivalence between trusted IPs and the corresponding instantiated IPs. This suggests that Functional Assurance can be used to verify that a user's IP

[3]The means by which we extracted the instantiated IP's netlist from the user's design was simple, but not obvious, and is worth mentioning here. When using Vivado to export a netlist, Vivado gives you the whole design—you can't tell Vivado to export only a submodule of a design. However, to successfully compare to the trusted netlist, we want only the parts of the user's design that correspond to the IP. Conformal allowed us to select which module (or submodule) of a design we wanted to be the "top" module. Because our approach instantiates IP as Verilog netlists, the IP was already instantiated as a module, and could easily be selected to act as the top module and be compared to the trusted netlist.

[4]Because Vivado netlists are in terms of Xilinx primitives (see Chapter 2), Conformal needed to know what exactly these primitives are and how they behave. Thus we had to provide Conformal with a library of Xilinx primitives expressed as Verilog code, which could then synthesized into logic gates by Conformal.

has not been tampered with. To get to our 100% pass rate, though, we had to do some tweaking and tuning of our system, which is described in Section 7.4.

7.2.2 Detecting Tampering

As with Physical Assurance, we needed to verify that the Functional Assurance approach could actually detect modifications. To simulate tampering, we made the following modifications to each of our 53 benchmark designs:

- *Changing the logic*: In each design, a random LUT was selected, and modified. Specifically, the `INIT` property of the LUT, which represents the LUT's initialized logic equation, was inverted.
- *Adding a backdoor*: In each design, an extra port was added to the design. The added port was wired to a random internal signal in the IP. This modification was made to simulate a secret, eavesdropping backdoor.

Each of the two mentioned modifications was applied individually to each benchmark, for a total of 106 modified design. All 106 of these modifications were detected by Conformal, which reported that the instantiated IP was `NONEQ` (non-equivalent) to the trusted, golden-model IP. This result, along with the results in Section 7.2.1, suggests that Functional Assurance can successfully distinguish between tampered and untampered designs.

7.3 Performance Impacts

How does the Functional Assurance approach impact the performance of a design? To answer this question, we compared the baseline designs (discussed in Chapter 5) and the Functional Assurance-based designs using two standard performance metrics: Area, and maximum operating frequency.

7.3.1 Area

To measure Functional Assurance's effect on circuit area, we compared slice count, *i.e.* how many slices a design utilized. Results from this comparison can be seen in Figure 7.3. On av-

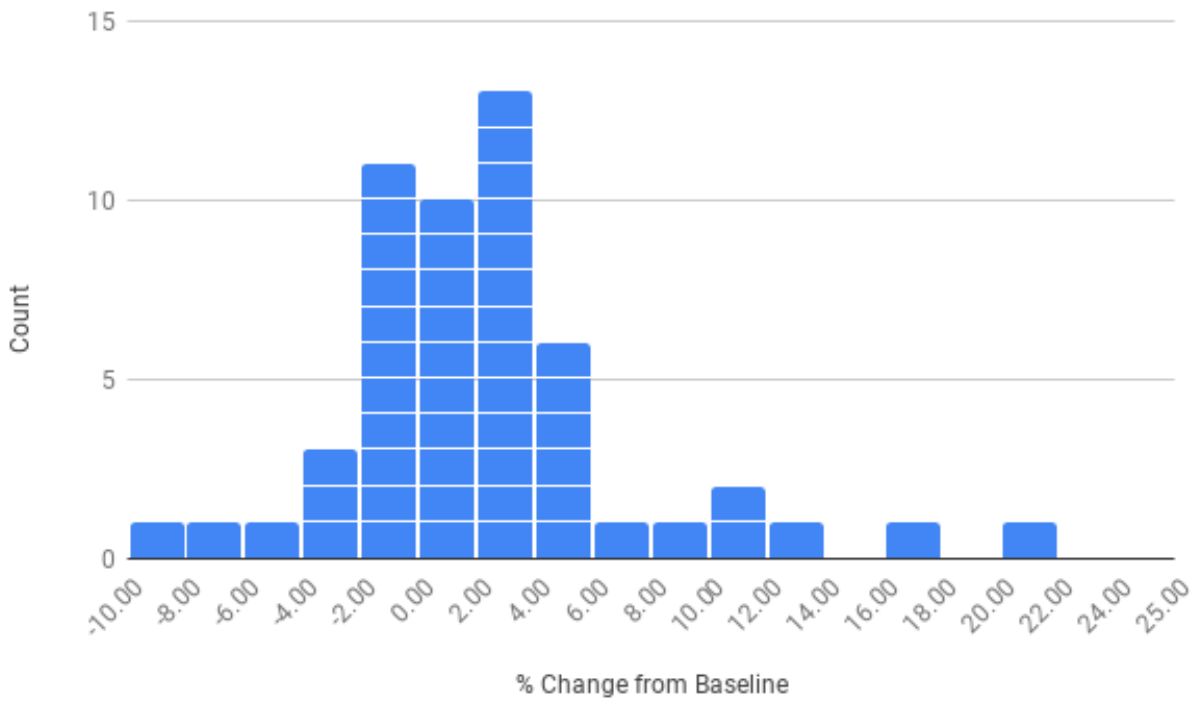

Figure 7.3: Histogram showing the percent increase or decrease in slices used between baseline designs and corresponding post-Functional Assurance designs.

erage, designs that used the Functional Assurance method used 2.81% more slices, with a standard deviation of 0.06. The median increase was 2.00%. Thus in most situations, Functional Assurance produces slightly larger circuits. This result is expected—the `DONT_TOUCH` constraints will naturally prevent some optimizations, resulting in a larger design.

Some outliers are worth noting. In particular, the worst effects of the Functional Assurance approached was observed in the `sudoku` design, which used 27.73% more slices than its corresponding baseline design. Conversely, the `msp430_vhdl` design used 7.63% *less* than its baseline design. Although unexpected, this is not altogether unexplainable—the stochastic nature of CAD tools will inevitably produce circuits of varying sizes, regardless of any assurance approach; In the case of `msp430_vhdl` and a few other designs, the CAD tools likely just got "luckier" when placing the design.

7.3.2 Operating Frequency

As discussed in Section 6.3.2, it's difficult to determine exactly how an approach such as Functional Assurance effects maximum operating frequency. Chapter 6 concluded that the only fair way to determine if the implementation approach has impacted maximum operating frequency is to observe which assurance-approach-based designs (if any) failed to meet timing at a specified clock rate after the corresponding baseline benchmark succeeded in meeting timing at the same specified clock rate. With this as our metric, Functional Assurance had no observable effect on timing. That is, no design which met timing when implemented as a baseline benchmark then failed to meet timing when implemented using the Functional Assurance approach.

7.4 Challenges

The process of creating, instantiating, and comparing IP using Functional Assurance is a working and satisfactory implementation of the IP Assurance Framework, according to our results. However, while the process is fairly straightforward and automated, some minor challenges still exist. Most significant is the issue of *port renaming*. In rare and unexplained situations, Vivado will sometimes rename ports in the instantiated IP. For example, the port `\uarti[rxd]` on the out-of-context `ahbuart` module is renamed to `.\uarti[rxd] (uarti)` in the instantiated version of the `ahbuart` design (see Listings 7.2 and 7.3). This causes problems during equivalence checking: Conformal, which relies on names to map key points, isn't able to resolve the name difference. As a result, Conformal can't properly map the renamed inputs to each other, and finds the two modules to be non-equivalent, even though they are functionally the same.

To address this issue, we wrote a Python script `sanitizeBoundaryNets.py`, which parses netlist port lists and replaces renamed ports with their original names. This technique worked on several benchmarks; However, it still does not work on about half of the LEON3 benchmarks. To compensate, we had to manually fix the renamed ports ourselves in order for these LEON3 benchmarks to pass the equivalence checks. Improving this script is discussed in further detail in Section 9.

```
module ahbuart
    (rst,
    clk,
    \uarti[rxd],
    \uarti[ctsn],
    \uarti[extclk],
    \uarto[rtsn],
    \uarto[txd],
    \uarto[scaler],
    \uarto[txen],
    ...
endmodule
```

Listing 7.2: Port list of trusted `ahbuart` IP

```
module ahbuart
    (rst,
    clk,
    .\uarti[rxd] (uarti),
    \uarti[ctsn],
    \uarti[extclk],
    \uarto[rtsn],
    .\uarto[txd] (uarto),
    \uarto[scaler],
    \uarto[txen],
    ...
endmodule
```

Listing 7.3: Port list of instantiated `ahbuart` IP

7.5 Limitations

Functional Assurance gives the user more flexibility than Physical Assurance, but still has its limitations. Primarily, in the current iteration of Functional Assurance, the user must prevent cross-boundary optimizations between the IP block and the surrounding design. Depending on performance constraints, this may be a nontrivial trade-off for IP users.

7.6 Removing the DONT_TOUCH Constraint

To see what would happen if the `DONT_TOUCH` constraint was removed, I rebuilt the benchmarks without any constraints on the IP block. After running these new benchmarks through the Functional Assurance process, only 16 out of the 53 benchmarks passed the equivalence checking step. This low result says two things: First, it is evident that a naïve implementation of Functional Assurance is likely to fail. To avoid false positives, it is best to apply the `DONT_TOUCH` constraint to preserve IP blocks, and suffer the hit to performance and area. Second, there is much room for improvement in getting Functional Assurance to pass *without* resorting to the `DONT_TOUCH` constraint. This is discussed in further detail in Chapter 9.

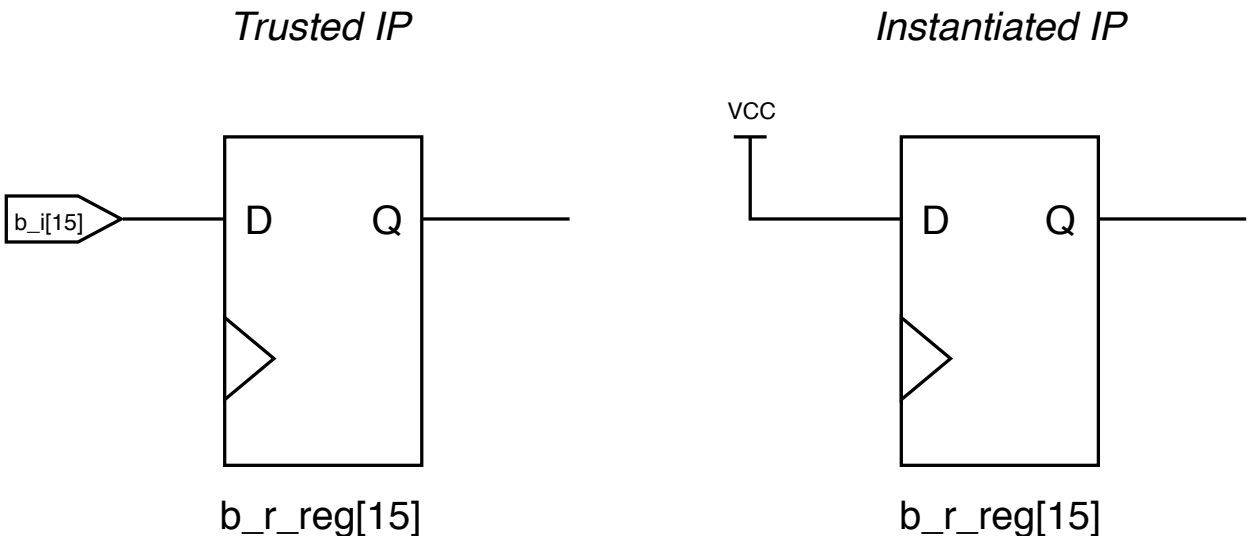

Figure 7.4: Example of an optimized input port in the `bcd_adder` benchmark. In the trusted IP, the input to `b_r_reg[15]` is supplied via an external source. However, in the instantiated version of this IP, the external source is tied to VCC. Without the `DONT_TOUCH` constraint, the input port `b_i_reg[15]` is optimized away, and the input to the instantiated IP's `b_r_reg[15]` is tied to `VCC`.

Why did removing `DONT_TOUCH` cause some of the benchmarks to fail equivalence checking? The answer is due to *cross-boundary optimizations*. When Vivado is given *a priori* information on how a module will be instantiated, it can modify and optimize the module to achieve better performance. Optimizations typically happen near an IP's inputs and outputs; both cases are discussed below.

Input Optimizations

External knowledge about how an IP block is instantiated can affect the internal contents of an IP. For example, some of the `b_i` inputs on the `bcd_adder` IP block in the `bcd_adder` benchmark are hard-coded[5]. Vivado recognizes this, and determines that the input ports with hard-coded inputs are no longer necessary, and that the signal supplied by these inputs can be replaced by `GND` and `VCC` signals (see Figure 7.4). Conformal sees this optimization as a change in functionality, and declares the trusted and instantiated IPs to be functionally nonequivalent.

A similar variant to this scenario can be found in the `lfsr_randgen` IP block. In this IP, the input `seed` is a 4-bit wide signal. In the instantiated benchmark, the IP's surrounding circuitry

[5]Specifically, the upper eight bits are tied to the hex value `0xAB`.

gives the same single wire to each of the 4 inputs of `seed`, causing all four signals in `seed` to always be equal to each other. Vivado sees this as an opportunity to optimize, and removes all `seed` signals except for `seed[2]`. Vivado then routes `seed[2]` into all places where `seed[0]`, `seed[1]`, and `seed[3]` once led. Again, Conformal only sees this optimization to be a change in functionality, and declares the trusted and instantiated IPs to be nonequivalent.

Output Optimizations

External knowledge about how an IP block is instantiated can also affect the internal contents of an IP. In some benchmarks, entire ports on an IP block were optimized away. This is not too surprising, given how some of the synthetic benchmarks were constructed; In many cases, outputs of IP blocks went unused. Vivado, in an effort to make designs smaller and faster, removes both these unused outputs and outputs' upstream logic. This logic pruning results in designs that Conformal deems to be nonequivalent. An extreme example of this is the `pic_dt` benchmark. In the `pic_dt` IP block, there were two outputs. One of the outputs (`DataBus`) was unused, while the other output (`INTR_O`) always output a 0. This caused the entire `pic_dt` IP block to be removed from the design! Because the instantiated IP block was completely removed, the gold standard `pic_dt` had no corresponding instantiated IP block to compare itself to, and the comparison is undefined.

CHAPTER 8. DISCUSSION

This chapter expands on the ideas presented in the previous chapters. Physical and Functional Assurance are compared in Section 8.1. Section 8.2 discusses potential weaknesses of the IP Assurance Framework. An argument for applying this work to beyond just IP is then presented in Section 8.2.4.

8.1 Physical vs. Functional Assurance

This thesis presented Physical and Functional Assurance, two different implementations of the IP Assurance Framework. Both approaches can quickly and easily verify if an IP had been tampered with, and are viable options for engineers looking to secure their IP. However, the two approaches have differences that are worth examining.

8.1.1 Physical vs. Logical Comparisons

The primary difference between approaches is how they make comparisons. Physical Assurance compares physical-level data about the trusted and instantiated designs, while Functional Assurance examines the netlist-level designs. This in turn affects *what is actually assured* by the two approaches. Physical Assurance tells the user if their IP has changed at the physical level, securing IP up to the post-implementation, placed and routed level of the design. In contrast, Functional Assurance deals with the logical-level design through netlists, and has no notion of a physical design. Functional Assurance can only assure an IP up to the logical level.

A naïve way to interpret this difference between approaches would be to say that Functional Assurance assures an IP up to the post-synthesis level of a design, while Physical Assurance assures up to post-implementation. Indeed, Functional Assurance requires the user to at least run Synthesis, while Physical Assurance requires a user to run Synthesis and Implementation. However, netlists can be extracted from post-implementation designs, and, as we found out,

can sometimes be slightly different than post-synthesis netlists. Functional Assurance can use these post-implementation netlists, which are "closer" to the final, placed and routed design than post-synthesis netlists. This would then be a post-implementation, logical level comparison of IPs. Therefore, Functional Assurance does not necessarily mean "post-synthesis comparison", and likewise Physical Assurance does not mean "post-implementation comparison". They are strictly logical and physical-level design comparisons, respectively. It is important to keep these distinctions in mind when discussing or using either assurance approach.

The difference between physical and logical comparisons may have security implications. It is easy to argue that Physical Assurance is objectively a more "secure" way of assuring IP—There are simply fewer attack vectors standing between a physical-level design than a logical-level design. Physical-level designs are also not susceptible to the poor placement routing attack, as discussed in Section 8.2.5.

8.1.2 Tradeoffs

Physical and Functional Assurance both rely on the ability to preserve a notion of "identicality" between a gold standard IP and a future implementation of that IP, either through the use of pblocks or the `DONT_TOUCH` constraint. However, in many cases, IP is parameterizable, meaning that the user, after receiving the IP, can configure or customize it in a way that suits their needs. This presents a challenge for both Physical and Functional Assurance: Neither approach can handle any kind of significant change in the structure of the IP, since a custom tailored IP will almost certainly not match a set gold standard. At present, this is a limitation of both assurance approaches.

Both assurance approaches also had observable effects on design size and area, as was discussed in previous chapters. Of the two approaches, Physical Assurance had a larger effect on area, with designs on average being 1.92% larger than corresponding Functional Assurance designs, and 4.73% larger than corresponding baseline designs. Physical Assurance also had an observable effect on speed, while Functional Assurance had no such observable effect. For these reasons, I conclude that Physical Assurance, with its use of constraining pblocks, places more stress on designs, resulting in larger, slower circuits.

Besides just the CAD tools, pblocks also constrain the IP user. When using Physical Assurance, an IP user must place the pblock in the exact same location as the gold standard IP. This

may be inconvenient for the IP user. A work-around to this problem would be for the IP vendor to synthesize and implement the IP pblock in many locations across a device, but in the end, the IP user is still fettered by the vendor's placement and routing. Functional Assurance has no such constraints.

Both approaches have strengths and weaknesses that must be considered when deciding between the two. Physical Assurance is likely the safer approach, but it comes at a higher cost of size, speed, and flexibility. Functional Assurance provides flexibility, but it is vulnerable to attacks that Physical Assurance is not. In general, it is a choice between security and performance. Engineers must decide which choice better suits their needs when deciding between IP assurance approaches.

8.2 Where Does the IP Assurance Framework Break Down?

Everything has its own breaking point, and the IP Assurance Framework is no exception. This section describes various scenarios in which the IP Assurance Framework could fail to protect users against hardware trojans lurking in their IP. This section also discusses how to address these various threats.

8.2.1 Root of Trust

In the field of trusted computing, there is a concept called the *root of trust*. A root of trust is a secure, trusted component (usually hardware) in a system which is capable of verifying its own integrity and can extend trust to its operating environment and even to other systems [37]. This idea that a trusted source can extend a secure environment through a chain of trust is analogous to the IP Assurance Framework. In the framework, the root of trust is the IP vendor. The entire framework rests on the integrity of the vendor. If the vendor is compromised in any way, the house of cards collapses, and the IP Assurance Framework ceases to offer any protection from hardware trojans. It goes without saying that in order for the framework to be of any benefit to IP users, the user must have absolute trust that the vendor is providing safe and secure IP.

8.2.2 You're Only as Safe as Your Vendor's Hardware Trojan Detection Techniques...

Even if your IP vendor is absolutely trusted, there is still the risk that your IP can be infected with hardware trojans. After all, the vendor can only feasibly detect hardware trojans insomuch as current detection techniques allow. For example, it might be possible to design a hardware trojan that evades all current detection techniques. If such a hardware trojan ended up in the vendor's gold standard design, there would be no way of knowing that the IP was compromised. Thus it could be possible for a user's IP to contain a hardware trojan even if the IP came from a trusted source. However, in such a scenario, *no* hardware trojan mitigation technique will help, until a sufficiently powerful detection technique is developed. Dealing with this scenario is out of the scope of this thesis.

8.2.3 What If Your Tools Are Hacked?

In this thesis, the two implementations of the IP Assurance Framework—Physical and Functional Assurance—rely on third-party tools to compare IPs. This should be of concern—after all, you shouldn't trust things that you didn't make yourself [2]. How can we deal with this problem? The solution is, unfortunately and rather obviously, to not rely on third-party tools. Some of third-party tools can be avoided easier than others. Let's look at the various third-party tools that have been used in this thesis:

- **Vivado**—used to create designs, extract physical data, and extract netlists
- **diff**—used to compare physical data files (Phyical Assurance)
- **Conformal**—used to compare netlists (Functional Assurance)

Let's first take a look at Vivado. In Chapter 1, we assumed that Vivado (like other CAD tools) could be compromised—this was one of the initial threats to secure design that motivated this work. So we should assume that designs created by Vivado could be insecure. Again, this fact is what motivated the IP Assurance Framework in the first place. But in this work, we used Vivado for more than just creating designs—we used it to tell us information *about* our designs, information which we then used to verify the security of an IP. This should raise some red flags. How do we

know that Vivado faithfully gave us the information we requested? What if Vivado hid details about a lurking hardware trojan, and spoofed the physical data or netlist it provided? One way to verify that Vivado faithfully retrieved design information is to reverse engineer the bitstream. This is by no means an easy task, but research suggests it is possible [38]. A fully reverse-engineered bitstream would contain all design information (allowing users to verify Physical Assurance) and could be used to reconstruct netlists (thereby allowing users to also verify Functional Assurance). The initial cost of reverse engineering a device's bitstream would be high, but after this initial effort, reconstructing design information and netlists could be automated.

Another step that must be secured is the IP comparison step. Although unlikely, it is possible that a corrupted diff or Conformal could spoof comparison results, such as by declaring two IPs to be identical when one contains a hardware trojan. Again, the only way to *really* have complete trust in your process is to eliminate these third-party tools, and do the comparison yourself. Of the two tools used to compare IP (diff and Conformal), diff would be the easiest third-party tool to eliminate. If you didn't trust it, it would be simple enough to write a similar program that compares text files.

But what if Conformal was compromised? Conformal is a huge and complex tool, and writing a comparable equivalence checker from scratch would be impractical. One option is to verify Conformal using other third-party equivalence checkers, such as Formality by Synopsys or FormalPro by Mentor Graphics. Again, though, this relies on third-party tools. How can two netlists be compared without relying on large third-party CAD tools? One approach could be to rely on graph matching algorithms. Since netlists can be represented as graphs (where edges are nets and vertices are cells), comparing netlists can be done using a graph isomorphism algorithm. Such an approach is discussed in further detail in Section 9.2.1. While we used a third-party tool to find isomorphisms, creating such a program from scratch would be feasible, and far easier than replicating Conformal. If you wanted to completely eliminate third-party tools in the IP comparison step, such an approach could be a realistic solution.

8.2.4 Hardware Trojans Outside of the IP

By assuring an IP at the physical or logical level, an IP user might be tempted to think that their IP is safe from all physical or logical-level attacks. Indeed, if the gold standard IP is

free of any hardware trojans, and if the instantiated IP is identical to the gold standard, then the instantiated IP is hardware trojan-free. But what about a hardware trojan *outside* of the IP that can affect the run-time performance of the contents *inside* the IP? Such a scenario was brought to my attention by Dr. Brian Dupaix of the Air Force Research Laboratory. In this attack, a malicious component outside the IP creates some kind of disturbance, most likely heat-based, to a running design. Such heat-based disturbances can be caused by ring oscillators or short circuits [39], for example. This extra run-time heat could cause the IP to slow down, possibly even to the point of causing failure. The only defense against such an attack would be to carefully examine the parts of the design that do not belong to the IP, and verify that no such hardware trojans exist.

8.2.5 Malicious Place and Route Attack

Because Functional Assurance only secures the logic level of IP, it does not detect physical-level attacks. For example, consider an IP that is untampered at the logical level but tampered at the physical level. Such tampering could be in the form of a route that is excessively long and tortuous, which could slow down the circuit, reducing its performance. A similar attack would be in the IP placement: By deliberately placing the IP's cells far from each other, an attacker could cause unnecessary delays in a circuit, again reducing its performance. Because these are physical-level attacks, Functional Assurance is of no use. A user must use Physical Assurance to mitigate these types of attacks.

8.3 "IP" in a Broader Sense

This thesis focuses on using the IP Assurance Framework to eliminate redundant hardware trojan detection in IP. While the focus of this work is on IP—defined as standalone, reusable design content—there are applications beyond this standard definition of IP. Generally speaking, this work applies to any piece of digital design where engineers wish to only run through the hardware trojan detection process once. For example, consider a scenario in which engineers have created a custom, one-time-use design component as part of a larger digital design. To verify that it is safe, the engineers may want to use hardware trojan detection techniques on this custom design component. However, in the digital design process, there might be many iterations of the

final design, each of which may modify this custom component. Each modification introduces the possibility of a lurking hardware trojan. How can the engineers know if these modifications were benign, or malignant?

A simple solution is to only look for hardware trojans once, at the very end of the design process. However, there are two problems with this approach. First, as every engineer knows, rarely is a "final" design *actually* final—slight tweaks and modifications are made right up until a product ships. Therefore, any "final design" hardware trojan detection will likely be preemptive, and will have to be re-done for each further design modification. Secondly, doing a large, "final-step" hardware trojan detection phase goes against principles of scalable design. It is better to build up from from smaller, verified components than to try to verify a completed, monolithic design.

A better solution to this problem would be to use the IP Assurance Framework. Using the framework, a custom design component can be built and verified, and still be secured at any point of the design process. Such an approach is also more scalable, as custom components can be verified independently of a parent design. Thus while such a custom, one-time-use design component wouldn't typically be considered to be IP, it can still benefit from the IP Assurance Framework. In this sense, this work extends to design components that generally wouldn't be classified as IP.

CHAPTER 9. CONCLUSIONS AND FUTURE WORK

9.1 Conclusions

This work addresses the problem of IP assurance. Previous solutions include a variety of hardware trojan detection techniques, but lack a satisfactory way of applying and managing these techniques, leading to either redundant HT checking or unsecured designs. This work introduced the IP Assurance Framework, a framework that allows users to quickly and easily verify that their IP matches a trusted, gold standard IP. Two implementations of the framework were discussed: Physical and Functional Assurance. A suite of 53 benchmarks was created to test both implementations. Both approaches successfully distinguished between tampered and untampered designs. Both approaches have mild to moderate effects on IP timing and area, but such effects may well be worth the assurance the framework provides. This work advocates for the adoption of the IP Assurance Framework in today's digital design.

9.2 Future Work

This thesis demonstrates that Physical and Functional Assurance are viable approaches to securing IP. However, more work can be done to further improve this research.

9.2.1 Graph Matching

Conformal is a program that is intended to run some very heavy algorithms—such as the Boolean satisfiability problem—as quickly and efficiently as possible. It's well-tuned and industry proven. It's also unnecessarily powerful for this work and prohibitively expensive (at least without an academic license). Is it possible to demonstrate the same functional equivalence without relying on Conformal or other LEC tools? In my opinion, that answer is yes. To explain my reasons for this, it is helpful to first understand some of the history behind this project.

Before relying on Conformal to make functional comparisons between designs, we looked into graph matching algorithms. The idea was to use graph matching algorithms to determine similarity between digital circuits. We modeled our netlists as graphs, where vertices are cell pins and edges are nets. Our first attempt was to use measures of graph similarity, such as graph edit distance. We eventually abandoned this approach when we realized that graph similarity is not a good way to check for tampered designs—hardware trojans can be very small, and a low graph edit distance does not necessarily preclude the existence of a malicious circuitry.

Another graph matching technique we explored was finding graph isomorphisms between netlists. We did this using the tool GraphGrep [40]. Initial tests showed that some instantiated IP designs (such as `counter` and `aes128`) were actually isomorphic to their respective trusted IP. However, most instantiated and trusted IP pairs were *not* isomorphic, and so we decided that we needed a different approach to determining functional equivalence. At the time, we had not yet figured out how to properly apply the `DONT_TOUCH` constraint to prevent IP optimizations. As a result, we concluded that graph isomorphisms couldn't be reliably used to demonstrate functional equivalence. At the time, it seemed like finding a more powerful comparison method (such as Conformal) was the right solution. In hindsight, this switch may not have been entirely necessary.

Now that we have a way of preserving netlists and preventing optimizations, I suspect that it would be possible to switch back to making comparisons using graph isomorphism algorithms. Because the "vertices" and "edges" of netlists all have unique names, the graph isomorphism algorithm runs actually rather quickly—graph isomorphism algorithms only run slowly when vertices and edges do not have unique names. Future work may demonstrate that we could have achieved the same results by using the `DONT_TOUCH` together with a graph isomorphism tool, such as GraphGrep. Such an approach could likely be just as effective as Conformal, although this will have to be demonstrated with future work.

9.2.2 Providing Context to Conformal

In Functional Assurance, IP blocks are constrained with the `DONT_TOUCH` property to prevent optimizations and subsequently allow Conformal to determine functional equivalence. It may be possible to achieve the same results as Functional Assurance *without* requiring the use of `DONT_TOUCH` by giving Conformal context of the surrounding circuitry. For example, consider

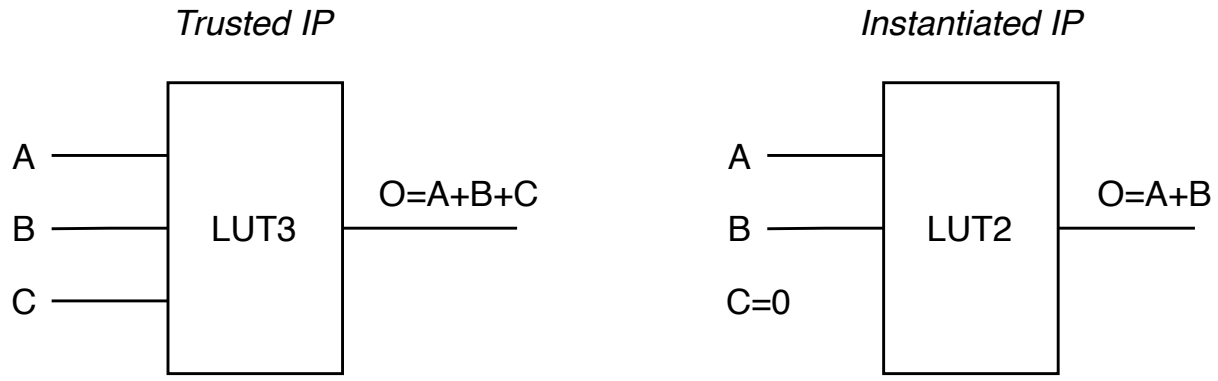

Figure 9.1: A LUT as part of a trusted IP and an instantiated IP. The inputs A, B, and C are boundary nets, and are sourced from outside of the IP. In the instantiating circuit, the value of C is tied to a 0, allowing the LUT to be optimized. It is clear that the two circuits have the same output, but only if the IP's instantiating circuitry is considered. In Functional Assurance, we gave Conformal only the netlist for the IP, which contains no information on surrounding context. This means that in our current approach, Conformal would declare these two circuits to be functionally inequivalent.

the two circuits shown in Figure 9.1, synthesized without the DONT_TOUCH constraint. In the instantiated IP, an input to the IP is tied to a logical 0, causing the circuitry near the input to be optimized. Upon observation, it is clear that the two circuits behave the same way. However, this can only be deduced by inspecting the IP *in context with the instantiating circuitry*. This poses a problem for Conformal, which only compares the insides of the IP, and has no knowledge of the logical 0 input. If the user could give Conformal the context in which the IP was instantiated, it might be possible to remove the DONT_TOUCH constraint, and still demonstrate functional equivalence. This would give the user even more freedom to optimize their IP.

9.2.3 Sweeping the Target Frequency

In Chapters 6 & 7, there was an attempt to measure Physical and Functional Assurance's effect on timing. As was discussed, however, the only way to get a good estimate of the effect on timing is to sweep the target frequency to find the fastest baseline and assurance-based designs Vivado can create, and then compare frequencies. Such a sweep would provide a better way of measuring the timing effects of Physical and Functional Assurance.

9.2.4 Addressing Limitations

As discussed in Chapter 8, there is concern that malicious circuitry *outside* of the IP could affect the timing and functionality of the circuitry *inside* the IP. So far, no work has been done to address this issue. However, the use of pblocks might provide a solution. If an IP can be contained in a pblock and sequestered from the rest of the design, it might be possible to provide assurance against malicious, external circuitry. The current pblock approach nearly achieves this; However, outside nets are still sometimes routed through the pblock. To fully sequester a design, isolation design flow might be needed. Such an approach may inhibit these types of attacks.

9.3 Contributions

As mentioned in the Acknowledgements, most of this work was done in tandem with my collaborator, Sean Jensen. Additionally, much of the theory in this work was developed with the help and guidance of Professors Brad Hutchings and Jeffrey Goeders. However, some work was done individually or independently of the rest of the team. I have listed below some of the portions of this project that I did either mostly or entirely on my own:

- I wrote the original TCL script that compared pblocks in Physical Assurance. Sean Jensen worked on the final version of this script.
- I built most of the synthetic benchmarks in the benchmark suite.
- I advocated for using Conformal to make functional comparisons between designs. I also did the initial work of figuring out how to use Conformal, and how to script it.
- I figured out to apply `DONT_TOUCH` to the cells and nets *within* the IP, in addition to IP cell itself. This application of `DONT_TOUCH` to these three items was the "secret recipe" that allowed us to demonstrate functional equivalence between trusted and instantiated IPs across our full benchmark suite.
- I used GraphGrep to find graph isomorphisms between netlists.
- Before the `DONT_TOUCH` technique was fully developed, I examined trusted and instantiated IPs that failed functional equivalence checking to determine why they had failed. Sean also

worked on this step, but we did so mostly independently of one another, and were typically examining different designs.

- I ran many of the physical and functional equivalence tests. I also helped automate these tests with Sean.
- I created modified user designs to make sure that Functional Assurance could detect tampered IP.
- I removed the `DONT_TOUCH` constraint in Functional Assurance, and compared these new designs with Conformal.

REFERENCES

[1] S. T. Jensen, "Towards Tools for Achieving Third-Party IP Assurance," Master's thesis, 2018. iii, 37, 38

[2] K. Thompson, "Reflections on Trusting Trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984. [Online]. Available: http://doi.acm.org/10.1145/358198.358210 iii, 54

[3] F. Lau, S. H. Rubin, M. H. Smith, and L. Trajkovic, "Distributed Denial of Service Attacks," in *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, vol. 3. IEEE, 2000, pp. 2275–2280. 1

[4] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, vol. 2. IEEE, 2000, pp. 119–129. 1

[5] M. Tehranipoor and F. Koushanfar, "A Survey of Hardware Trojan Taxonomy and Detection," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 10–25, Jan 2010. 1, 13, 14, 16, 17

[6] D. Drako, "New data from 372 engineers and managers surveyed on real IP reuse," [Online], Mar 2013, http://www.deepchip.com/items/0520-01.html. 3

[7] U. Farooq, Z. Marrakchi, and H. Mehrez, "FPGA Architectures: An Overview," in *Tree-Based Heterogeneous FPGA Architectures*. Springer, 2012, pp. 7–48. 7

[8] "Vivado Design Suite Synthesis Reference Guide (UG901)," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug901-vivado-synthesis.pdf, accessed: 2018-7-30, Version: 2016.2. 10

[9] X. Wang, M. Tehranipoor, and J. Plusquellic, "Detecting Malicious Inclusions in Secure Hardware: Challenges and Solutions," in *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, June 2008, pp. 15–19. 14, 16

[10] Y. Jin, N. Kupp, and Y. Makris, "Experiences in Hardware Trojan Design and Implementation," in *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*. IEEE, 2009, pp. 50–57. 15

[11] A. Baumgarten, M. Steffen, M. Clausman, and J. Zambreno, "A Case Study in Hardware Trojan Design and Implementation," *International Journal of Information Security*, vol. 10, no. 1, pp. 1–14, 2011. 16

[12] X. Wang, H. Salmani, M. Tehranipoor, and J. Plusquellic, "Hardware trojan detection and isolation using current integration and localized current analysis," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*. IEEE, 2008, pp. 87–95. 16

[13] R. S. Chakraborty, S. Narasimhan, and S. Bhunia, "Hardware Trojan: Threats and emerging solutions," in *2009 IEEE International High Level Design Validation and Test Workshop*, Nov 2009, pp. 166–171. 16

[14] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith, "Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 159–172. 16

[15] A. Malekpour, R. Ragel, A. Ignjatovic, and S. Parameswaran, "TrojanGuard: Simple and Effective Hardware Trojan Mitigation Techniques for Pipelined MPSoCs," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2017, pp. 1–6. 16

[16] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "MERO: A statistical approach for hardware Trojan detection," in *Cryptographic Hardware and Embedded Systems-CHES 2009*. Springer, 2009, pp. 396–410. 16

[17] S. Narasimhan, D. Du, R. S. Chakraborty, S. Paul, F. Wolff, C. Papachristou, K. Roy, and S. Bhunia, "Multiple-Parameter Side-Channel Analysis: A Non-Invasive Hardware Trojan Detection Approach," in *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 13–18. 16

[18] R. M. Rad, X. Wang, M. Tehranipoor, and J. Plusquellic, "Power Supply Signal Calibration Techniques for Improving Detection Resolution to Hardware Trojans," in *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2008, pp. 632–639. 16

[19] Y. Alkabani and F. Koushanfar, "Consistency-based characterization for IC Trojan detection," in *Proceedings of the 2009 International Conference on Computer-Aided Design*. ACM, 2009, pp. 123–127. 16

[20] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey, "Hardware Trojan Horse Detection Using Gate-Level Characterization," in *Proceedings of the 46th Annual Design Automation Conference*. ACM, 2009, pp. 688–693. 16

[21] J. Li and J. Lach, "At-Speed Delay Characterization for IC Authentication and Trojan Horse Detection," in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*. IEEE, 2008, pp. 8–14. 16

[22] Y. Jin and Y. Makris, "Hardware Trojan Detection Using Path Delay Fingerprint," in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*. IEEE, 2008, pp. 51–57. 16

[23] S. Jha and S. K. Jha, "Randomization Based Probabilistic Approach to Detect Trojan Circuits," in *2008 11th IEEE High Assurance Systems Engineering Symposium*. IEEE, 2008, pp. 117–124. 16

[24] F. Wolff, C. Papachristou, S. Bhunia, and R. S. Chakraborty, "Towards Trojan-Free Trusted ICs: Problem Analysis and Detection Scheme," in *Proceedings of the Conference on Design, Automation and Test in Europe*. ACM, 2008, pp. 1362–1365. 16

[25] M. Banga and M. S. Hsiao, "A Region Based Approach for the Identification of Hardware Trojans," in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*. IEEE, 2008, pp. 40–47. 16

[26] M. Banga and M. Hsiao, "A Novel Sustained Vector Technique for the Detection of Hardware Trojans," in *2009 22nd International Conference on VLSI Design*. IEEE, 2009, pp. 327–332. 16

[27] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. ACM, 2013, pp. 697–708. 17

[28] "Trust-hub," http://www.trust-hub.org. 17

[29] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan Detection using IC Fingerprinting," in *2007 IEEE Symposium on Security and Privacy (S&P '07)*, May 2007, pp. 296–310. 17

[30] M. F. Jacome and H. P. Peixoto, "A Survey of Digital Design Reuse," *IEEE Design & Test of Computers*, vol. 18, no. 3, pp. 98–107, 2001. 22

[31] E. Girczyc and S. Carlson, "Increasing Design Quality and Engineering Productivity Through Design Reuse," in *Proceedings of the 30th International Design Automation Conference*. ACM, 1993, pp. 48–53. 22

[32] "Vivado Design Suite User Guide: Hierarchical Design (UG905)," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug905-vivado-hierarchical-design.pdf, accessed: 2018-7-30, Version: 2016.2. 27, 37

[33] "Vivado Design Suite Tutorial: Hierarchical Design (UG946)," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug946-vivado-hierarchical-design-tutorial.pdf, accessed: 2018-8-22, Version: 2014.1. 28, 29

[34] X. Forums. BUFG in OOC module cannot be routed from. [Online]. Available: https://forums.xilinx.com/t5/Design-Methodologies-and/BUFG-in-OOC-module-cannot-be-routed-from/m-p/819160 37

[35] Xilinx Forums. Cell count in dcp changes when using read_checkpoint tcl command. [Online]. Available: https://forums.xilinx.com/t5/Design-Methodologies-and/Cell-count-in-dcp-changes-when-using-read-checkpoint-tcl-command/td-p/820660 38

[36] "Cadence Conformal Logic Equivalence Checker Datasheet," https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/conformal-equivalence-checker-ds.pdf, accessed: 2018-9-1. 44

[37] W. D. Casper and S. M. Papa, *Root of Trust*. Boston, MA: Springer US, 2011, pp. 1057–1060. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5_789 53

[38] "Project X-Ray," https://github.com/SymbiFlow/prjxray/blob/master/README.md. 55

[39] "Power Side-Channel DAC Implementations for Xilinx FPGAs," Master's thesis. 56

[40] R. Giugno and D. Shasha, "Graphgrep: A fast and universal method for querying graphs," in *Object Recognition Supported by User Interaction for Service Robots*, vol. 2, Aug 2002, pp. 112–115 vol.2. 59