Hardware Trojans in Reconfigurable Computing

DISSERTATION

A thesis submitted to the
FACULTY FOR COMPUTER SCIENCE, ELECTRICAL ENGINEERING AND MATHEMATICS
of

in partial fulfillment of the requirements
for the degree of *Dr. Ing*

by

QAZI ARBAB AHMED

Paderborn, Germany
Date of submission: November 2021

SUPERVISOR:
Prof. Dr. Marco Platzner

REVIEWERS:
Prof. Dr. Marco Platzner
Prof. Dr. Sybille Hellebrand

ORAL EXAMINATION COMMITTEE:
Prof. Dr. Marco Platzner
Prof. Dr. Sybille Hellebrand
apl. Prof. Dr. Wolfgang Müller
Prof. Dr. Christian Plessel
Dr. Heinrich Riebler

DATE OF SUBMISSION:
November 2021

To my Mother,

Father (Late),

Brothers,

Sisters,

Wife,

and

Son

Acknowledgements

Inestimable appreciation and earnest gratitude for the help and support are extended to many remarkable people who have been around me throughout this journey and without whom the accomplishment of this thesis would not have been possible. I would like to meekly express my thanks to all my colleagues, friends, family, and specifically to those mentioned in the following lines.

First and foremost, I would like to express my deepest thanks to my supervisor, Prof. Dr. Marco Platzner, for his invaluable guidance, persistent help, and immeasurable support throughout this research. He has always been kind and his generosity, dynamism, sincerity, scholarly advice, and professional attitude have deeply inspired, motivated and helped me to a great extent to complete this thesis. It was a great privilege and honor to work under his supervision where I have learnt a lot from his experience, skills, and leadership. Furthermore, I would like to thank him and Prof. Dr. Sybille Hellebrand for devoting their time and effort to review this thesis, together with all the committee members for evaluating my research.

I am sincerely grateful to the people of the CEG for providing such a pleasant and professional working environment during my stay. A special thanks goes to my colleague cum mentor Tobias Wiersema who has been there all the time for scientific discussions in a very professional way and also his help regarding social matters in Germany is highly regarded. Besides, many thanks also goes to Linus Witschen, Muhammad Awais, Hassan Ghasemzadeh Mohammadi, Lennart Clausing, Tim Hansmeier, Christian Lienen, Felix Jentzsch, Achim Lösch, Nam Ho, Tobias Kenter, Michael Lass and Heinrich Riebler for useful discussions over the time. A big thanks to Jennifer Lohse and Andre Diekwisch for their support in administrative and technical issues. In addition, I would also like to thank my friend Sajjad Hussain for his encouraging and motivational conversations in hard times. I would also like to express thanks to the HEC Pakistan and DAAD Germany for financing this work.

Last but not least, I am extremely grateful to all my family members for their infinite support and prayers. Above ground, I am indebted to my mother for her endless support and prayers throughout my life. A huge thanks to my wife Shafaq and son Maalik for their patience in both my physical and mental absence during my stressful days. Many thanks for counterbalancing my restless working days with their generous love, joy, and keeping peace and harmony at home. May God bless them!

Abstract

The battle of developing hardware Trojans and corresponding countermeasures has taken adversaries towards ingenious ways of compromising hardware designs by circumventing even advanced testing and verification methods. Besides conventional methods of inserting Trojans into a design by a malicious entity, the design flow for field-programmable gate arrays (FPGAs) can also be surreptitiously compromised to assist the attacker to perform a successful malfunctioning or information leakage attack. The advanced stealthy "Malicious Look-up-table (LUT)"-hardware attack activates a Trojan only when generating the FPGA bitstream and can thus not be detected by register transfer and gate level testing and verification.

This thesis mainly focuses on the two aspects of hardware Trojans in reconfigurable systems, the defender's perspective which corresponds to the bitstream-level Trojan detection technique, and the attacker's perspective which corresponds to a novel FPGA Trojan attack. From the defender's perspective, we introduce a first-ever successful preconfiguration countermeasure against the "Malicious LUT"-hardware Trojan, by employing bitstream-level Proof-carrying Hardware (PCH) and present the complete design-and-verification flow for iCE40 FPGAs. Likewise, from an attacker's perspective, we present a novel attack that leverages malicious routing of the inserted Trojan circuit to acquire a dormant state even in the generated and transmitted bitstream. The Trojan's payload is connected to primary inputs/outputs of the FPGA via a programmable interconnect point (PIP). The Trojan is detached from inputs/outputs during place-and-route and re-connected only when the FPGA is being programmed, thus activating the Trojan circuit without any need for a trigger logic. Since the Trojan is injected in a post-synthesis step and remains unconnected in the bitstream, the presented attack can currently neither be prevented by conventional testing and verification methods nor by bitstream-level verification techniques.

Zusammenfassung

Im Wettstreit zwischen der Entwicklung neuer Hardwaretrojaner und entsprechender Gegenmaßnahmen beschreiten Widersacher immer raffiniertere Wege um Schaltungsentwürfe zu infizieren und dabei selbst fortgeschrittene Test- und Verifikationsmethoden zu überlisten. Abgesehen von den konventionellen Methoden um einen Trojaner in eine Schaltung für ein Field-programmable Gate Array (FPGA) einzuschleusen, können auch die Entwurfswerkzeuge heimlich kompromittiert werden um einen Angreifer dabei zu unterstützen einen erfolgreichen Angriff durchzuführen, der zum Beispiel Fehlfunktionen oder ungewollte Informationsabflüsse bewirken kann. Die hochentwickelte Attacke "Heimtückische Look-up-table (LUT)" aktiviert ihren Tarnkappentrojaner beispielsweise erst, wenn der FPGA Bitstrom geschrieben wird, so dass sie weder von Tests noch Verifikationen auf der Register-Transfer-Ebene oder der Gatterebene erkannt werden kann.

Diese Dissertation beschäftigt sich hauptsächlich mit den beiden Blickwinkeln auf Hardwaretrojaner in rekonfigurierbaren Systemen, einerseits der Perspektive des Verteidigers mit einer Methode zur Erkennung von Trojanern auf der Bitstromebene, und andererseits derjenigen des Angreifers mit einer neuartigen Angriffsmethode für FPGA Trojaner. Für die Verteidigung gegen den Trojaner "Heimtückische LUT" stellen wir die allererste erfolgreiche Gegenmaßnahme vor, die durch Verifikation mittels Proof-carrying Hardware (PCH) auf der Bitstromebene direkt vor der Konfiguration der Hardware angewendet werden kann, und präsentieren ein vollständiges Schema für den Entwurf und die Verifikation von Schaltungen für iCE40 FPGAs. Für die Gegenseite führen wir einen neuen Angriff ein, welcher bösartiges Routing im eingefügten Trojaner ausnutzt um selbst im fertigen Bitstrom in einem inaktiven Zustand zu verbleiben: Nachdem der Trojaner in der Place-and-route Phase fertig mit der Schaltung und den Ein- und Ausgängen verbunden wurde, kappen wir das Aktivierungssignal an einer der programmierbaren Verbindungspunkte, so dass er nicht aktiviert werden kann. Erst wenn wir mit diesem Bitstrom tatsächliche Hardware rekonfigurieren, stellen wir die Verbindung wieder her, so dass der Trojaner auf dem Gerät

aktiviert werden kann. Hierdurch kann dieser neuartige Angriff zur Zeit weder von herkömmlichen Test- und Verifikationsmethoden, noch von unserer vorher vorgestellten Verifikation auf der Bitstromebene entdeckt werden.

Table of Contents

Chapter 1

Introduction

This chapter motivates the readers by broadly depicting the significance of hardware security challenges to the modern computing paradigm due to the hardware Trojans threat and outlines the fundamental concepts of reconfigurable computing, particularly about FPGAs in Section 1.1, overviews the focus of this thesis in Section 1.2, briefly discusses our contributions towards the FPGA security in Section 1.3, and then details the organization of the rest of the thesis document in Section 1.4.

1.1 Motivation

The hardware of computing systems was thought to be a safe and trusted entity because of its non-malleable nature, thus tampering with a manufactured device seemed impossible. On the contrary, software was considered more vulnerable to virus and malware attacks which usually resulted in corrupt or damaged software/files. As a result, the antivirus programs and the malware detection phenomena were introduced by computer scientists and programmers to mitigate these attacks. However, during the last decade, some of the potential security attacks on the hardware of a computing system have tremendously warned and alerted the hardware design community to put the efforts for the development of tamper-free designs, or anti-tamper mechanisms to nullify the effect of any extraneous logic, for assuring the customers to trust the hardware.

The institute of electrical and electronics engineers (IEEE) spectrum published a report in May 2008 about the Israeli attack on the nuclear system of Syria in September 2007 which claims that the strike between Israel and Syria was a failure of a radar system, that allowed Israeli jets to bomb a nuclear system in the north-east of Syria. Conceivably, there might be a hidden "backdoor" deliberately implanted into the commercial off-the-shelf (COTS) microprocessors used in the Syrian radar, which has helped the adversary to disrupt the functionality of the radar by sending a piece of code (program) to these chips. Such an attack thwarting all

the testing/detection methodologies available at that time was extremely surprising for the panel of defense advanced research projects agency (DARPA), therefore to mitigate such attacks, they initiated the establishment of the DARPA trust program for security and trust in the integrated circuits (ICs). Furthermore, it is stated by the Dean Collins, who was a deputy director of DARPA's microsystems technology office and program manager for the trust in IC initiative, that field-programmable gate array (FPGA) are the main driving force in many defense companies, which are generic and cheap as compared to application-specific integrated circuits (ASICs), "If you make a mistake on an FPGA, hey, you just reprogram it, that is the good news. The bad news is that if you put the FPGA in a military system, someone else can reprogram it" [1]. And the story of hidden backdoor/kill switch continued to rise over the last few years with the battle between the attackers and defenders. It is also reported recently that, at the DefCon cybersecurity conference 2019, a group of extremely expert hackers succeeded when they attempted to damage an essential flight system for a U.S. military fighter jet [2].

Hardware Trojans, i.e., malicious circuit inclusions, have grown into a mature research field over the last two decades, which has raised many questions regarding the integrity, security and trust in digital systems. Attack vectors are plentiful, especially since nowadays most IC designers rely on third-party IP cores and closed-source electronic design automation (EDA) tools, while the manufacturers outsource the actual fabrication step to third-party foundries that are often in different countries or even continents, to lower the cost and to speedup the development. As a consequence, attackers have the opportunity to manipulate a design at almost any stage of the IC development life cycle [3]. In fact, an attacker can insert malicious circuitry during each stage of system-on-chip (SoC) design flow such as, by exploiting the synthesis stage to insert additional malicious logic, modification of wires and transistors during the layout stage, and even, the design can also be manipulated by the untrusted foundry during fabrication process [4]. Such an undesired modification of a circuit, also known as hardware Trojan [5], can alter its functionality, provide a covert channel to leak sensitive information, or even open a back door into the IC.

A hardware Trojan[1] is usually defined to comprise a trigger and a payload [6, 7]. Typically, a trigger mechanism is implemented in a way that activates the payload mechanism either always, upon reception of some

[1]We will use the term hardware Trojan and Trojan interchangeably in this thesis.

stimulus specified at design time, or at a some pre-determined time during the operation. Triggers of Trojans play an important role to conceal them throughout the development process of an IC. In order to hide the malicious circuitry, an adversary would design the trigger in a way that the Trojan does not affect the functionality of the original circuit under normal conditions. In literature, various trigger implementations have been published so far and most of them depend on rare events such as counters or a specific unlikely signal pattern to evade detection at the functional testing stage [7–11]. However, such Trojans can be caught by extensive functional simulation and testing during design time, as state-of-the-art detection techniques are exploiting the fact that malicious circuitry is more likely to reside in the rarely or unused portions of the circuit and thus investigate these areas with much more scrutiny [12–14].

Reconfigurable computing has become a more dominant paradigm in modern semiconductor-based integrated circuits (ICs) since the development of FPGAs, which possess the (re)configurable fabric for enhanced computing efficiency while offering the software-like flexibility to update the design at runtime. FPGAs have been a large driving force for reconfigurable computing for many years. Reconfigurable computing generally refers to the methods that fully utilize the potential of underlying reconfigurable hardware resources. Depending upon what the reconfigurable computing user might want, FPGA devices could be the reasonable choice concerning cost and rapid time to market for reconfigurable computing. Compared to application-specific integrated circuits (ASICs), FPGAs benefit from a flexible and re-programmable computation fabric and short time-to-market and have thus become a fundamental part of the embedded systems space, including the internet-of-things (IoT) and end-user electronics.

The real-time processing efficiency and cost-effectiveness in reconfigurable modules and accelerators have rendered them incredibly prominent and preferable in autonomous and performance-critical applications such as aerospace, autonomous driving/advanced driver assisted systems (ADAS), and deep learning (DL) [15, 16] and also used in military purposes. Some of the big partners in the software and information technology (IT) industries, i.e., Amazon, Alibaba, and Huawei [17–19], are already using FPGAs to accelerate the computing power in their respective data centers/clouds.

Microsoft is primarily using FPGAs to boost the efficiency of artificial intelligence (AI) algorithms in some applications, e.g., Bing search engines [20]. Furthermore, the rise of the internet-of-things (IoT) in recent

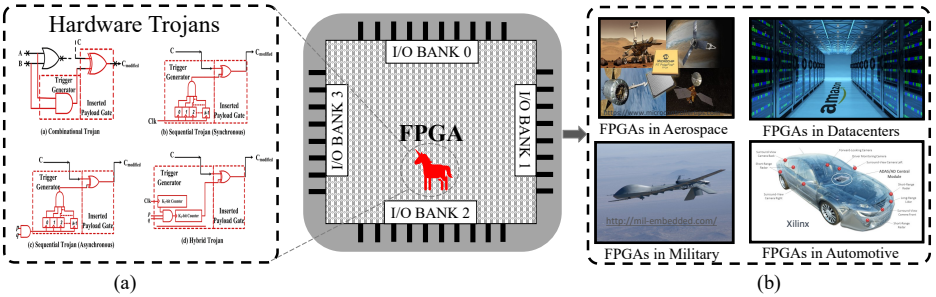

Figure 1.1: (a) Hardware Trojan circuits types (b) Use of FPGAs in critical applications.

years has shifted the view of the digital world towards smart and connected devices which are being used in many critical applications such as healthcare systems and wearables, smartphones, smart cities, and industrial or home automation systems, just to mention a few.

Figure 1.1 exemplifies the adoption of FPGAs in data-sensitive applications by different companies, such as automotive and e-commerce companies, however, these devices may also be vulnerable to hardware security threats. Figure 1.1 (a) shows the types of hardware Trojan circuits based on different triggers and payload implementations, whereas Figure 1.1 (b) shows the use of FPGAs in various critical applications. Security of these critical applications is becoming a big challenge due to the comprehensive software and hardware attacks on the devices themselves and the networks used. Similar to the virus attacks in the software of computer systems, hardware Trojans attack has emerged as one of the most serious attacks in integrated circuits that can not only damage the integrity of an IC but can leak the secret key from a cryptographic device too [21]. Their property of staying dormant during post-manufacturing testing makes it challenging for a verification team to detect such Trojans using test vectors and fault detection methods.

Most of the work on hardware Trojans has been focused on fixed-function circuits, i.e., application-specific integrated circuits (ASICs), while very little effort has been shown towards dynamically reconfigurable hardware such as FPGAs, where Trojans can affect not only the fixed-function ASIC part, but also the dynamic configuration. In FPGAs, the implemented design is loaded onto the manufactured and tested device in form of a configuration bitstream. This process resembles software programs but differentiates FPGAs from ASICs, where the functionality of the manufactured device cannot be changed once it is fabricated. Hence, there is the flexibility to the customer to update erroneous designs or configure new designs on-the-fly. While hardware tampering attacks

by untrusted foundries, such as the insertion of hardware Trojans into silicon, are possible, FPGAs can be considered more resistant to such attacks compared to ASICs since at manufacturing-time an attacker would have no knowledge about the design which would later be implemented on the FPGA device. Nevertheless, a bitstream configuration file may be susceptible to reverse engineering attacks, which would allow an attacker to steal the implemented design or insert malicious circuitry and regenerating the bitstream [7, 22]. Thus to avoid any malfunctioning during operation, it is of utmost importance to provide security guarantees for these reconfigurable systems used by the customer.

FPGAs being flexible and reprogrammable, have been widely used as a prototype to implement and verify application-specific integrated circuit (ASIC) designs. The reconfigurability and re-programmability property of FPGAs provides the opportunity to the designers and developers to implement their designs and to make quick changes, if needed, according to the requirements. Reconfigurable hardware is made up of an array of identical re-programmable cells connected via distributed programmable interconnect structure, thus more vulnerable to attacks, such as counterfeiting and reverse-engineering, when compared to ASICs, because the majority of the chip's region is covered by the uniform structure of logic blocks and interconnects. However, the post-manufacturing programmability, which enables the user to upload the functional design after the device fabrication and delivery, makes FPGA device more reluctant to the hardware Trojan attacks and thus reliable than ASICs because the malicious modification made in the FPGA device (the base array) will almost not affect the design uploaded afterward.

The attacks on the FPGA device containing fixed function blocks are different from its dynamically reconfigurable fabric counterpart, where the latter is configured using a bitstream configuration file of an application design. There could be several possibilities for an adversary to attack the FPGA configuration, for instance, through physical access, directly modifying the bitstream, malicious third-party IP inclusion, or using subverted electronic design automation (EDA) tools. Thus, for the successful adoption of FPGAs to be used as an acceleration platform in future hardware technologies, a bitstream configuration file, besides the attacks on the FPGA device itself, also needs to be thoroughly investigated and verified against potential hardware Trojans threat.

The capability to compute functions in parallel have made FPGAs a

better choice for implementing cryptographic algorithms where extensive bit and byte level computations are required, such as advanced encryption standard (AES), that is used to allow for secure data transmission in many critical applications [23]. Subsequently, such devices are responsible for the secret flow of information, which may be directly related to the user's privacy. Consequently, there is a high risk of secret information being stolen unnoticed through inserted hardware Trojans. Using side channel analyses to leak secret information has been proven to be a practical attack for FPGAs where the attacker can gain physical access to the FPGA and use power side channels to retrieve, e.g., a secret key [24, 25]. Sophisticated attackers might even attempt to insert malicious circuitry that remains inactive until it is triggered by a specific condition or external input to circumvent all the design-time testing and verification processes. Such an attacker could be a dishonest employee in the design house who inserts the Trojan in a design, or there may be a Trojan in a third-party Intellectual Property core IP provided by an untrusted vendor. Besides that, there is also a chance that both, the design house and IP vendors, are trustworthy, but the EDA tools maintained by the vendors are undermined, resulting in an enhanced power to the attack by infecting a bundle of designs in just one go. The authors in [26] recently presented an attack by introducing a stealthy "malicious lookup table (LUT)" hardware Trojan inserted during design flow that employs a two-stage mechanism of insertion and activation. This Trojan remains dormant throughout the design phase and is activated when the bitstream is written, thus circumventing design-time verification techniques such as [13, 14].

This thesis addresses the new bitstream-level countermeasures against the malicious circuit inclusion, specifically hardware Trojans, in an application design of a reconfigurable system during design time or even during the FPGA design flow which activates only in a bitstream. Further, it introduces a novel attack based on the malicious routing of a Trojan circuit, in which the Trojan remains dormant even in a bitstream and only activates when the device is configured, thus alerting the verification teams and end-users to explore new solutions/countermeasures for security and trust in FPGAs. To the best of our knowledge, detection of hardware Trojans in the bitstream and the post-configuration activation of a Trojan, i.e., malicious routing-based Trojan attack, in FPGAs have not been investigated prior to this thesis.

1.2 Focus of this thesis

The focus of this Ph.D. thesis is to present and discuss the new types of security threats in FPGAs, in particular, hardware Trojan insertion during the FPGA design flow by a malicious entity or a subverted design tool, which adopts a silent state through the design flow and becomes active only in the bitstream [26]. In this thesis, we experimentally analyze and demonstrate a successful countermeasure against such bitstream-level Trojans, which has not been explored prior to this thesis. Further, we present a new Trojan attack where the Trojan could still maintain its dormant state in the bitstream[2] and only activates in the FPGA device. This kind of attack resembles ASICs where some of the Trojans activate only when the device is in use. However, in the case of FPGAs, it is the bitstream configuration file that carries the Trojan all the way to the FPGA device without being revealed during the testing and verification stages. To the best of our knowledge, this kind of threat is also the first to show that even a verified bitstream may cause a device malfunctioning or even, leak sensitive information physically or through remote access.

1.3 Contributions

This Ph.D. thesis mainly contributes to show that bitstream-level Trojan attacks in FPGAs are realistic, more sliest and might easily and automatically be inserted by using a subverted FPGAs design flow. Towards this, bitstream level-verification in FPGAs could be the possible solution to detect and report such kind of stealthy Trojans.

During this thesis, we experimentally confirm that a bitstream-level proof-carrying hardware (PCH) approach is able to detect the stealthy malicious Trojan presented in [26]. We follow the attack presented in [26], in fact using the reference implementation provided by the authors, where compromised EDA tools add and activate a hardware Trojan into a design in two stages, making sure that it is dormant, and thus virtually undetectable, in every step of the design flow except the final bitstream. We propose to use a bitstream-level proof-carrying hardware (PCH) approach to detect the stealthy Trojan that is injected and activated in the compromised design flow. Replacing the consumer's need to trust in other parties with hard evidence is the core benefit of our approach, which places the computational burden of verification on the producer of a hardware module.

[2]We will use the term bitstream and bitstream configuration file interchangeably in this thesis.

However, irrespective of being a powerful verification tool, our proposed PCH flow would only detect the Trojans which change the functionality of the circuit upon activation, i.e., the Trojan in malicious LUT attack. Skillfully designed non-functional Trojans, e.g., the Trojan that provides a covert channel upon activation to leak secret information or the Trojans that are externally triggered to change the behavior of device parameters, may even avoid our proposed bitstream-level verification. To show the ineffectiveness of a bitstream-level verification for a non-functional key leakage Trojan, we propose a novel malicious routing-based Trojan that can intensify the stealthy nature of the inserted Trojan, such that the Trojan remains dormant in the bitstream as well and is only activated when actually configuring the device. The Trojan circuitry is introduced after synthesis, i.e., when the netlist is being read by the placement and routing tool, and maliciously routed so that it circumvents even bitstream verification. As the presented novel Trojan is activated after the bitstream generation step, the certificates generated by the PCH technique would lead to false-negatives. The contributions added by this thesis towards manifesting the trust and verification of the reconfigurable systems are as follows:

- We present a bitstream-level proof-carrying hardware (PCH) method within the scenario defined in [26], which, to the best of our knowledge, is the first that is able to detect a stealthy Trojan before it runs on an FPGA. Using also tools from the IceStorm project [27], we present a complete design-and-verification flow for iCE40 FPGAs that is able to protect bitstreams for these targets with the full power of PCH (Chapter 3).

- We introduce a novel FPGA Trojan that remains inert throughout the design flow, even in the bitstream, which to the extent of our knowledge is the first to circumvent even bitstream-level verification techniques. Trojans inserted by EDA design tools so far are inserted either at a design stage or into the bitstream. Our attack is based on the malicious routing of a Trojan circuit that is unconnected from the actual user design before configuring the target device, and that is reconnected by a malicious programming tool to activate the Trojan (Chapter 4).

1.4 Organization of the thesis

This thesis document is structured as follows:

Chapter 1, the current one, we introduce the research topic and the motivation for this thesis, the focus and the contributions of this thesis followed by the organization of the document.

Chapter 2 explains the background of and related work of the research area in the following order: The field-programmable gate array (FPGA) with respect to its technology, architecture and design flow tools, hardware Trojans in general with respect to their structure and classification, hardware Trojans in FPGAs and the related work on hardware Trojan designs and state-of-the-art detection methods in FPGAs.

In Chapter 3 a bitstream-level proof-carrying hardware approach is proposed to detect the Trojans that are activated in the post-synthesis step or only in the bitstream, e.g., stealthy malicious LUT hardware Trojan that is inserted by the compromised FPGA design flow. We discuss the steps involved in our approach for the verification of a bitstream configuration file along with experimental setup and implementations. Finally, we evaluate our methodology to detect stealthy Trojan detection based on three different scenarios and compare our findings to state-of-the-art techniques to substantiate the effectiveness of the approach.

In Chapter 4 we propose a novel post-configuration activation of an inserted Trojan circuit that leverages the malicious routing during the placement and routing step in the design flow. We give an overview of the two-phased attack methodology, where the first phase of the attack is responsible for the Trojan circuit insertion and disconnection from the original circuit, therefore, remains inactive, whereas the second phase is held responsible for the establishment of the connection to the original circuit for activation of a payload during the configuration of the device. Next, we evaluate the approach experimentally using examples and discuss the results.

The last two chapters, Chapter 5 and Chapter 6 briefly summarize the contributions to conclude the thesis and provide insights into the possible future research directions in the domain of hardware Trojans in dynamically reconfigurable computing and emerging technologies.

Chapter 2

Background and Related Works

This chapter throws light on the relevant background knowledge that is necessary to understand the rest of the chapters of this thesis and provides a detailed classification of the existing works. First, it explains different aspects of field-programmable gate arrays field-programmable gate arrays (FPGAs) such as the technology, architecture, and design flow tools in Section 2.1. Then, it studies a plethora of hardware Trojans threat in modern digital circuits concerning their structure, sources, taxonomy, and platform in Section 2.2, and finally, categorically explains the relevant work in the context of this thesis in Section 2.3.

2.1 Field-Programmable Gate Array

As the name implies, field-programmable gate array (FPGA), is a programmable integrated circuit (IC), comprised of two dimensional arrays of configurable logic blocks (CLBs) and programmed interconnects arranged in a matrix-like pattern that can be reconfigured recurrently in

the field, i.e., after deployment of the device. The arrangement of the elements in an FPGA is somewhat identical to the masked programmable gate arrays (MPGAs), from where the name "gate arrays" is adapted for FPGAs, which is a type of application-specific integrated circuit (ASIC), comprising the logic gates structured as an array, connected through mask programming, used to analyze numerous designs [28].

The concept of programmable logic was introduced in the early 1970*s* as a programmable logic array (PLA), the first-ever programmable logic device (PLD), which had two-level programmable logic structures. Considering the tradeoff between the programmability feature, poor speed, and high cost, PLAs did not succeed much which resulted in the innovation of programmable array logic (PAL). The logic in PALs is implemented using a configurable `AND` array, which produces the product terms, along with a fixed `OR` gate which sums the product terms, thus completing the functionality of the implemented logic. Any logic function could be implemented using PAL but not in a single stage which ultimately causes delays. However, larger PAL designs could improve speed but the subsequent increase in area, performance, and power consumption is not feasible. To overcome the speed and cost issues in PALs and still benefiting the programmability aspect, PLAs and PALs were combined to form simple programmable logic devices (SPLDs). SPLDs, over the years, continued to be developed as more prominent devices, such as field-programmable devices (FPDs), which provide the user an ease to configure by using the straightforward electrical apparatus, contrary to the previous devices, which required a lot of work and special equipment for configuration [29]. Later in the 1980s, SPLDs evolved as complex-programmable logic devices (CPLDs) to implement larger and complex designs. However, over time, as the user logic circuit size increased, the logic capacity of CPLDs was not sufficient, thus MPGAs were used for larger designs with the overhead of cost and manufacturing time and reduced flexibility. This limitation thus led the engineers to come up with a more generic and efficient programmable architecture, in particular, low cost and short time-to-market solution, which was fulfilled by Xilinx later in 1985, by introducing the first-ever FPGA, the Xilinx XC2064, that had the array of 64 logic blocks, containing 3-input lookup tables (LUTs) and one register each [30]. FPGAs provide highly scalable architectures where any combinational and sequential logic function could be implemented without the trade-off of performance, capacity, and delay between the growing array size (like AND array in PALs) and routing layout. As stated by Jonathan Rose et al, "the major difference between FPGA's and

MPGA's is that an MPGA is programmed using integrated circuit fabrication to form metal interconnections, while an FPGA is programmed via electrically programmable switches much the same as traditional PLDs". Thus a single FPGA can be repeatedly programmed by the user without the burden of fabrication cost and time.

The early FPGA devices, categorized as the age of invention (1984-1991) by Trimberger [31], included very few logic cells for the implementation of the user logic function, i.e., only 64 logic cells in XC2064, however, the size of the die was even greater than the today's microprocessors with a substantial cost factor. Increasing the logic on the die to implement the larger design would potentially increase the die size, consequently increasing the cost. To balance the cost per die, the designers enhanced the efficiency and flexibility, which resulted in growing of FPGAs in the market with success.

2.1.1 Programming Technology

FPGAs are generally configured by a bitstream configuration file, which contains configuration bits to be stored at the programmable locations, i.e., logic blocks and programmable switches. To this end, how a bitstream file can be stored, FPGAs have been classified into the three main categories based on the programming technology used, which are briefly discussed as follows:

1. **SRAM based FGPAs:** The fabric in SRAM-based FPGAs is composed of the volatile static random access memory (SRAM) cells, where the configuration data bits are stored and held as long as the voltage is applied. Therefore such an FPGA would need to be reconfigured at each power cycle. However, the major advantage is that SRAM technology-based FPGAs can be rapidly reprogrammed an unbounded number of times, with the support of external nonvolatile memory for storing and loading the data on a power reset, thus signifying the dynamic reconfiguration behavior [25, 32, 33]. Since SRAM-based FPGAs, such as Virtex, Artix, and Kintex by Xilinx, and Stratix, Cyclone by Intel/Altera, are most widely used as reconfigurable computing hardware, the experimental evaluation in this thesis will be done on the same technology device.

2. **Flash memory-based FPGAs:** Flash memory-based FPGAs are originated from flash technology, for instance, floating gate transistors, that are non-volatile cells characteristically, hence, do not need the power to

hold the configurations bits [25, 32, 33]. The key benefit is thus non-volatile nature yet reconfigurable numerous times by setting the reconfiguration mode using, for example, erasable programmable read-only memory (EPROM), however, due to the larger structures of flash cells, they lack in performance. Examples of flashed-based FPGAs include Igloo and ProAsic from Microsemi semiconductor corporation.

3. **Antifuse FPGAs:** Antifuse FPGAs are one-time programmable devices, made up of antifuse technology in which an antifuse element controls the configuration of the device. Antifuse technology works opposite to fuse technology where the fuse blows up when the high current is applied to the circuit, however, an improperly blown fuse can grow back and reconnect over time. Whereas an antifuse element starts with the high resistance and creates a permanent conductive path when the current/voltage exceeds a certain value in a circuit. Although not reconfigurable, antifuse FPGAs are the most reliable option for integrated circuits where the disconnection of the circuit network is not desirable, for instance in the military and aerospace missions. Antfiuse switch technology offers lower delays in the routing and wires because of their smaller sizes and programming is normally faster as compared to other technologies. Some of the recent commercially offered anitfuse FPGAs offered by Microsemi semiconductor corporation are Axcelerator®, SX-A, MX, and eX. [25, 34].

2.1.2 Architecture

The core components of FPGA architecture include configurable logic blocks (CLBs), programmable interconnects (PIs) with switch box and routing channels, and configurable input/output (I/O) blocks, while some of the modern FPGAs also consist of block RAM (BRAM) to implement larger memories and to store several kilo-bits of data, and some special blocks, e.g., digital signal processing (DSP). The most common structure of an FPGA with some basic components is shown in Figure 2.1. The connection between these components is made possible by a configurable interconnect to implement any arbitrary application/user design. Some of the core components are briefly described to understand the architecture of the FPGA in the following sections.

Configurable Logic Blocks

Fundamentally, a configurable logic block in FPGAs architecture consists of several basic logic elements (BLEs) that are grouped using intra-CLB

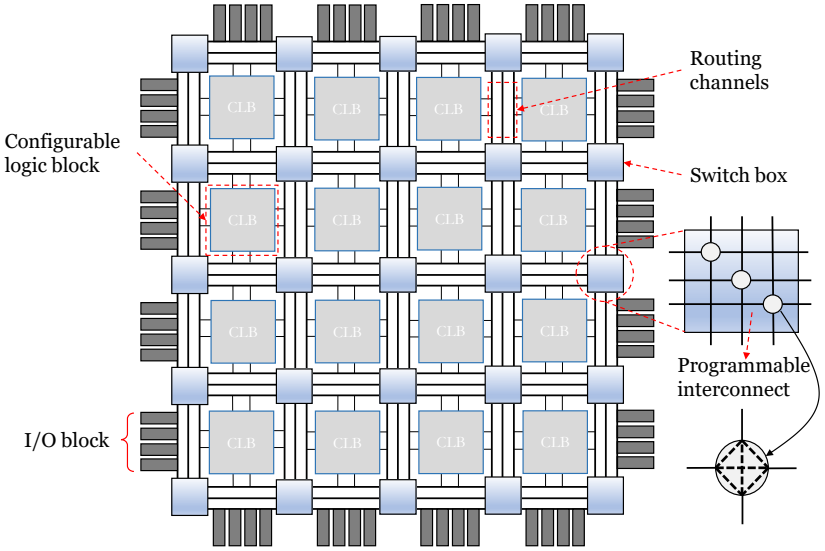

Figure 2.1: A typical FPGA 4×4 array structure with island style routing layout comprising of CLBs, programmable interconnects with three wiring channels vertically and horizontally, and I/O blocks.

interconnects to form a larger block to implement the application design. A simple basic logic element BLE comprises of one or more *K*-LUTs that are used to implement any combinational logic function: where *K* is the number of inputs to the LUT, which for modern FPGAs varies from 4:1 to 6:1, flip-flop (FF) (typically D-FF) to implement the sequential logic and a MUX to switch between the outputs of the combinational and the sequential circuit [33]. Figure 2.2 illustrates the internal view of a simple CLB. In some devices, a BLE also consists of a carry-logic chain to efficiently execute the arithmetic functions along with LUT, FF, and a MUX. In the modern-day devices by Xilinx however, each CLB contains two *slices*, where a *slice* is an additional intermediate level to their devices having four BLEs, MUXes, and carry-logic [35]. Note that the different vendors may use distinct names for the components in FPGA architecture, for instance, the term used for a logic block by Xilinx is CLB, whereas Intel/Altera calls this a logic array block (LAB). Similarly, Xilinx refers to the basic building blocks as logic cells, while Intel/Altera names them as BLEs [36].

Programmable Interconnects

In addition to the number of reconfigurable resources available in FPGA architecture, the overall performance and efficiency also depend on the style of how these resources are connected and the flexibility with respect

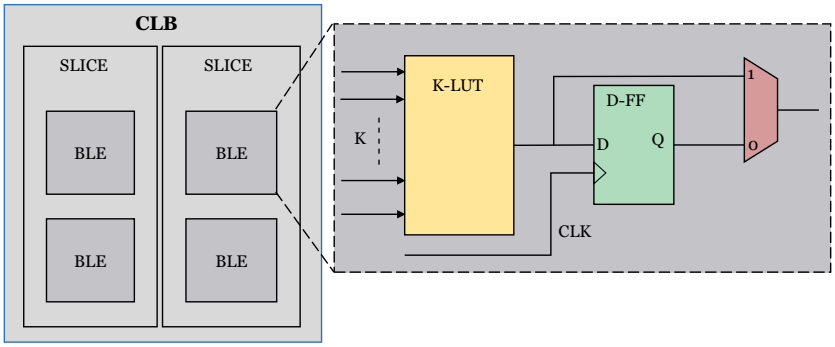

Figure 2.2: The inside view of a simple CLB containing two slices and each slice containing two BLEs, along with a BLE comprising of LUT, FF, MUX, and a clock (The number of BLEs in a slice and the number of slices in a CLB depends on the FPGA family).

to the routability, which is measured as the ratio of the overall number of successfully routed connections [37]. The flexibility, amount, connectivity, and arrangement of FPGA components, listed above, are referred to as routing architecture [29]. The most essential aspect in FPGA architecture is the routing and interconnect network of a design, that should successfully route the application design to the correct paths with the minimum delay and no congestion on the FPGA. The area covered by routing is mostly larger than the logic area and can take up to $70 - 90\%$ of the total FPGA's area [28, 38].

For the application designs, where more than one CLB is required to implement the logic functions, programmable interconnects serve the purpose to connect several CLBs to perform the required computation and to route the signals coming from the I/O block or going to the I/O block, which in turn could provide connectivity to the off-chip resources, if required, or any other external peripherals [39]. Before we go more into the details of routing, some terminologies need to be understood beforehand. For example, a wire segment is a wire that can be connected to one or more switches from both ends to carry the signal within the components. A track, that is formed when there appears an order of one or additional wire segments. Further, the group of these parallel tracks makes a routing channel [28, 38], as highlighted in Figure 2.1.

The basic routing block in almost all FPGA architectures are connection block and switch block. Depending on the speed and area parameters, the wire segments, tracks, routing channels, connection blocks, and switch block/box, also called collectively, segmented interconnect, the routing architectures can be categorized as island-style routing and hierarchical routing. The island-style architecture is depicted in Figure 2.3,

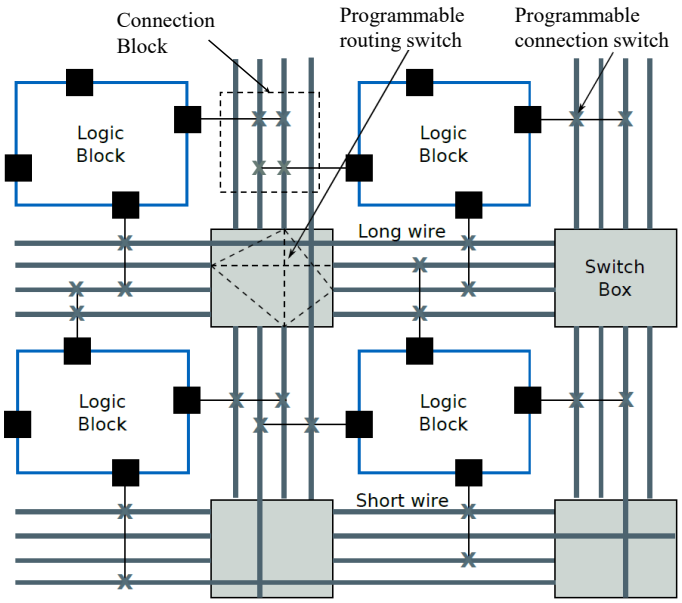

Figure 2.3: Island-style routing architecture for FPGAs. Taken from [40].

which is the most popular layout style in modern FPGAs. The arrangement is a reflection of an island of logic blocks floating in the sea of interconnects. In this layout, the vertical and the horizontal routing channels are uniformly distributed along with the connection blocks, which provides the connectivity of the logic[1] blocks to the routing network. To maintain a balance between the area and delays of the routing network, in particular, the logic block's inputs and outputs are connected through various switches boxes by using long wires, and the short wires are used to connect the I/Os of a logic block to the neighboring switch box.

The connection block may consist of one or more programmable connection switches, as shown in Figure 2.3, which offers the connectivity of a CLB from its inputs and outputs to the wire segments in a routing channel, i.e., horizontal and vertical tracks thus advancing the routing flexibility. Similarly, the switch block consists of programmable routing switches in Figure 2.3, which is responsible for offering the connectivity between the wire segments or tracks to all its four sides. The connectivity of these tracks highly depends on the design of the switch block, which determines whether to route the signal straight or turn it in either direction. Hierarchical routing is another area-efficient approach of FPGA routing that is used to minimize the signal delay caused in the long wires when it travels from far away. To do so, more local connections are

[1]The term logic block will be used interchangeably with CLB throughout this chapter.

made possible by making small clusters of logic blocks, e.g, a 2×2 cluster could have 16 logic blocks in a hierarchy, thus providing intra-cluster routing using only short wires, eluding the congestion on the lines connecting various clusters, while one a few long wires are preserved for larger distances [39]. In the context of this thesis, we will use the FPGAs with island-based routing layout architecture for experimental validation where we will introduce an FPGA routing-based hardware Trojan attack that uses one of the programmable interconnect points (PIPs), as a Trojan PIP (TPIP), which plays the role of the Trojan trigger and helps the Trojan circuit to remain undetectable even in the bitstream. More details will follow in Chapter 4.

I/O Blocks

FPGAs normally consist of the special purpose logic blocks at the boundary of the device for peripheral connections, referred to as I/O blocks (IOBs), which in many architectures are grouped to form a bank that provides supply voltage from a single source to all its I/O blocks. In the FPGA model shown in Figure 2.1, the I/O blocks are placed to all four sides for a better understanding, however, in real chips, they are typically arranged at the bottom. Depending on the device size and architecture, the number of I/O blocks in a bank and the number of banks in the FPGA may vary [41]. The purpose of the I/O blocks is to connect the FPGA to the external world, by providing the input/output functionally through its elements, i.e., bidirectional buffers, MUXes, and registers. Each of the I/O blocks is connected to the programmable interconnects to carry the data/information to be transmitted. For instance, an IOB is configured to define the type/function of the pin used by the device, i.e., input or output, and is usually connected to the physical device pin from one side while from the other side, it is connected to the end of a particular wire of the routing network. In Chapter 4, we will demonstrate that how an unused I/O pin of a block can be exploited as a covert channel to leak sensitive information from FPGAs.

2.1.3 FPGA Design Flow and Tools

FPGA design flow can be categorized into the base-array design flow, i.e., the FPGA device design, or a user application design flow, the so-called, bitstream life cycle. The design and manufacturing process of a base-array or an FPGA device is somehow similar to the ASICs, where

the standard IC development flow is followed. Using commercial electronic design automation (EDA) tools and libraries, the base-array design is articulated, which is then manufactured and primarily tested at the foundry, following the packaging and final test and eventually sent to the FPGA vendor. The flow used for the base-array/FPGA device manufacturing is out of the scope of this thesis, thus we will focus only on the design flow used for the development of application design. The application design in the form of a bitstream configuration file, as a final result, makes use of the reconfigurable hardware when the device is programmed with these configuration bits. To create the bitstream configuration file for an application design or intellectual property (IP), vendor-specific EDA tools can be used.

Since FPGAs provide the reusability of the design which has given rise to the market of IP cores, as a result, a customer or a system-on-chip (SoC) integrator is equipped with the desired design in a shorter amount of time. Thus our experiments in this thesis will be based on the IP core or IP module. Next, we will discuss some of the important steps in the FPGA design flow that are performed for the conversion of IP design specifications to the equivalent bitstream configuration file.

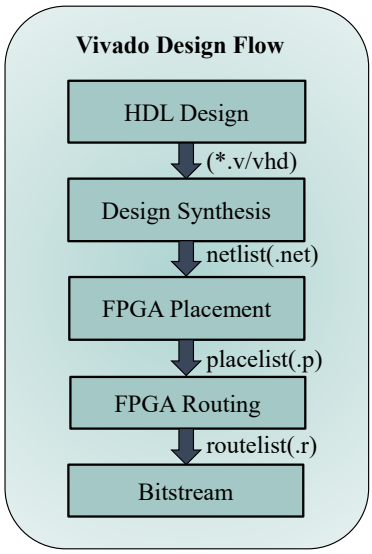

Figure 2.4: Vivado design suite: An FPGA design flow from Xilinx.

The design specifications are typically provided in the hardware description language (HDL) as a source file, which is suited for larger designs, however, a design source could also be the representation of the gates connected to each other, e.g., schematics, which could only be

preferable for smaller designs. The source file is then given to the vendor-specific computer-aided design (CAD) flow for further processing, for instance, Xilinx has its own electronic design automation (EDA) tool flow, e.g., Vivado design suite [42], which performs the following operations in a stepwise manner: design synthesis, placement, routing, and bitstream generation, as shown in Figure 2.4. Each of the steps provide the design representation of different abstraction levels and the design after each abstraction can be optimized according to the area and timing constraints.

The source design is first converted in the synthesis step from the behavioral level, i.e., the algorithmic description, to the register-transfer level (RTL), which is the next level in the hierarchy and consists of data and control paths. Next, the RTL synthesis converts the data and control paths to the equivalent network of logic gates and registers, also referred to as gate-level netlist to realize the Boolean functions. Further, in the logic synthesis, the gate-level netlist is transformed to the circuit level by technology mapping [43, 44], i.e., packing the set of the logic gates according to their feasibility of being mapped on the lookup tables (LUTs), thus a final network of logic blocks is created to implement the actual functionality. In the placement step of a design flow, according to the physical constraints available on the device, the logic blocks are packed and placed on the specific location in the FPGA. In the following step, the router using an efficient routing algorithm first analyzes and reduces the span of the critical routing path and the total wiring lengths, and the design is then routed and fully connected to implement the circuit. The output of this step is the placed and routed netlist, which eventually is converted to the final design representation as a bitstream configuration file by the bitstream generation tool. The bitstream, e.g., binary file, contains all the necessary information about the configuration of LUTs, MUXes, switches, and PIs of the implemented design. Note that the commercial EDA tools are FPGA vendors' proprietary, thus full manipulation of the design after each step is not possible. Therefore, for our experiments in this thesis, we will use the tools from the project *IceStorm* [27] that is an open-source FPGA design flow, which is explained in detail in Chapter 3, Section 3.4.

Security in FPGAs, however, in both the design flows, discussed above, should be taken into account. In the case of base-array design, the security concerns are similar to ASIC or other semiconductor devices [25], such as untrusted supply-chain control or foundry, tools-based tampering, and reverse engineering, which are not the focus of this thesis. Rather, we will discuss the hardware security attacks and potential

defense mechanisms on the IP core, especially regarding hardware Trojans insertion and detection, which is the topic of the next section.

2.2 Hardware Trojans

With the modernization of technology, the size of semiconductor devices has been decreased to nano-scale while the cost for design and fabrication of ICs is increased. It is hard to afford the cost of entire manufacturing processes of ICs for semiconductor companies. To lower the cost and to speed up the development cycle an integrated circuit (IC), outsourcing is being done by most of the companies, e.g., as IP cores, EDA tools, and fabrication to a third-party foundry [45]. This outsourcing can result in the alteration or insertion of a malicious circuit at any of the stages of the ICs development, which in turn can act malevolently or may lead the system towards failure during runtime. Exposures of ICs towards malicious intrusions have adorned serious security threats to military and, financial systems, factories, and household appliances as well. Hardware Trojans, so-called hardware Trojan horses, are malicious circuits purposely inserted into the original design to distract the intended functionality or disclose confidential information by untrusted designers, vendors, tools, communication channels, or semiconductor fabrication companies [46]. The undesired modification of an original circuit can alter the functionality of the circuit, provide a covert channel, or a back door to leak sensitive information from the IC. Modification is generally referred to as tampering, e.g., add/delete certain components to or from the original circuit which may cause reliability/security issues in the circuit [47]. Software Trojan is a malware, resides in the operating system (OS), commonly known as a virus, which could harm the data and may cause an information leakage by getting control on the software of the certain electronic device. Such kinds of Trojans are not permanent and can be mitigated by changing the operating system or using software support. Unlike software Trojans, hardware Trojans once inserted can not be removed from a fabricated device. The infected device will operate properly until the Trojan gets activated. After activation, depending upon the type and nature, the inserted Trojan may harm the device. This intentional damage may include, complete failure of the device, leakage of key information, performance degradation, and denial-of-service (DoS) [47, 48].

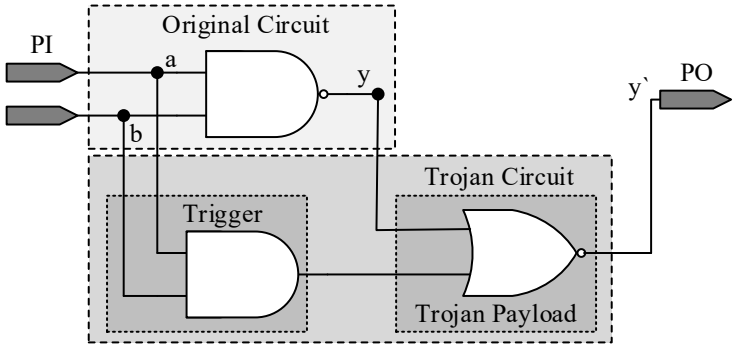

Figure 2.5: Circuit exemplifying Trojan circuit insertion into the original circuit. The Trojan circuit consists of a trigger logic and a payload. Taken from [49].

2.2.1 Hardware Trojan General Structure

The primary intention of the Trojan inserted by any of the malicious entities is to modify the functionality of the original circuit. In the case of functional Trojan, the trigger is usually connected to the inputs of an original circuit which bypasses the original circuit upon activation, hence sending the undesired/modified result generated by the Trojan payload at the output of the circuit. Whereas in the case of non-functional Trojans, the payload of the Trojan circuit might assist the attacker in leaking the secret information from a crypto-circuit, e.g., providing a covert channel, or a Trojan upon activation starts sending a stream of random messages to cause a denial of service attack [5, 50]. The general structure of a Trojan comprising of a trigger and a payload mechanism attached to the original circuit is depicted in Figure 2.5.

2.2.2 Hardware Trojan Sources and Threat Model

A comprehensive list of hardware Trojan sources and related threat models is given in Table 2.1, which is categorized into seven different threat models (A-G) based on the trusted and untrusted parties involved during the IC design and manufacturing process [45].

In threat model A, the source of Trojan could be a third-party IP vendor while SoC developer and foundry are trusted. For the SoC developers, it is nearly impractical to build all required IPs in-house in a short amount of time, so they generally purchase them from third-party IP vendors to integrate on the chip, which may contain hardware Trojan. In threat model B, third-party IP and SoC developers in an IC design flow are trusted while foundry is untrusted. Such IC design companies outsource fabrication to advanced technology foundry which could have

Table 2.1: Sources of hardware Trojans and related threat models (A-G) based on trusted and untrusted parties. Taken from [45, 51].

Threat Model	Hardware Trojan Sources		
	IP Vendor	SoC Developer	Foundry
A	Untrusted	Trusted	Trusted
B	Trusted	Trusted	Untrusted
C	Trusted	Untrusted	Trusted
D	Untrusted	Untrusted	Untrusted
E	Untrusted	Untrusted	Trusted
F	Untrusted	Trusted	Untrusted
G	Trusted	Untrusted	Untrusted

complete access to layout and thus manipulate design by adding or deleting gates. Even the attacker could just modify the transistor parameters, therefore causing potential reliability issues, or could insert Trojans that can cause random failures instead of targeted changes. In threat model C, the third-party IP vendor itself and the foundry are trusted but the SoC developer source is untrusted which can insert Trojan during the design of an IC. These types of Trojans could be added by an insider in the design house, i.e., rogue designer or the compromised third-party EDA tools. The threat model D in Table 2.1 refers to the commercial off-the-shelf (COTS) components used to develop an IC owing to their sufficient production and low cost. All the stages of an IC development in this model are untrusted, therefore, a Trojan can be inserted in any of the development stages. In threat model E, only the foundry is trusted while all other development cycles could be the source of Trojan insertion, e.g., a Trojan can be inserted by an adversary during the design phase, or the third-party IP vendor may intrude IP design. The threat Model F might be considered as a joint model of A and B, where untrusted third-party IP is integrated into the design by a trusted (in-house) designer, and fabrication is done by the untrusted foundry. Some of the fabless IC design companies such as Apple and Xilinx might be the observers of this model. Finally, in the threat model G, the third-party IP vendor is considered as a trusted source while SoC integration and fabrication sources might be untrusted. However, in the perspective of FPGAs Trojans, we will adhere to the threat model C for our contributions in this thesis, see Section 4.1, Chapter 4.

2.2.3 Hardware Trojan Taxonomy

In general, hardware Trojans are categorized into two main classifications based on their components and characteristics [6, 52, 53]. Components-based Trojan classification emphasis on the basic structure and the components involved in the Trojan design while the characteristics-based Trojan classification is about the behavior of Trojan at different levels and conditions, which are explained in the following:

Components Based Trojan Classification

In this type of classification, it has been observed that a hardware Trojan, in general, follows two mechanisms: activation, and actions performed after activation. The activation mechanism of a Trojan is based on the underlying logic implemented to one of its essential parts called *Trojan trigger* while the action mechanism relies on the other part called *Trojan payload*, which performs actions, e.g., malfunctioning, on receiving the activation signal from the trigger circuit. On the basis of its attributes, a *trigger* circuit can have two types: *digital* and *analog*. *Digital* type of triggers consists of logic gates that can be combinational or sequential. Combinational triggers activate the payload of a Trojan circuit when a certain condition is met, whereas a sequential trigger activates the payload when a certain sequence of inputs occurs. In each type of triggers, the adversary would like to develop a logic such as value/sequence that occurs very rarely, so that the Trojan remains inactive during the conventional post-manufacturing testing. A sequential trigger can be synchronous, asynchronous, and hybrid in nature. On the other hand, an *analog* type of triggers includes on-chip sensors and circuit activity which can activate the Trojan when a specific condition is fulfilled. More detailed studies about triggers can be accessed from [6].

A *Trojan payload*, is also classified into two main classes: *digital* and *analog*. A *digital* Trojan payload may consist of one or more logic gates that can modify the value of circuit nodes, alter the memory content, or assist in disclosing confidential information. On the contrary, an *analog* payload may affect the circuit parameters, like performance reduction, power dissipation, and reliability. The components-based classification of Trojans is shown in Figure 2.6.

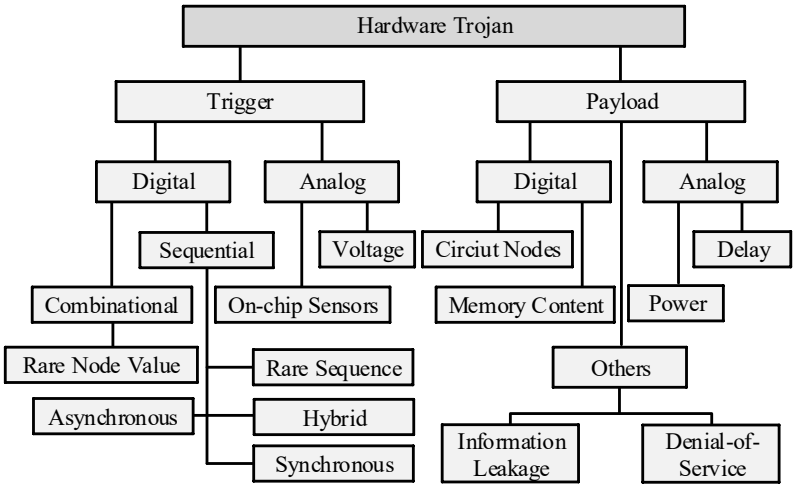

Figure 2.6: Hardware Trojan classification based on components. Taken from [47].

Characteristics Based Trojan Classification

On the basis of characteristics, Trojans can be classified into three main categories: *physical*, *activation* and *action* [52, 54]. The purpose of this classification in the state-of-the-art is to provide a clear concept of Trojans, which could help in the design and development of the Trojan detection techniques in the future against various types of Trojans at different phases of IC development, such as design, fabrication, test, and runtime.

Physical characteristics focus on distribution, size, structure, and the type of inserted Trojan. The distribution of a Trojan could be tight or loose depending upon the way they are located on the chip. The number of gates or transistors, does a Trojan have, corresponds to its size, whereas the structure of Trojan refers to its effect on the layout of the original circuit, i.e., whether it remains the same or gets changed. This can happen when an adversary just has access to the layout, and he aims to insert Trojan somewhere in the original layout. The type of the Trojan can be functional or parametric. Functional Trojan means that the gates in the circuit are added, deleted, or modified maliciously to change the functionality of the original circuit while, in the parametric type attribute of a Trojan, only existing wires or logic of the original circuit are modified.

Activation characteristics are dependent on the components of a Trojan, explained in the previous section, which may cause the Trojan to trigger externally or internally. Externally triggered Trojans make use of the input provided by an external source, e.g., an antenna or a sensor that is inserted maliciously during the design phases. Whereas internally triggered Trojans depend on the logic employed in a trigger which could be

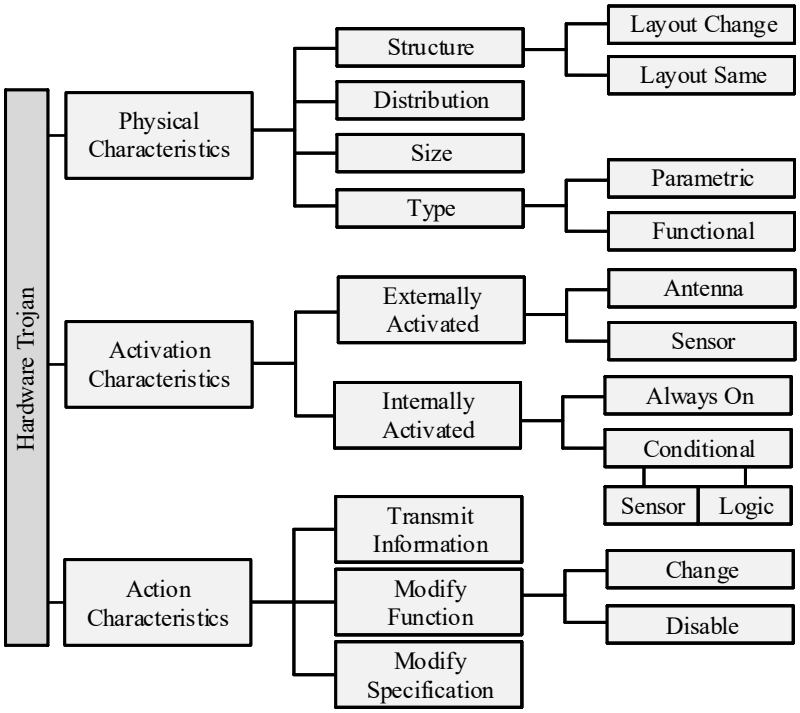

Figure 2.7: Hardware Trojan classification based on characteristics. Taken from [6, 47, 54].

always on or *condition based*. As the name implies, always-on Trojans continuously monitor the activity of the circuit and can disturb the functionality of the original circuit anytime due to its always active nature. These types of Trojans are usually implemented by modifying the exiting wires and transistor geometries etc. The conditions-based internally triggered Trojans, however, depend on the logic-based condition, e.g., a certain sequence of inputs to the logic gates, or an analog-based, i.e., senor input. The third main category is based on *action* performed after activation, where Trojan can transmit information to an adversary, modify specification, i.e., parameters like delay, and modify function which may disable the device or change the desired output [52, 54]. The characteristics-based Trojan classification is shown in Figure 2.7.

Additionally, the advanced taxonomy of hardware Trojans based on the distinct levels of abstraction has been explained in [51]. Figure 2.8 explains the threat of Trojan insertion during each of the design steps by an adversary at different abstraction levels. To develop a Trojan or the corresponding countermeasure, it is important to analyze the design at different abstraction levels, which defines the attacker's control and flexibility to implement a Trojan during the design flow. At the top abstraction level, i.e., system level in Figure 2.8, a circuit is defined by the number of modules and their interconnections to each other, thus

restricting the adversaries to insert Trojans only to modules levels. In the register-transfer level (RTL) abstraction, the design is represented in terms of registers (i.e., flip-flops (FFs)) and set of transfer functions (i.e., combinational logic blocks) which describe the flow of data among registers, where the adversary might have more knowledge about the design to be implemented. Further, at the gate-level abstraction, the design is converted to a network of logic gates and registers with delay constraints instantiated from a technology library to implement the functions. The gate-level design/netlist is then converted to a physical layout of the circuit which describes all the components along with their dimensions and location. At this level, an adversary having access to the layout of the design, can implement a smaller yet effective Trojan, such as an additional wire connecting the original circuit to an I/O port that could be utilized in later stages to leak the sensitive data/information.

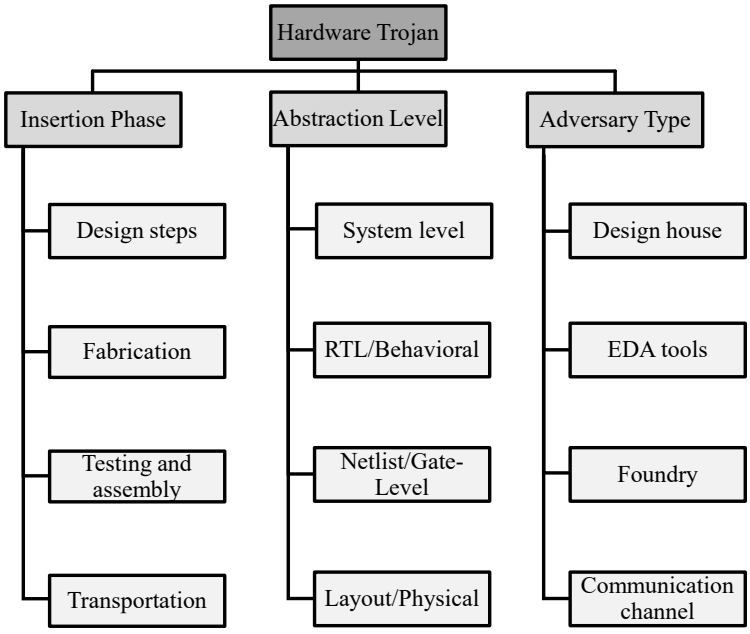

Figure 2.8: Hardware Trojan taxonomy. Taken from [51].

The difficulty level of Trojan insertion increases if we move down in the abstraction levels, for instance, the Trojan insertion by an attacker in the design house, or malevolent EDA tools is simpler than the Trojan insertion in the foundry where an attacker may need to reverse the mask or GDSII file of the design to insert Trojan. Thus the untrusted design house or EDA tools could potentially have a large insertion space in terms of control and access to each of the abstraction levels of the design, whereas, other steps, such as transportation and communication might need a lot of engineering work to insert and hide a Trojan into the target design.

Similarly, the detection rate decreases if the Trojan is designed to be triggered in the lower levels of abstraction, i.e., in the post-synthesis step. The taxonomy shown in Figure 2.8 is quite generic and can be applied to both ASIC and FPGA design flow except for Trojan insertion in the layout when considering the IP-based FPGA design. In the course of this thesis, we would explore the possible countermeasures against the Trojans that are inserted/activated in the last abstraction level, i.e., bitstream level, in Chapter 3. Furthermore, we would demonstrate the insertion of a Trojan by EDA tools during the implementation of a design at lower levels of abstraction such as during the placement and routing step in the FPGA design flow in Chapter 4.

2.2.4 Hardware Trojans in FPGAs

Today, FPGAs are considered a substantial market element in microelectronics, particularly in the embedded systems and smart devices. ASICs are unique custom ICs designed for a specific purpose, i.e., the logic funtion cannot be changed once manufactured, however, they are high-performance devices. The advantages of FPGAs over ASICs are; their flexibility in terms of re-programmability and reconfigurability, the reduced development costs, and the short time to market. Additionally, developers have a limited implementation risk because:

- There is a possibility to revise an inaccurate design
- It is already known that the silicon devices are verified and the underlying technology works properly under particular conditions.
- FPGAs have a simpler and faster design flow process compared to the complicated and design-intensive process for ASICs.

Other advantages include dynamic partial reconfiguration, which gives the flexibility to upload a new design on a portion of the device during runtime.

Reconfigurable computing has a different structural design than ASICs, so the hardware security threats concerning the reconfigurable devices, such as FPGAs, may vary likewise. For instance, an FPGA device consists of a certain configurable logic block, embedded memory, and LUTs to perform the functionality, the designers just need to upload a design in the form of a bitstream for the intended design. In terms of security, one should consider the security of the FPGA device as well as the bitstream configuration file used for configuring the device, where the

former is usually tested and verified by its vendor after manufacturing, therefore it could be considered as tamper-free.

Akin to that, Trimberger in [55] confers the risks of untrusted foundries fabricating FPGAs, where it is challenging to devise an application-specific attack as the foundry would not know the distribution of the different devices to the customers. The FPGA architecture is generic in nature, thus tampering with the base array would have an effect on all the FPGA devices, which can theoretically be exposed by any of the thousands of users, so, letting the cost of testing be compensated through the FPGA end-users. Furthermore, the actual design is unavailable while manufacturing, so there is no chance of manipulating or stealing the design, as the reconfigurable hardware is programmed with the actual design after fabrication. The better the coarseness of programmability, the less an attacker at a foundry would know about a design. The chances that an attacker in the foundry gets a hint about the design depends on the programmability feature of FPGAs, i.e., better coarseness of programmability means difficulties for the attacker to understand design. For instance, in contrast to FPGAs, the attacker in the case of microprocessors, can focus on certain components of the processor because of the knowledge about the program execution and design structure [56]. This is also demonstrated in [8], where King et al. have found that a microprocessor can be hacked to gain high-level access with only a small hardware circuit, which could contain thousand logic gates. Whereas an FPGA consist of an array of configurable logic blocks to implement the application design and the attacker in the foundry would not be able to guess which configurable logic block is used by which part of the application, thus hypothetically makes it challenging for her to attack the FPGA device during fabrication. However, in contrast to this concept, Mal-Sarkar et al. in [53], demonstrated that there is still a diversification of potential attacks that could be achieved by the attacker in the foundry.

Although there are very few chances for the Trojan, inserted into the FPGA device during the fabrication, to be triggered by the user design, an adversary in a foundry could still insert an additional circuit (Trojan) that may be dependent on the user design for activation or independent of the design to induce reliability issues only. Furthermore, Mal-Sarkar et al. have classified FPGA Trojans into two different categories, such as Trojans in FPGA devices and Trojans in IP, also depicted in Figure 2.9, where the former is further divided into two main subcategories: Trojans based on varying triggers and Trojans based on varying payloads. As discussed

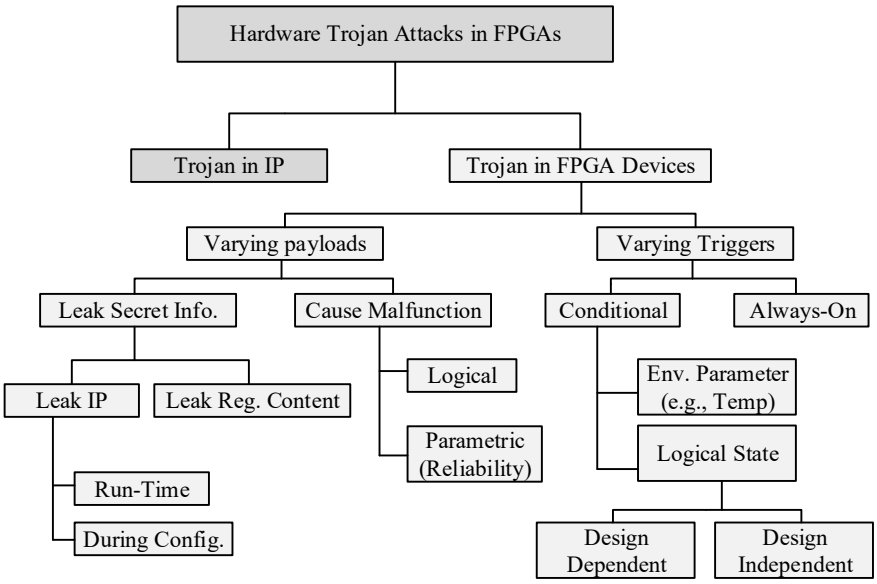

Figure 2.9: FPGA's hardware Trojan taxonomy. Taken from [53].

in Section 2.2.3, the trigger circuit could be fired by a condition or it remains active all the time. The conditional triggers could be based on the logical state or environmental parameters, such as temperature. Based on the FPGA device and the IP used to configure it, the logic-based triggers could further be categorized as *IP-dependent* and *IP-independent* classes of Trojans. The difference between condition-based triggers in FPGAs to the ASICs is that the former could contain a trigger that is IP dependent, while the latter does not fall into this category.

IP-dependent Trojans are the subclass of the Trojans, inserted during the manufacturing of an FPGA at the foundry, where the activation of the trigger circuit to drive the Trojan payload depends on the design that will be implemented in the device. Since, at the time of manufacturing, the attacker would not have any knowledge about the design to be implemented into the FPGA device, it is unlikely that she can implement a Trojan based on the accurate trigger conditions. However, to increase the chances of being activated, an attacker can scatter multiple Trojans over the entire chip, which upon activation, could perform malfunction in several different ways, i.e., the values stored in the LUTs could be modified, the configuration cells can be manipulated to cause routing problems and the embedded memory can be loaded by writing the additional random values. In contrast to IP-dependent Trojans, the objective of *IP-independent* Trojans is to damage the critical modules of the FPGA device, regardless the design to be implemented, such as the digital clock manager (DCM) in Xilinx Spartan-3, Virtex-II, Virtex-II FPGAs, where the

clock frequency can be increased by modifying the configuration parameters saved on its local SRAM, consequently causing the failure of the critical path logic in sequential circuits. Besides altering the logic functionality, a Trojan payload in FPGAs can cause certain reliability issues such as increasing the chip temperatures, delay, and power consumption. Furthermore, the payload could also be characterized to leak the register contents of an IP or the complete IP could be leaked.

This bitstream file is encrypted in modern high-end FPGA families where it is difficult to expose and exploit the design to cause malfunctioning using Trojans, e.g., Xilinx Virtex family. For decrypting the encrypted bitstream, FPGA devices consist of a separate module, which is also responsible for preventing the data and decryption keys from being stolen if a read or write attempt is made, along with limiting the post configuration access to the decryptor. Nonetheless, a Trojan circuit inserted by an adversary in the foundry could be tied to the wires connecting the decryption module to the keys in non-volatile memory, consequently storing the decryption keys to be leaked through side-channels (e.g., [57]) or a covert channel [53]. The scenario is illustrated in Figure 2.10. In addition to that, it has been shown that a Trojan can also be inserted into the CLBs or the memory blocks of an FPGA.

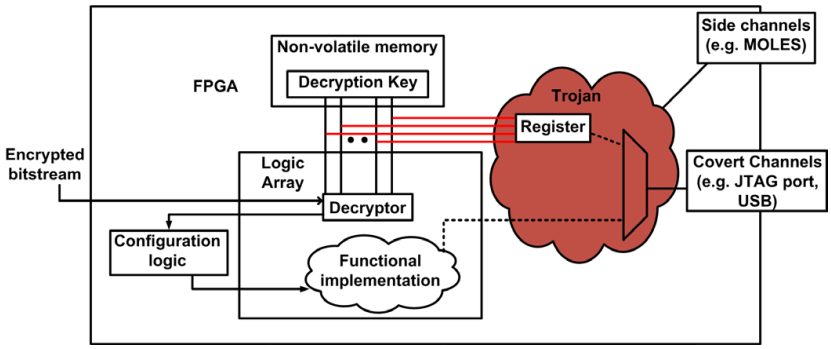

Figure 2.10: Trojan circuit attached to decryption module in FPGA. Taken from [53].

Even, if the encryption module of the FPGA device is not tainted during its manufacturing, there is still a possibility to leak the confidential information from the encrypted IP core[2] implemented on the FPGA device. A practical illustration of this has been shown by Moradi et al. in [58] which confirms that the bitstream encryption keys from an encryption/decryption engine of a device can be extracted during cryptographic operation by using side-channel analysis. In their work, they have demonstrated a successful attack to extract the encryption key from

[2]The terms IP, IP core, and IP design will be used interchangeably in this thesis.

a triple data encryption standard (DES) module implemented on a Xilinx Virtex-II Pro FPGA by employing power analysis techniques, i.e., differential power analysis on the power traces measured during the encryption process. In [59], the attack is further enhanced by using the EM side channel to extract the encryption key even from recent devices such as Xilinx 7 series FPGAs. The FPGA devices from different vendors may have different encryption engines, such as Xilinx, and Intel, however, it has been shown that the devices from Intel are also vulnerable to this attack. For instance, Swierczynski et al. in [60], showed that full encryption keys from Altera Stratix II and Stratix III devices can be extracted by applying the side-channel analysis methods. The Trojan insertion into the FPGA device, however, is out of the scope of this thesis. Therefore, our focus will remain on the Trojan in IP core.

Hardware Trojans in IP Cores

Most of the modern FPGAs used nowadays are (re-)configured with the IP design that is synthesized and converted to a bitstream using the FPGA design flow tools. In general, the IP cores can be divided into three types: *Soft-IP cores*, *Firm-IP cores* and *Hard-IP cores* [36] which are briefly described as follows:

Soft-IP cores: In these types of IP cores, the design is available to be implemented as an HDL representation to the designer, thus flexible to be incorporated into a user design, optimized, and easily implemented on any available FPGA device.

Firm-IP cores: These types of IP cores are also known as semi-hard IPs which are available as a library of high-level functions with certain placement constraints, i.e., the design is available in the form of placed and routed netlist.

Hard-IP cores: Hard-IP cores are usually less flexible than soft and firm IP cores, where the functionality is pre-implemented and is fixed for a specific FPGA family, i.e., adders, media access controller (MAC) functions, and the like. Due to the extensive reuse of circuit designs of FPGAs, there is a possibility that untrusted third parties can inject malicious code into soft IP cores. Additionally, hard IP cores are susceptible to hardware Trojans introduced into FPGA circuitry during fabrication by an untrusted foundry, [61], however, Trojan insertion in hard IP cores

is beyond the scope of this thesis, thus we will evaluate our approaches based on soft-IP cores.

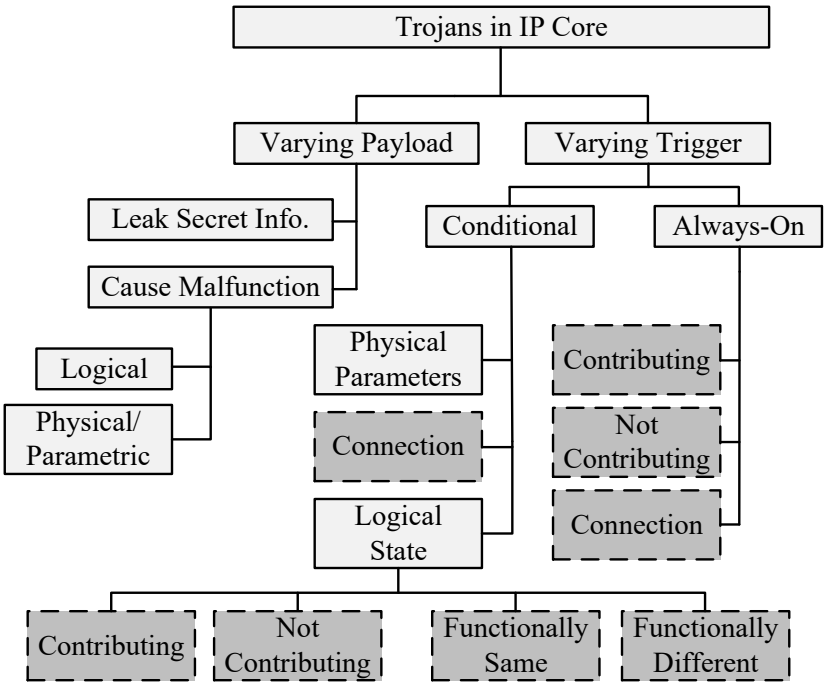

Figure 2.11: Taxonomy of hardware Trojans in IP cores.

We propose a hardware Trojan taxonomy for soft IP cores used for dynamic reconfiguration of the FPGAs based on the different implementations of trigger circuits. So far, the Trojan in IP cores, especially soft IPs, has not been taxonomized. In our classification, the general structure of the Trojan for the IP cores remains similar to the hardware Trojan taxonomy presented in Figure 2.9, however, the trigger implementation could be further categorized that is shown in dark-gray boxes with dotted outlines in Figure 2.11. Following the general Trojan classification, a Trojan circuit is usually implemented with the help of a varying trigger and payload. The trigger circuit may consist of an *always-on* implementation or it may get activated by a specific condition, i,e., *conditional* trigger. However, for IP cores, we further categorize the always-on triggers based on their *contribution* to the output of the original circuit, i.e., whether or not the triggers' signals of always-on Trojans take part in the output of the original circuit. For example, if the trigger signals change the behavior of the output of the circuit, they are termed as contributing, otherwise, they are referred to as not contributing.

A conditional trigger, on the other hand, like in general Trojan classification, may depend on the conditions such as physical parameters

based-conditions and logical state-based conditions. A physical parameter based-conditional trigger relies on the input from certain physical parameters, i.e., temperature, to get activated. The logical states, in general, are the logic conditions that drive the payload upon activation. However, they can be further divided into four new sub-classes based on their effect on the overall *functionality* and their *contribution* to the output of the circuit. For instance, a trigger could be implemented in a way that, during its inactive state, the functionality of the original circuit remains the same or it gets changed due to the extra logic. Similarly, the additional logic of a trigger could contribute to the output of the circuit to mimic an authentic logic, or it can be a dummy logic that does not contribute to the output. A *connection-based* conditional trigger, which is a new class introduced in this thesis, determines if the trigger is actually connected to the original circuit or disconnected temporarily to evade any testing mechanisms and gets activated after re-connection. Both always-on and conditional triggers can further depend on their *connection* to the original circuit for activation. Such kinds of triggers can be inserted during the routing stages and can be connected in the latter stages for activation (cp. Section 4.2). We consider the general payload implementations in IP cores, that are similar to the payload implementations in FPGA devices/ASICs, where a payload on activation can cause malfunctioning or assist the attacker to leak secret information. In literature, various trigger and payload implementations along with their corresponding countermeasures at different abstraction levels have been proposed for Trojans in IP cores [62], which we will discuss in the following section.

2.3 Related Works

In general, it is harder to attack a circuit on an FPGA in a foundry compared to an ASIC, as the actual functionality of the FPGA is only determined at runtime, when the fixed-function part and the dynamic configuration are combined to form the circuit. Attacks on the fixed-function part of the FPGA are closely related to ASIC attacks and out of the scope of this thesis. Thus, Trojan attacks and defenses on the dynamic configuration aspect of the FPGAs will be explored in this thesis. There could be several ways to insert and implement hardware Trojans in IPs, depending upon their design, characteristics, activation, and actions. A variety of hardware Trojans submitted in the CSAW Embedded Systems Challenge [63–65] held in 2012, have been detected using functional testing, power analysis, and direct analysis of bit-file. A multi-faceted technique to detect

Trojans in FPGAs has been demonstrated in [66], where the authors develop a Trojan detection framework leveraging testing techniques to identify different Trojans introduced in CSAW Embedded Systems Challenge.

FPGA devices can be compromised by utilizing malicious hardware blocks, like reconfigurable blocks, sometimes referred to as IP blocks. Such malicious IP blocks can be employed to gain access to unauthorized parts of memory and modify its contents. However, a security wrapper for IP cores proposed as a countermeasure in [67], can be incorporated by the FPGA system-on-chip (SoC) designers to secure the system against these attacks. Also, counter-based ring oscillators have been widely used in ICs to make the detection of Trojans easier due to their oscillation. A ring oscillator (RO) follows the phenomena of oscillation mainly due to its intuitive logic. The number of components and the size of the circuit derive the oscillation frequency which may vary with the addition or deletion of the components in a circuit. This feature is used for non-destructive testing to detect different modifications in a hardware design by integrating ring oscillator(s) (of various lengths) [68]. The extension of this RO has been made in [68] known as transient effect ring oscillator (TERO) digital sensor which is more sensitive to intrinsic noise, thus providing better results in Trojan detection in FPGA implementation of advanced encryption standard (AES) algorithms [69].

Design-time hardware Trojan detection in IPs may roughly fall into two categories, such as dynamic and static detection. In the dynamic detection methods, the design is usually simulated to check the behavior of the logic circuit using different test patterns. However, the Trojan is designed with the intention to evade the simulation testing, thus an adversary could choose a trigger condition that is activated very seldom. Also, the verification team would have no idea about the type and location of the Trojan circuit, it would be hard to generate the patterns for huge design space to trigger the rare states [62]. On the other hand, static detection methods do not rely on circuit simulation but use the information related to the Trojan, i.e., structural information. In literature, both methods have been explored to detect the Trojans at register-transfer level (RTL) or gate level. King et al. in [8] presented three powerful attacks, i.e., privilege escalation attack, shadow mode/backdoor attack, and service-based attack, by implanting two malicious circuits in the Leon3 processor. The malicious logic implemented in both the circuits had a low impact on the size of the processor in terms of the logic gates and the number of lines in a VHDL code. Since it is difficult to verify each gate and line of

code for such a large design, so the malicious circuitry would likely bypass the traditional testing methods. However, Hicks et al. presented an algorithm, so-called unused circuit identification (UCI) in [12], to detect the malicious circuits in the design, i.e., in [8], which are then removed by the BlueChip model. Primarily, the UCI is a design-time testing algorithm that identifies the parts of the circuit that do not contribute to the output during simulation-based testing and flag them as suspicious. The general *UCI* algorithm's steps are shown in Figure 2.12.

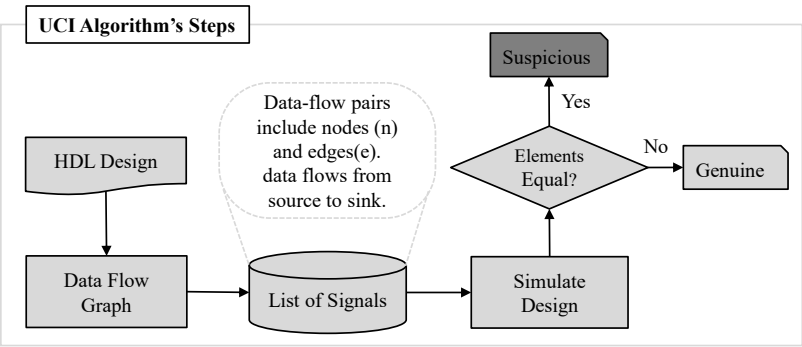

Figure 2.12: Steps of unused circuit identification (UCI) algorithm to detect malicious hardware. Derived from [12].

The HDL design is first converted to the data-flow graph to generate the list of signal pairs, i.e., data-flow pairs, where the nodes represent wires (signals) and the edges refer to the data flow between signals, in which the data flows from a source to a sink. Then, in the design verification through simulation step, each of the signals that do not alter the data values of the data-flow pairs during the flow from source to sink is obtained by applying an inequality check on the delay values of the source and sink. If the delay values of each of the elements in data-flow pairs are not equal, the circuit is considered benign, however, if the delay values are equal, the circuit between signal pairs is removed and placed into the suspicious list and a wire is used as a replacement. Subsequently, to beat the *UCI* approach, Sturton et al. [10] implemented a stealthy malicious backdoor in a Leon3 processor, where no pair of the dependent signals are equal at design time, under non-triggered circumstances, thus can not be marked suspicious by *UCI*. The circuit performs malfunctioning upon activation when a specific pattern, e.g., a trigger input is received, which is concealed at the design time [10]. Even though these types of Trojans may evade *UCI*, exhaustive simulation and formal verification would have a tendency to find them, as the infected circuit would not behave functionally and formally equivalent to the original design.

Zhang et al. in [13] proposed a dynamic detection technique called *Veritrust* which investigates the verification corners to automatically identify the trigger inputs of the Trojan circuit. Essentially, the inputs of the circuit that are not triggered under normal conditions are marked as redundant and thus suspicious. The technique is comprised of two stages, the *tracer*, and the *checker*, which is also shown in Figure 2.13. The *tracer* mechanism traces the redundant inputs of the circuit under non-triggered conditions through verification tests, i.e., sum-of-products (SOPs) or product-of-sums (POSs) of flips flops and primary outputs in a design netlist. The *checker* then applies the functional simulation to unactivated SOPs and POSs to realize the redundant inputs, which are then flagged as potential Trojan trigger inputs. However, to minimize the area overhead, the functional verification is not applied to the entire circuit, therefore, it may result in a high false-positive rate, e.g., the benevolent input is flagged incorrectly as a trigger input.

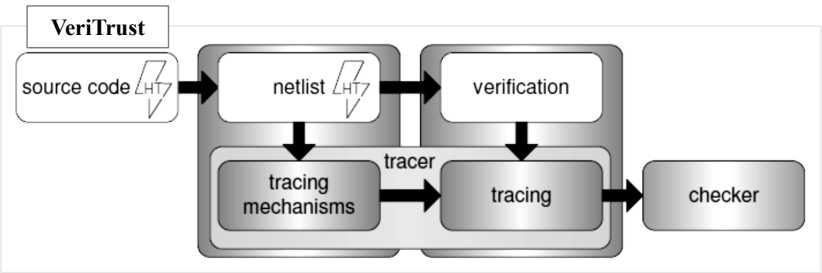

Figure 2.13: Veritrust: Unused input identification framework for verification. Taken from [13].

Furthermore, to detect the stealthy malicious circuits introduced in [10], Waksman et at. presented a static method of detecting hardware Trojan that leverages the Boolean functional analysis of a circuit to detect Trojans instead of relying on a design-time validation test suite through simulation [14]. In essence, the trigger inputs of a Trojan, in general, have no or a weak effect on the output of the circuit to stay unnoticed during verification. Exploiting this fact, the authors have anticipated the influence of an input signal on output by using a Control Value *(CV)*.

The working flow of *FANCI* demonstrated in Figure 2.14 is described as follows: Initially, the truth table for a fan-in tree is generated which contains the information of each wire *(w)* in a gate of each module in a design. Then, the control values are determined for each of the input columns in a truth table *(T)* to get the Boolean difference of the two

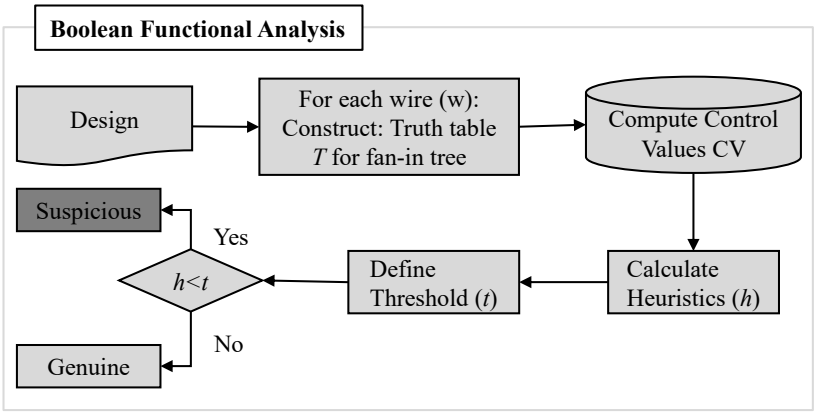

Figure 2.14: Boolean function analysis for nearly-unused circuit identification. Derived from [13].

functions. Once the vector of values for each gate's output is obtained, the heuristic on this vector is applied to make sure whether the wire could be marked as suspicious for further inspection. The heuristic (e.g., median) is computed as a heuristic function *(h)* and the threshold *(t)* is set between zero and one to realize the values of *(h)*. If the value of the heuristic function of a given wire is less than the threshold, the wire is labeled as suspicious, otherwise, it is deemed genuine. Since the hardware Trojan problem is the manifestation of the arms race between the attacker and the defender, where each party tries to innovate the corresponding methodologies. Hence, Zhang et al. in [70] presented yet another hardware Trojan attack that evades the state-of-the-art dynamic and static detection mechanisms [10, 13, 14]. To do so, the authors proposed a methodology named as DeTrust [70], in which the Trojan is implemented in a way such that, the trigger logic is carefully distributed between various combinational logic blocks to evade the *FANCI* approach [14]. Whereas, to dodge *Veritrust* [13], the Trojan triggers are concealed into various sequential levels which are also combined to the original functional logic of the circuit.

However, the problem with the above-mentioned techniques for Trojan detection in an IP core is that they are applied only at the RTL/gate level, thus any malicious insertion by an adversary after this step in a design flow will leave a question on their applicability. Furthermore, none of the techniques will detect always-on Trojans which do not affect functionality but intend to decrease the reliability of the circuit, i.e., RO-based Trojan. Also, most of the techniques often require a golden model to detect the Trojans which is the vendor-proprietary and might not be

available for verification purposes, e.g., in the case of the firm and hard-IPs. Nevertheless, Oya et al. in [71] presented a score-based classification technique that does not require a golden circuit to detect the hardware Trojans. In the proposed methodology, the Trojan nets are distinguished from the original nets of the design based on an incremental metric applied to the Trojan features. The technique showed promising results for the Trust-HUB Trojans [51, 72], however, a considerable amount of processing time is required for larger designs. Also, the technique is valid for only Trust-HUB Trojans, while the new unseen Trojans would likely escape this approach.

Another golden-reference free Trojan detection method on a gate-level netlist has been presented by Hassan Salmani [73] that is based on the testability measure of the circuit where the testability is observed in terms of controllability and observability analysis derived from numerical values. A *K*-means clustering algorithm [74] is applied to get the list of Trojan signals and genuine signals to distinguish between the Trojan-inserted and Trojan-free designs. The technique is capable of detecting both always-on and condition-based Trojans, however, it does not apply to the Trojans that are inserted or activated in the post-synthesis steps of the design flow, e.g., bitstream-level Trojans [75].

An FPGA bitstream configuration file containing particular design information may also be susceptible to hardware Trojans. The three major entities during the bitstream generation processes that may insert malicious logic/functionality in the legitimate design are a) a design house or a malicious designer in the design house, b) compromised electronic design automation (EDA) tools, and c) malicious communication channels, i.e., via man-in-the-middle (MiM) attacks. Attacks on the FPGA configuration during the design step, can, e.g., be performed by compromising the EDA tools which are used to synthesize the design. This attack can be carried out either by actually modifying or replacing the tools themselves, as is described for instance in [76], or as a post-processing step, where the authors investigate the possibilities and limitations of direct bitstream modification attacks. For an attacker, compromising EDA tools can be very attractive, as potentially a higher number of designs can be compromised in one attack.

Recently, Duncan et al. [77] classified the different threats to a bitstream at various stages during the FPGA design flow. In the first stage of this taxonomy, the threat to a bitstream generation by the design house

and third-party IPs are categorized into malicious and non-malicious intent, where the latter refers to tools-induced vulnerabilities. The introduction of vulnerabilities or malicious circuitry through design tools could be more interesting for the attacker due to the simplicity of the attack and the implanted malicious circuitry can also be more devious. The activation of dormant logic inserted at any point could force the device during operation to go to certain undesired states such as the denial of service, change of functionality, and secret information leakage.

In 2013, Chakraborty et al. [75] presented a mechanism to taint the FPGA bitstream that was first to insert a Trojan directly in the bitstream. The Trojan is inserted in an unencrypted bitstream using an add-on program that modifies the bitstream configuration file. Based on their connectivity to the original circuit, two types of Trojans have been proposed that can be inserted: A type-I Trojan has no connection to the original circuit (hence non-functional Trojans) and is inserted into the free resources of FPGAs. A Trojan circuit consisting of ring oscillators has been implemented to increase the temperature of the FPGA, which causes reliability issues and early aging. Such kinds of attacks are denial-of-service (DoS) attacks. The success of the attack relies on the availability of resources in the proper locations of the FPGA. However, such a Trojan circuit may be difficult to insert if a) the bitstream is encrypted or b) the unused FPGA resources are filled up with dummy logic [78] in a bitstream. Trojans that have a connection to the original circuit are considered type-II and require sufficient design knowledge to implement, hence not contemplated by the authors.

Nonetheless, if we recall the threat model presented in Section 2.2.2, where the design house and IP vendors, are considered trustworthy, but the EDA tools maintained by the vendors are undermined, resulting in an enhanced power to the attack by infecting a bundle of designs in just one go. Undermining EDA software tools also resembles the compiler attack introduced by the authors in [79], where the back-end tools activate the malicious activity. One such kind of attack for reconfigurable hardware has been presented in [26] by introducing a stealthy "malicious LUT" hardware Trojan inserted during design flow that employs a two-stage mechanism of insertion and activation. The Trojan remains dormant throughout the design phase and activates only when the bitstream is written, thus circumventing design-time verification techniques, such as [12–14]. The attack is explained in more detail in the next chapter, Chapter 3, Section 3.1.

Chapter 3

Proof-Carrying Hardware Versus the Bitstream-level Hardware Trojans in FPGAs

Hardware Trojans detection in a bitstream, or the Trojans that only activate in an FPGA bitstream, is quite challenging due to the unavailability of the bitstream formats as most of them are vendors proprietary. A malicious lookup table (LUT)-based Trojan attack that is injected and triggered by the field-programmable gate array (FPGA) design flow is an example of such an attack which we will explain in detail in Section 3.1 of this chapter. Next, the overview and a short background of proof-carrying hardware (PCH) concept is given in Section 3.2, followed by the comprehensive discussion on the proposed bitstream-level proof-carrying hardware (PCH) for Lattice iCE40 FPGAs that is capable of effectively detecting the stealthy malicious LUT hardware Trojan in Section 3.3, the description of the Lattice iCE40 FPGA, and associated

open source tools flow used to evaluate our approach against malicious LUT attack under different attack scenarios, is described in Section 3.4 and 3.5, respectively, followed by the discussion about our approach in Section 3.7, and in Section 3.8, we will conclude this chapter.

The work described in this chapter has been presented in *International Symposium on Applied Reconfigurable Computing* (ARC) in 2019 and published in [80].

3.1 Malicious LUT Hardware Trojan

In the recent past, Krieg et al. [26] presented an FPGA design flow attack that has been carried out by subverting the vendor's provided electronic design automation (EDA) tools. Their attack works in a two-stage manner by using compromised design tools that basically adds a second trigger to the Trojan design, which is tied to specific steps in the FPGA design flow, as depicted in Figure 3.1 (a). Their stealthy Trojan relies on compromised design tools: First, the front end synthesis tool injects the Trojan into a user design (Figure 3.1 (b)), and then the back end synthesis tool activates (triggers) it when writing the bitstream configuration file, and only then (Figure 3.1 (c)). Essentially, the malicious circuitry, i.e., Trojan, is first inserted by the synthesis tool while reading the design by employing the search and replace method to the hardware description language (HDL) code of the design. For instance, a certain module or just a few lines of the module can be replaced with a new module or code. After that, the design along with the malicious added circuit/Trojan by the front-end of the synthesis tool is synthesized and optimized by the synthesis tool which results in a gate-level netlist of the design, that can be simulated to verify the correctness of the design. At this stage, the Trojan is inactive, so the compromised design turns functionally and formally equivalent to the original circuit, thus the functional simulation verifies it as a correct design. However, during the placement step, the malicious back-end searches for the specific patterns based on some properties of the Trojan trigger that were injected by the malicious front-end tool. If the trigger cells match the properties, they are reconfigured to turn on the malicious logic. Hence the Trojan is only activated by the back-end tool when writing the configuration bitstream of the design to the file.

The authors, for illustrative purposes, have demonstrated a privilege escalation attack on an instruction decoder of a CPU where for one particular instruction the privilege verification is influenced. As a result, the instruction decoder executes that privileged instruction with no superuser

rights when the Trojan signal is high, i.e., when the output of the malicious lookup table (LUT) is high. The implementation of a smart trigger using a malicious LUT, which remains dormant throughout the design phase and activated when the final bitstream is written, helps the Trojan to evade all the design-time verification techniques relying on functional verification or rare events occurrence.

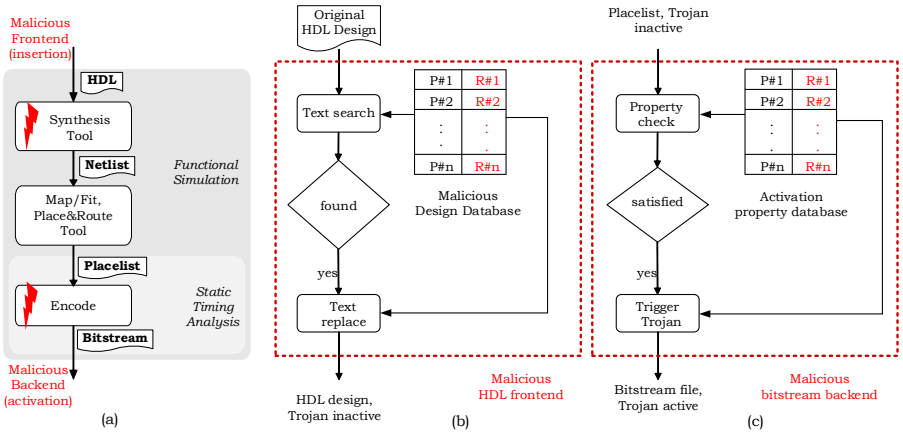

Figure 3.1: (a) depicts the function principle of the attack. In (b), a malicious front end injects malicious HDL code into an original design based on pattern matching. Whereas (c) shows a malicious back end which activates the attack by looking for the previously inserted cells and altering them. Taken from [26].

Also, the activation characteristics of the injected Trojan can still comprise a classical trigger-payload pair, which can then, e.g., be activated at operation time. The novelty of the approach lies in the fact that the infected circuit is functionally equivalent to the hardware specification during post-implementation simulation and testing, as the Trojan is dormant after insertion. This Trojan can thus circumvent all state-of-the-art detection techniques that rely on identifying unused, nearly unused, or redundant inputs or portions of the circuit, such as unused circuit identification (UCI), VeriTrust and functional analysis for nearly-unused circuit identification (FANCI) [12–14].

It is believed by the authors in this work and the prior attack [75], explained in Section 2.3 of Chapter 2, that due to the lack of a verification mechanism for the bitstream configuration file it would be enormously difficult to counter such attacks. The only possibility to detect this stealthy Trojan pre-configuration, i.e., before the FPGA is configured with the design, is to analyze the configuration bitstream itself, as also the authors point out. However, we propose a bitstream-level verification technique using proof-carrying hardware (PCH) which effectively detects

the stealthy two-stage malicious LUT hardware Trojan attack presented in [26].

3.2 Proof-Carrying Hardware

Inspired by the proof-carrying code (PCC) technique [81] which was intended to quickly verify the executable code of an application before running on to the system by the host, Drzevitzky et al. [82] proposed the proof-carrying hardware concept for the verification of an intellectual property (IP) module before a system integrator incorporates it to a reconfigurable device. Their approach considers the end-user of the IP module or the system integrator who integrates an IP into the SoC device, as the consumer, who is interested in instantly verifying the desired functionality of third-party IPs, and the one who produces an IP and ships to the consumer is considered as the producer. Moreover, both sides come to terms with certain formal properties that need to be met, i.e., the consumer along with the design specifications, specifies the list of safety/security properties that should be satisfied by the IP module generated and supplied by the producer, before using the IP for the required objective. Fundamentally, the producer is obliged to construct the set of proof together with the creation of an IP module and send it to the consumer who then could quickly validate the proof.

The general overview of a bilateral agreement between the producer and the consumer within PCH is shown in Figure 3.2. First, the producer receives the design from a consumer, specified mainly in hardware description language (HDL), synthesizes it for the given target FPGA device, and generates a bitstream configuration file. In the next step, the miter is formed using the extracted logic functions from a reversed bitstream and the proof is generated which is then attached to the bitstream. At this point, the producer does the main job of developing the IP module and the proof of correctness and sending the resultant proof-carrying bitstream (PCB) to the consumer. The consumer, on the other hand, receives the PCB and extracts the bitstream and proof, and using the original specifications and trusted tools also forms a miter function and proof, which is then used to compare against the received miter and proof. So, the consumer only checks the received proof against his/her in-house generated proof and decides whether to IP module is functionally correct or not. If the module is functionally correct, i.e., passes the miter and unsatisfiability checks, the hardware is configured with the received bitstream,

otherwise it is rejected. In this way, the consumer would be able to validate the safety/security properties even from an untrusted producer of the IP in a short time, thus having theoretical and formal reasons to accept or reject the IP for reconfigurable hardware.

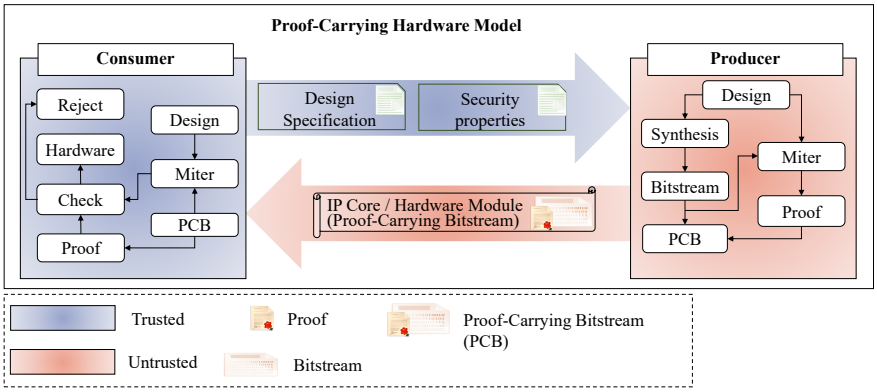

Figure 3.2: Overview of two-party agreement model of PCH between consumer and producer. Derived from [82, 83].

PCH considers third-party IPs that are integrated into FPGAs in a bitstream configuration format, as this is the lowest possible abstraction in reconfigurable hardware, which explicitly excludes the closed-source vendor EDA tools from the trusted base. To verify the third-party IPs, PCH uses automatic formal verification techniques that are easy to retrace, so as to enable the recipient of module and proof to perform a lightweight verification with the full power of the initial one. Principally, PCH has been evaluated using abstract and virtual FPGAs due to the unavailability of bitstream documentation to be used as a prototype for FPGA providing companies. Therefore, Wiersema et al. [83] evaluated and implemented PCH on a fine-grained virtual fabric, where they demonstrated and experimentally evaluated PCH at the bitstream level of virtual bitstreams for an overlay placed on a real FPGA.

We apply the idea of [82] to the tool flow from [26] to use PCH to detect the stealthy hardware Trojans for the lattice semiconductor iCE40 family of FPGA [84] using mainly tools from the open-source flow IceStorm [27]. Our approach differs from state-of-the-art Trojan detection techniques where comparison or checking is done at the *register-transfer level (RTL)* or *netlist* level to detect stealthy Trojans. While the attack we counter will not be caught by such techniques, as the activation is done at the last stage, i.e., while writing the bitstream, our bitstream-level PCH scheme is able to detect Krieg et al.'s stealthy Trojans using an open-source tool chain.

3.3 Bitstream-level Proof-Carrying Hardware for ICE FPGAs

As discussed in the previous section (cp. Section 3.2), so far, the PCH concept has made progress to be accessed on abstract and virtual FPGA or a virtual bitstream, however, the recent hardware Trojan attack on a real bitstream of a commercial FPGA [26] necessitates a powerful mechanism to detect any manipulations made during production time. In this work, we present a PCH prototype directly for the bitstream of a real FPGA by leveraging reverse engineering-based open-source design tools from the IceStorm project [27].

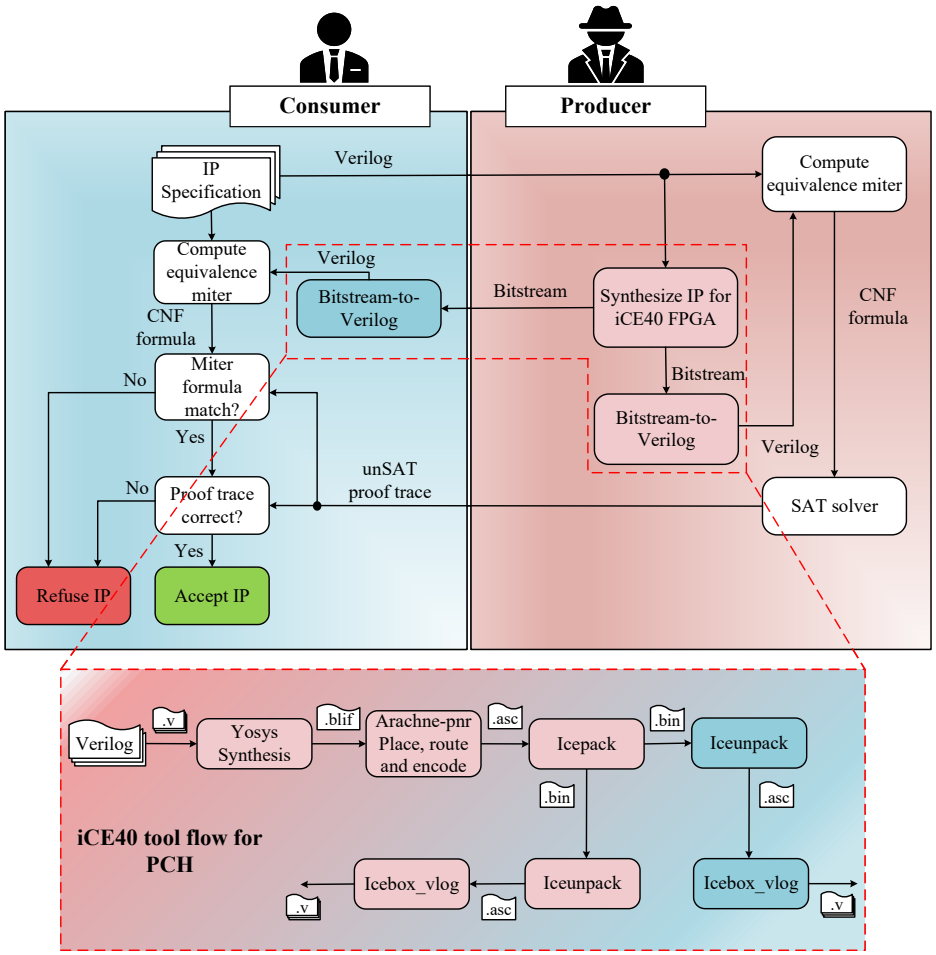

Figure 3.3: PCH tool flow for bitstream-level verification in iCE40 FPGAs.

Figure 3.3 outlines the steps performed in our PCH scenario, derived from the steps described in [83]. Each of the steps is briefly discussed in the following: First, the consumer specifies the functionality of the IP module as well as the safety/security specifications and sends it to the producer. As a simplifying assumption for our prototype, the design

specification is provided as Verilog source code and the security specification is agreed upon in advance as demanding the full functional equivalence between the (golden) Verilog source and the circuit represented in the final bitstream, thereby detecting any Trojans that alter the functionality.

In the next step, the producer synthesizes the IP module for the target platform, in our case the iCE40 FPGA which involves the design synthesis, placement and routing, and then the generation of a bitstream configuration file. Each of the steps performed at the producer side along with the tool used is depicted in Figure 3.5. The employed PCH property demands functional equivalence between specification and implementation of the circuit to ascertain the correct behavior; which is formally verified using that miter. So in the following step, the producer re-extracts the Verilog from the bitstream to combine it with the original specification into a miter function in conjunctive normal form (CNF) form. The miter is a function, which for the same inputs of the circuit, generates both specification and the implementation as the output and then matches their result pairwise with XOR gates for equivalence. If we consider $S(\underline{x})$ as the specification function and $I(\underline{x})$ as the implementation function of a circuit input $\underline{x}$, respectively, then the miter function is formed by adding (OR-ing) all the outputs of XOR gates together [82]. The equation (4.1) shows miter (M) function as :

$$M = \sum (S(\underline{x}) \oplus I(\underline{x})) \tag{3.1}$$

Figure 3.4 shows the structure of the miter function for equivalence checking of combinational and sequential circuits.

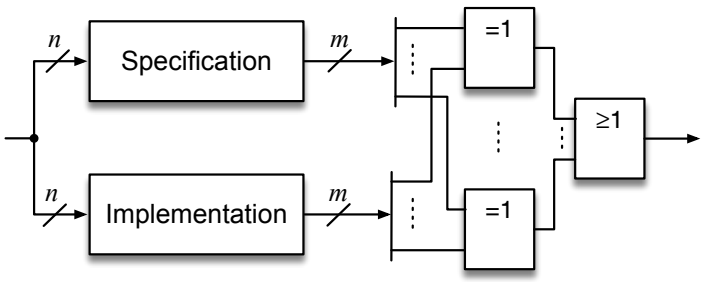

Figure 3.4: Miter function for functional equivalence checking. Taken from [85].

The CNF formula is proven to be unsatisfiable by a boolean satisfiability (SAT) solver, which proves functional equivalence between specification and implementation. This works for combinational and time-bounded sequential circuits; for unbounded ones, or ones with no practical bounds like triggers using long-running counters, more advanced proving techniques like induction-based proof-carrying hardware flow are needed [85], in which the inductive invariants act as a testimony of the formal verification. Thus for the counter-based Trojans, also known as a time bomb [6], this provides an effective set of security guarantees, hence detecting any malicious circuits that are based on sequentially triggered Trojans. The resulting proof trace together with the bitstream is then sent to the consumer, who first converts the received bitstream again to Verilog using the trusted tools (Figure 3.5) and then formulate the miter with the reversed Verilog in order to compare this miter conjunctive normal form (CNF) with the one that is the basis of the proof trace. The consumer can only be sure that the provided proof trace is actually about the specified IP module, if the miters match. In case of a match, the consumer verifies the proof using the proof trace. If this step also is successful, the consumer configures the FPGA with the bitstream. If any of the two checks fail, the consumer indicates the failure through its user interface on the PC, about the rejection of an IP module.

3.4 Tool Flow for ICE FPGAs

Project IceStorm is an open-source project by Wolf and Lasser that provides tools to create and manipulate or analyze bitstreams for Lattice iCE40 FPGAs [27]. The project aims at reverse engineering and documenting the iCE40's bitstream format, and together with other open-source tools they have defined a completely open-source tool flow for the iCE40 FPGAs. Lattice FPGAs, and especially the iCE family, can be used for smaller circuits for the verification of the design. Figure 3.5 shows the parts of the iCE40 tool flow that we used to implement the PCH approach. Here, *Yosys* is used to synthesize the hardware description language (HDL) description of the design. The output of Yosys is a synthesized (and optimized) netlist in Berkeley logic interchange format (BLIF), that is then given to the place and route tool, *Arachne-pnr* [86], which after placement and routing, encodes the placed and routed design/netlist into a text file american standard code for information interchange (ASCII). To generate the binary file, e.g., a bitstream configuration file, we use the *IcePack* tool. To reverse convert the bitstream we use *IceunPack* which

converts the binary file again to a text file (.asc). Furthermore, we use the *icebox_vlog* tool within the *IceBox* tool to convert this text file again to HDL, i.e., structural Verilog which we use to form the miter function.

In Figure 3.5, the iCE40 tool flow used by the producer site, shows that the producer is responsible for the development of an IP using the complete design flow, along with the proof, whereas, the consumer only needs a tool or two from the complete design flow, e.g., *IceunPack* and *icebox_vlog*, which convert the bitstream into the Verilog representation, that is then used for checking the combinational equivalence checking (CEC) to the original specifications, thus clearly indicating towards the prompt verification of an IP module at the consumer site.

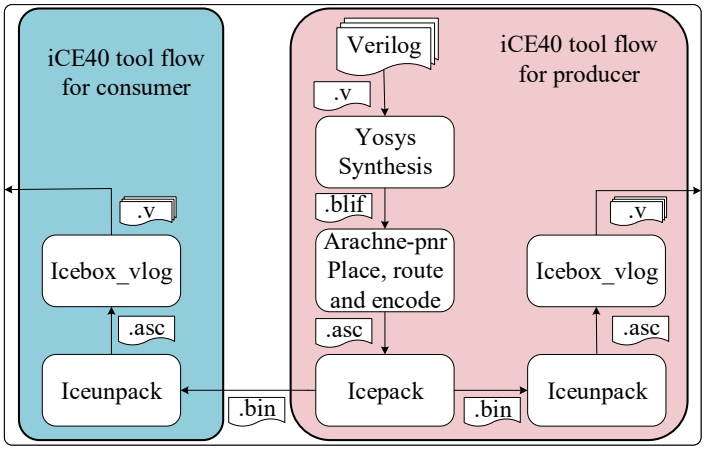

Figure 3.5: iCE40 tool flow used for PCH.

3.5 Attack Scenarios

As explained in Chapter 2, Section 2.2.4, IP modules obtained from third parties can be maliciously infected. We explain our threat model by discussing three different possible scenarios and show that in all three scenarios of attack, our bitstream-level PCH approach identifies the malicious intrusion and notifies that the implemented design is not functionally equivalent to the specified design. In the following subsections, we highlight the resiliency of the verification technique against attacks from various parties; Table 3.1 summarizes the details.

3.5.1 Scenario: 1

In this scenario, as illustrated in Figure 3.6, we assume that the adversary is either the vendor of an underlying IP core used by the producer to make their own core, design house or an employee of the producer in a

design house who deliberately inserts the stealthy Trojan into the original design specification received from the consumer, while the EDA tools and the transportation of IP modules in this case are not compromised. The Trojan would then remain dormant through all internal validation steps performed by the producer to be only activated upon writing the final bitstream.

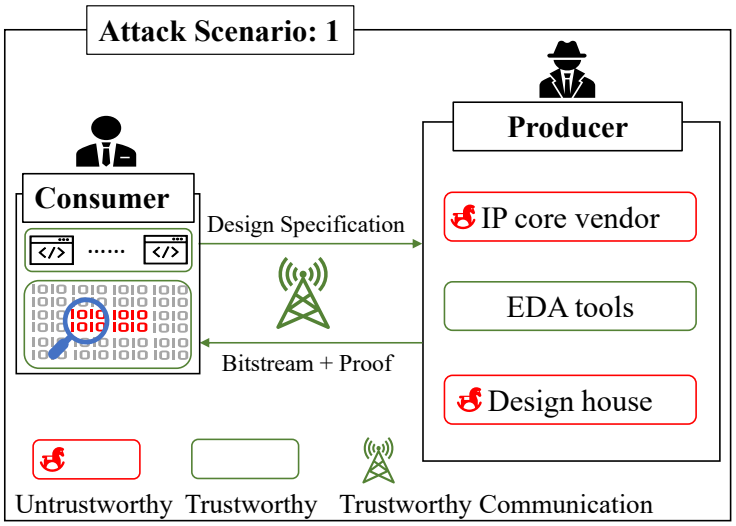

Figure 3.6: Attack scenario 1

Since this bitstream forms the basis for the remaining PCH verification, the producer would subsequently compute the miter function using the compromised implementation with the active Trojan, as discussed in Section 3.4. There are two possibilities now:

1. The producer uses the consumer's original specification. This would lead to a satisfiable miter function, as the design extracted from the bitstream has altered behavior, and hence the proof creation step would fail, meaning the producer would not be able to send a proof-carrying bitstream (PCB) to the consumer. Remark: Were the miter not satisfiable in this step, would the alteration of the Trojan be deemed harmless, since that would essentially mean that the inclusion of the Trojan does not violate the previously agreed upon property (functional equivalence in this case).

2. The producer uses a compromised specification with activated Trojan instance. With this specification, the producer would be able to create a proof of conformity of their core with the specification. The consumer, however, would compute the miter using their original copy of the specification, which cannot be compromised, which

would lead them to a different miter compared to the producer. They would thus not accept the received proof, as the miter comparison step would fail.

Our proposed PCH approach would thus alert the consumer during the verification of the bitstream against the Trojan's presence in every possible case in this scenario, as long as the (activated) Trojan violates the property. Whereas, for the same scenario, the Trojan remains undetectable when applying unused circuit identification (UCI) approach [12] because it identifies the parts of the circuit that do not contribute to the output, based on functional simulation, and hence marks them as suspicious. Conversely, the improved version of the trigger implemented in[26] is masked in a way that the output is derived from almost every input, thus the Trojan can dodge the UCI approach. Also, the Trojan in the scenario is assumed to be implemented by an antagonist as a smart trigger circuit, like in [26], that uses all of its inputs as a trigger logic, thus none of the inputs will be marked as redundant by the VeriTrust approach [13] which targets the identification of unused, potentially redundant, inputs by maintaining a record of their activity with the help of a tracer, besides functional simulation. Similarly, unlike our approach, the FANCI technique [14] will fail to identify the trigger signal presented in [26], because the control value of the output signal for its input signals remains above the defined threshold value as the trigger signal will always be contributing to the Trojan payload.

3.5.2 Scenario: 2

In this threat scenario, we assume that the IP core vendor, design house, and the communication channel can be trustworthy, but the EDA tools used by the producer are compromised. Figure 3.7 demonstrates the threat scenarios where the Trojan is inserted by the compromised EDA tools within the producer site. In this case, there could be two prospects, first, the producer or the design house has no knowledge about the subversion of the EDA tools thus malicious logic is inserted into the original by the compromised EDA tools without being noticeable to the producer. Second, the producer has access to executable binaries of compromised tools and replaces the benign ones with the compromised versions in the design house to launch the attack in an automated fashion targeting multiple designs, while the design house might not be aware of the tools used being compromised. One reasonable instance of this scenario would be a pre-compiled version of an open-source EDA tool that includes malicious code. This basically matches the attack from [26] shown in Figure 3.1.

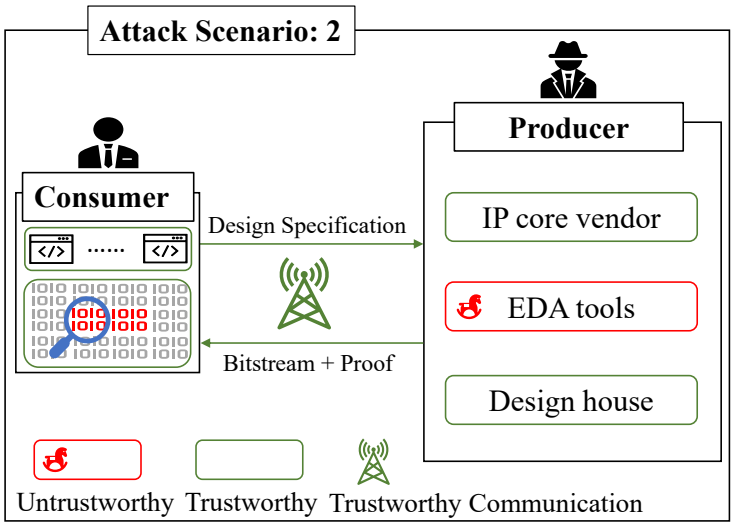

Figure 3.7: Attack scenario 2

Since in the first prospect, no collaborator within the producer would replace the design received by the consumer, the verification miter would always be formed using the original specification and the infected bitstream with activated Trojan, which would lead to again either a satisfiable miter, which would alert the producer to the fact that their tools are compromised, or it would help to ensure that the Trojan is harmless, if its activation does not violate the property. In any case, the producer again cannot, even by accident, create a malicious bitstream and matching proof that would fool the consumer into configuring an FPGA with it. In the second prospect, however, the producer using compromised EDA tools intentionally sends the infected bitstream and the proof to the consumer. Nevertheless, any malicious logic that violates the property, i.e., changes the functionality of the circuit, will be detected at the consumer site during verification. Whereas, the previous techniques [12–14] cannot detect the Trojan inserted by the malicious tools, because the malicious logic is added by the front-end synthesis tool which only stimulates the Trojan payload just before the bitstream generation step, while maintaining the infected circuit to behave functionally equivalent to the original circuit in all the prior design flow stages.

3.5.3 Scenario: 3

In this scenario, we assume that the design house and the EDA tools are trustworthy and the adversary has compromised the communication channel between producer and consumer, e.g., by performing a man-in-the-middle attack, which is also shown in Figure 3.8.

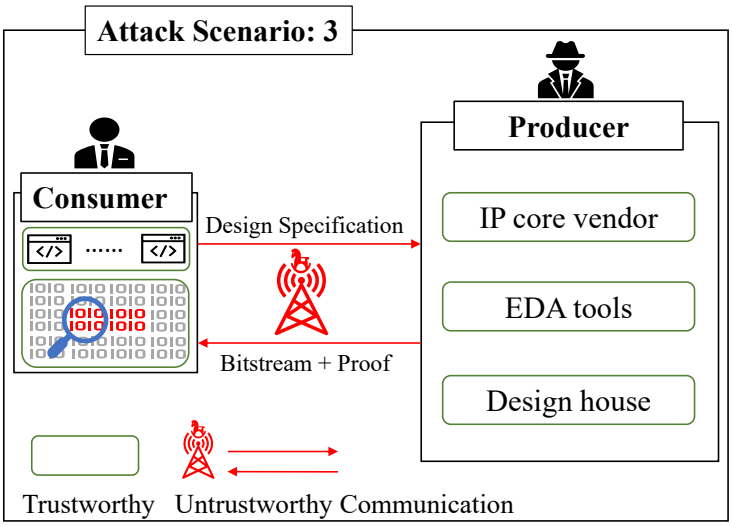

Figure 3.8: Attack scenario 3

There are several attack vectors, which would be detected in different steps of the flow:

1. The attacker replaces the design or security specification before it reaches the producer, inserting the Trojan in the specification or relaxing the properties that need to be proven. The producer would then go ahead and unknowingly a) produce a wrong design, or b) create an erroneous proof for the property afterwards. Since the consumer has an unmodified version of both specifications, however, and will use these to create their own version of the proof basis, both a) and b) would lead them to reject the module.

2. The attacker reverse engineers the proof-carrying bitstream and injects the Trojan into it. Since in this case the proof would not match the bitstream, the consumer would also be alerted when comparing the miter functions that the module is not trustworthy.

3. The attacker injects the Trojan into the PCB and modifies the proof. This leads to different outcomes, depending on the proof modification. If the new proof uses the correct miter, but consequently cannot actually prove its unsatisfiability, the consumer will reject the module. If the new proof is a correct proof of unsatisfiability using an alternative miter, the miter mismatch will be detected again. If the miter matches and the new proof indeed shows its unsatisfiability, then the attacker effectively has proven that their addition is

not violating the property and is hence not considered to be malicious by the consumer. The last case would hence not be detected, but that would be considered a non-issue.

The consumer would thus be alerted of any malicious, i.e., property-violating, alteration of the design in the PCB, no matter which communication direction the attacker compromises or what vector they choose, allowing the consumer to reject the module as untrustworthy. From the Table 3.1, we have shown that our method is able to detect the stealthy Trojan during design-time verification 3.5.1 either at the producer or consumer site which is indicated by ✓mark. Whereas, the previously presented approaches such as [12–14] are not able to detect the two-stage stealthy malicious LUT hardware Trojan which is indicated by ✗. Likewise, in the scenarios 3.5.2 and 3.5.3, the stealthy Trojan is detected by any of the party in our PCH flow, while, the state-of-the-art approaches are either not applicable or fail to detect it. However, the Trojans that do not violate the functional property, would not be detected but it may be considered harmless, concerning the agreed security specification as a full functional equivalence between the source design and the final design represented in a bitstream.

Table 3.1: Threat scenarios and possible detection of malicious LUT hardware Trojan in comparison to the previous works

Scenario	**Malicious party**	**Attack vector**	**Detected at** [Ours]	**Related Works** [12–14]
1	Design House	Design specification	Producer ✓	✗
		Both specifications	Consumer ✓	✗
	IP Vendor	Underlying IP Core	Producer ✓	✗
2	EDA Tools	Bitstream [26]	Producer ✓	✗
			Consumer ✓	✗
3	Communication	→ Design specification	Producer ✓	✗
		→ Both specifications	Consumer ✓	✗
		← Bitstream	Consumer ✓	✗
		← Bitstream & proof	Consumer ✓	✗
Any	Any	Trojan infection with valid new proof	Undetected: Harmless ✗	✗
			✓= can be detected	✗= can not be detected

3.6 Experimental Validation

3.6.1 Experimental Setup

We perform our experiments on a machine having an Intel Core i7-2600 CPU @ 3.40 GHz processor that includes eight cores with 8 GiB RAM and running a 64-bit Ubuntu 18.4 operating system (OS) on it. We have installed the original versions of the EDA tools and their dependencies together with the source files from the Icestorm project [27] on our machine. The original EDA tools are used to verify the efficacy of the attack performed by the infected tool flow. Moreover, for our PCH flow, we have integrated the consumer and producer flow in accordance with the outputs generated by different tools within the EDA toolchain. We use the following FPGA devices provided by Lattice semiconductor to evaluate our approach: iCE40HX-1K, which has 1280 logic elements (LUTs and FFs) with five user LEDs, and iCE40HX-8K having 7680 logic elements with eight user accessible LEDs.

3.6.2 Results

In this section, we present experimental results obtained using an example module represented in Verilog description. In order to demonstrate the ability of the presented iCE40 PCH flow to counter the attack presented in [26], we have used the example Verilog provided by the authors and the tools that they described where applicable. We have infected the EDA tools with the patches provided by them, and thus to the best of our knowledge have a faithful recreation of one of their experiments at our disposal, which we embedded into the flow described in Section 3.3.

For our experiments, we have first used the example Verilog code with unmodified EDA tools, thus validating the overall flow depicted in Figure. 3.3. Using uninfected versions of yosys and arachne-pnr on the producer side allowed us to synthesize the design for an iCE40-HX1K FPGA, and icepack produced the corresponding bitstream (.bin in Figure 3.5). Using the reverse tool iceunpack together with one of the icebox scripts allowed us, also on the consumer side, to obtain a Verilog representation of the implemented design. Using the original (behavioral) Verilog as specification counterpart, generically synthesized by a clean yosys, we formed a miter circuit and successfully proved its unsatisfiability using a SAT solver. On the consumer side, we then matched the miter functions and retraced the proof, leading us to an accepted proof-carrying bitstream. This confirmed that we had indeed successfully merged the

iCE40 tool flow with the PCH tool flow to enable PCH-certified bitstreams for iCE40-HX1K FPGAs. We then replaced the synthesis tools with the infected versions and reran the experiment. After computing the bitstream on the producer side, we unpacked and reversed it again in order to form the miter function with the specification (the specification blif was still generated by a clean yosys). This miter, however, proved to be satisfiable, since the implementation now actually had been altered. We hence achieved the expected result from Table 3.1, where PCH allowed us to detect the malicious modification of the design at the producer. A malicious producer could now try to hide the infection from the user, but as detailed in Section 3.5.1, this would then definitely be detected by the consumer in a later phase. Also, we compare our results against the stealthy attack presented in [26] with the state-of-the-art techniques and show that our bitstream-level verification approach is able to detect even the Trojans which remain inert in early design stages and are not detected by the design time testing techniques [12–14]. Furthermore, to evaluate our technique against all the possible configurations of a malicious LUT explained in [26], we have done experiments with the iCE40-HX8K device which confirms that our PCH tool flow is effective against all the configurations of malicious LUT and also supports iCE40-HX8K FPGA device.

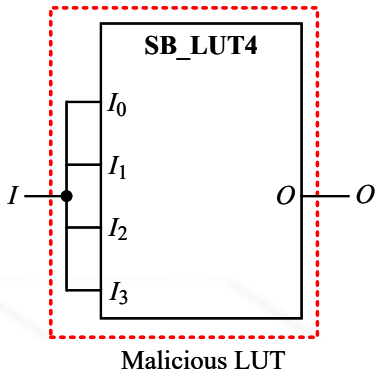

Figure 3.9: 4-input malicious LUT exhibiting malicious unary operation. Taken from [26].

The functional behavior of the malicious LUT remains the same as the original unary operations of a LUT during the simulation and varies in hardware, this is referred to as the malicious unary operations by the authors in [26] which are shown in Table 3.3. Since the input I of the malicious LUT is simultaneously applied to all of the four inputs of the LUT which results in either the least significant bit (LSB) or most significant bit

(MSB) bits to the output, hence realizing a malicious unary operation, as depicted in Figure 3.9. The Table 3.3 shows the 16 possible configurations of malicious LUT based on the KEEP and FLIP patterns, such that each pattern consists of 7-bits wide configuration words, i.e., higher-order configuration word (HOCW) and lower-order configuration word (LOCW). Fundamentally, these patterns form the basis for the malicious LUT to behave functionally equivalent to the original design during the design-time testing, therefore, helping the Trojan to evade any verification at this stage. To better understand the configuration of malicious unary operation of malicious LUT explained in Table 3.3, it is however important to know about the ordinary unary operation in LUT that is shown in Table 3.2 where the two inputs MSB and LSB drive the output function. For instance, if both the inputs are "0", the output re-mains "ZRO", similarly, the LUT operation will be "ONE", or a constant one, when both the inputs are "1", for the inputs "01" the LUT will perform "NOT" operation, and for the inputs "10" the output of the LUT will be "BUF" operation.

Table 3.2: LUT's unary operation

MSB	LSB	Unary operation
0	0	ZRO
0	1	NOT
1	0	BUF
1	1	ONE

We evaluate our example design for each of the 16 configuration patterns. To confirm that the malicious LUT's output does not change after the synthesis step, i.e., the first step in the attack, we connect the output of the malicious LUT to the light emitting diode (LED) of the iCE40-HX8K FPGA device. From the given KEEP and FLIP patterns, we can determine the expected behavior at the output, e.g., if there is no change at the output of the LUT, the LED will not blink and if there is a FLIP, the LED will indicate the flipping of the bits at the output. This behavior reflects the results given in columns 5 and 6 of the Table 3.3 where the $f_{simulated}$ column shows the output that is similar to the original unary output (cp. Table 3.2), while the $f_{hardware}$ column shows that the bits are flipped in a bitstream after the activation of a Trojan, i.e., in the second phase of the attack. Note that here the $f_{hardware}$ means that the Trojan is activated in the bitstream, which is the final design used to configure the reconfigurable hardware. We repeat the same phenomena after the place-and-route step where the blinking of LED will show the successiveness of the attack and otherwise, the activation of the trigger may either failed or the backend

tool could not match the properties of malicious LUT to activate the Trojan. After this step, we generate a bitstream for each of the 16 configuration patterns and verify the bitstream against any malicious behavior by applying our proposed bitstream-level PCH approach as explained in Section 3.3. Based on the observation, it is shown in Table 3.3, (cp. Column 7), that our PCH approach is able to detect the malicious LUT Trojan for all its possible configurations. Since the state-of-the-art such as [12–14] approaches mainly focus on the outcomes of the functional simulations of the design at RTL, which in the case of malicious LUT does not vary from the expected behavior (cp. $f_{simulated}$ in Table 3.3), thus the malicious LUT would easily evade them and would change the function of the design in the final stage, i.e., bitstream (cp. $f_{hardware}$ in Table 3.3). As our bitstream-level PCH approach compares the functional equivalence of the specified design, i.e., design at RTL, and the design that is obtained by reversing the final bitstream, i.e., the final design, so any functional changes made by the malicious LUT in the bitstream would be detected. For example, for each pattern used for configuring the malicious LUT in Table 3.3, where the functionality differs from the original implementation would result in a proof that can not prove its unsatisfiability using its miter at the consumer site in our approach, thus detecting any configuration of malicious LUT.

Table 3.3: Malicious LUT's possible configurations detected by our proposed PCH approach

MSB	HOCW	LOCW	LSB	Function		PCH observation
15	14–8	7–1	0	$f_{\text{simulated}}$	f_{hardware}	
0	KEEP	KEEP	0	ZRO	ZRO	Detected
0	KEEP	FLIP	0	ZRO	NOT	Detected
0	FLIP	KEEP	0	ZRO	BUF	Detected
0	FLIP	FLIP	0	ZRO	ONE	Detected
0	KEEP	FLIP	1	NOT	ZRO	Detected
0	KEEP	KEEP	1	NOT	NOT	Detected
0	FLIP	FLIP	1	NOT	BUF	Detected
0	FLIP	KEEP	1	NOT	ONE	Detected
1	FLIP	KEEP	0	BUF	ZRO	Detected
1	FLIP	FLIP	0	BUF	NOT	Detected
1	KEEP	KEEP	0	BUF	BUF	Detected
1	KEEP	FLIP	0	BUF	ONE	Detected
1	FLIP	FLIP	1	ONE	ZRO	Detected
1	FLIP	KEEP	1	ONE	NOT	Detected
1	KEEP	FLIP	1	ONE	BUF	Detected
1	FLIP	KEEP	1	ONE	ONE	Detected

The runtimes of our flow to detect the stealthy malicious LUT Trojan

(example design from [26]) at both the parties are discussed in Table 3.4. Column one of the table shows the infected designs with an inactive and active Trojan. In a first experiment, the infected design with an inactive Trojan, i.e., the malicious LUT is not triggered, is given to the producer flow which takes ≈ 2.317 seconds for the complete design synthesis and verification where the time taken to generate the miter and the proof validation is ≈ 0.088 and ≈ 0.001 seconds respectively. Since the Trojan is inactive thus the design is functionally the same which would lead to the unsatisfiable miter function and the proof is created and sent to the consumer alongside the bitstream. The consumer then regenerates the miter and a proof trace from the reversed bitstream for the comparison with the received proof trace which takes only ≈ 0.853 seconds along with the ≈ 0.084 seconds for miter generation and ≈ 0.011 seconds for proof checking. The success of both the proof checks shows that the infection of the design is harmless at this stage. The results from the first row on the table show a significant amount of workload shift from a consumer to a producer.

In a second experiment, the design in which the malicious LUT Trojan is activated using the compromised PnR tool is developed at the producer site. The runtime for synthesizing the design is ≈ 1.995 seconds, whereas the miter generation and proof check take ≈ 0.075 and ≈ 0.001 seconds respectively. In this case, the activation of Trojan would change the functional behavior of the circuit therefore the miter generation step would result in a satisfiable miter function which would prevent the producer to send the Trojan-infected design to the consumer. Columns two, four, and six of the table show that the consumer flow, in this case, would be irrelevant/not applicable. It can also be observed from Table 3.4 that the Trojan can be detected by any of the parties in a short amount of time which might be fast in comparison to the side-channel analysis technique based on the EM [97] and thermal scans [87] where the additional time is required to collect and process the EM and thermal signals.

Table 3.4: Verification runtimes to detect malicious LUT hardware Trojan. Averages of 5 runs.

	Runtimes [s]					
	Prod.[a]	Cons.[b]	Miter gen.[c]		Proof check	
Infected design [26]			Prod.	Cons.	Prod.	Cons.
Trojan inactive	2.317	0.853	0.078	0.084	0.001	0.011
Trojan active	1.995	–	0.075	–	0.001	–

[a] Producer
[b] Consumer
[c] Miter generation

3.7 Discussion

If we bring up the discussion about the hardware Trojans inserted during the FPGA design-flow, that would refer all the Trojans inserted at RTL to bitstream level. Although the threat model at different levels could be different, the purpose of an adversary might always be to configure an FPGA device with the intended malicious design. However, the design-time verification in FPGAs makes it difficult for an attacker to preserve the extra logic in the original design until configuration. Nonetheless, design-time verification can be eluded using the compromised FPGA design flow attack discussed in Section 3.1. It can be concluded that functional simulation at the RTL or gate level is not sufficient for the FPGA/bitstream security, as the Trojan could be injected in higher levels of the design flow which remain inactive to thwart functional verification and become active only at the last stage of the design flow. As there is no standard verification mechanism at lower levels in the design hierarchy, i.e., bitstream-level, there are chances that the adversary may bring out the attack by configuring the FPGA with a malicious bitstream.

Our bitstream-level PCH approach provides an opportunity for the verification engineers to successfully validate the FPGA designs against hardware Trojans that are either inserted directly in the bitstream or the higher levels of the design hierarchy and evaded the earlier design-time verification. Our technique for detecting the bitstream-level Trojans, such as stealthy malicious LUT Trojan, is effective yet scalable for the larger circuits containing Trojan-based on a two-stage infection mechanism. Our approach depends on the scalability of PCH technology presented in [85]. Since the presented PCH approach shifts the burden of verification from the consumer to the producer, therefore, allows the end-user to verify the bitstream quickly. On the other hand, if we discuss the limitations of our approach, we assume that in its current state, it is challenging for the bitstream-level proof-carrying hardware (PCH) to detect the Trojans that are not functional, i.e., a Trojan that provides the covert channel to leak some secret information from a circuit. Even, for the functional Trojans, if the malicious activation does not happen in the bitstream but in actual hardware, the proofs generated by the PCH would not be valid ones.

3.8 Chapter Conclusion

As already concluded by Krieg et al. [26], bitstream formats have to be publicly available to enable users to reveal and protect themselves

against malicious bitstream manipulations. Unfortunately, for commercial EDA tools this is usually not the case. In this chapter, we have demonstrated that bitstream-level verification using proof-carrying hardware is indeed able to reveal the stealthy two-stage hardware Trojan attack presented in [26], which is undetectable using regular state-of-the-art preconfiguration detection approaches. The power of PCH ensures that even in a two-party contract work scenario, where the producer's tools are compromised, the consumer is protected against the modifications. We thus underline the claim by Krieg et al. that their attack is only possible because of the closed nature of commercial bitstream formats, and conclude that in a world with open bitstream formats the problem would not only be solvable but is indeed already solved. Not only for the one party version described in [26] (consumer + attacker), but also in the two party version defined by PCH (consumer, producer + attacker), which is a common case in today's market, where designers build their designs from a multitude of third-party IP cores.

However, irrespective of being a powerful verification tool, our proposed PCH flow would only detect the Trojans which change the functionality of the circuit upon activation, e.g., the Trojan in malicious LUT attack. Skillfully designed non-functional Trojans, e.g., the Trojan that provides a covert channel upon activation to leak secret information or the Trojans that are externally triggered to change the behavior of device parameters, may even avoid our proposed bitstream-level verification. To show the ineffectiveness of a bitstream-level verification for a non-functional key leakage Trojan, we propose a novel malicious routing-based Trojan in Chapter 4, which successfully leaks the secret key through a covert channel (e.g., I/O pin).

Chapter 4

Post-Configuration Activation of Hardware Trojans in FPGAs

We will begin this chapter with a brief overview of the attack and the threat model that is considered to realize the attack in Section 4.1, the detailed methodology concerning the malicious insertion, routing, and activation of a Trojan along with the communication between compromised tools in a design flow is described in Section 4.2, the experimental setup and the successful key-leakage from an AES-128 core are explained in Section 4.3 followed by the discussion on the attack with respect to its applicability and detectability in Section 4.4, finally, we conclude the chapter in Section 4.5.

The worked described in this chapter was presented in *Design, Automation and Test in Europe* (DATE) conference in 2021 and the findings are published in [49].

4.1 Overview and Threat Model

In this section we provide and overview of a propose a novel attack that exploits malicious routing of the inserted Trojan circuit to attain a dormant state even in the generated and transmitted bitstream, thus circumvents bitstream-level verification techniques described in Chapter 3. The Trojan is inserted in the second stage of the field-programmable gate array (FPGA) design flow, i.e., when the design's netlist is read by a malicious placement-and-routing (PnR) and is activated only in the FPGA device itself. The novelty of our attack lies in the fact that the routing tool disconnects the Trojan circuit from the original circuit before writing the bitstream and the FPGA programming tool again connects the Trojan circuit with the original circuit, thus activating the Trojan during/post configuration. Consequently circumventing bitstream-level verification which would lead to false negatives to the certificates generated by PCH, as the functionality of the circuit in the a bitstream is not changed but only the behavior in the step after that.

Trojan insertion by the design house and a third-party intellectual property (IP) provider are considered as major threats for FPGA system-on-chip (SoC), however, in our attack model we consider that both the design house and the IP vendor are trustworthy, but the EDA tools used by the design house are compromised by an attack; either by reverse-engineering the binaries of commercial electronic design automation (EDA) tools to insert malicious code, or via an insider in the EDA tools provider who maliciously swaps the legitimate binaries with malignant ones used for compilation by the design houses. We consider the foundry as a trusted entity in our threat model, since even though some FPGA vendors are fabless and outsource device fabrication to a third party [47], attacking the FPGA fabric itself is less effective than for application-specific integrated circuits (ASICs) as the design to be implemented is loaded after the device is fully tested and shipped. We follow the threat scenario presented in [26], where the malicious code is inserted into an open-source tool that is then compiled to a binary version which in turn is used to intrude on the design house, ideally over the Internet, in order to replace the legitimate binary of the tool in the design house with the malicious one to infect multiple machines of the design house in one go. However, the compromised EDA tools are not only limited to synthesis and place-and-route tools, but the tool that programs the FPGA could also be subverted to activate a Trojan inserted in the earlier stages.

Figure 4.1 highlights the entities involved for the development of the

hardware module specified by the consumer. It can be seen that the design house itself is trusted but the EDA tools used by the design house are subverted by attacker, as explained above, to gain control over the device when it is configured without being noticed by the producer or the consumer. In our threat model, the place-and-route tool and the FPGA

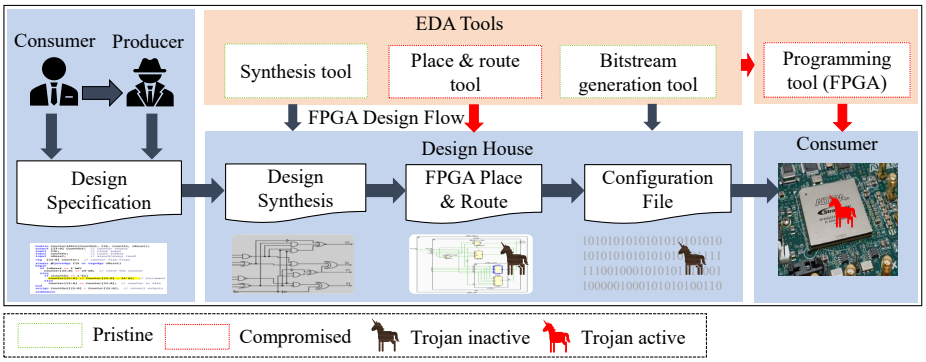

Figure 4.1: Threat model: Compromised EDA tools.

programming tool are compromised to inject and activate the Trojan, and thus their binaries are marked as a red dotted box in the compromised EDA tool-chain in Figure 4.1.

We would also like to stress that our attack only activates the Trojan if the FPGA programming tool used by the consumer to program the FPGA device is compromised along with the place-and-route tool used by the design house and the configuration bitstream is either un-encrypted or the programming tool is capable to decrypt it, otherwise, the output of the infected tools will behave similarly as the original ones when the FPGA is programmed with the genuine programming tool. In this way, the attacker can conduct multiple targeted attacks by infecting only the programming tools of intended targets, which implies that the inserted Trojan will remain inert and therefore virtually undetectable in most of the customers' designs, thus there are fewer chances of the attack being revealed by chance.

4.2 Methodology

4.2.1 Overview of Malicious Design Flow Attack

The general design flow of our attack for malicious insertion, routing, and activation is shown in Figure 4.2. The attack works in two phases, i.e., in the first phase the Trojan circuit is injected, attached and then disconnected at one of the FPGA's programmable interconnect points (PIPs)

by the PnR tool. We call this temporary breaking point Trojan PIP (TPIP). This step is marked with a dotted red box in Figure 4.2, to indicate that it is modified. In the second phase, i.e., the last stage in the design flow, the TPIP is activated again by the modified FPGA programming tool to connect the Trojan circuit to the original circuit.

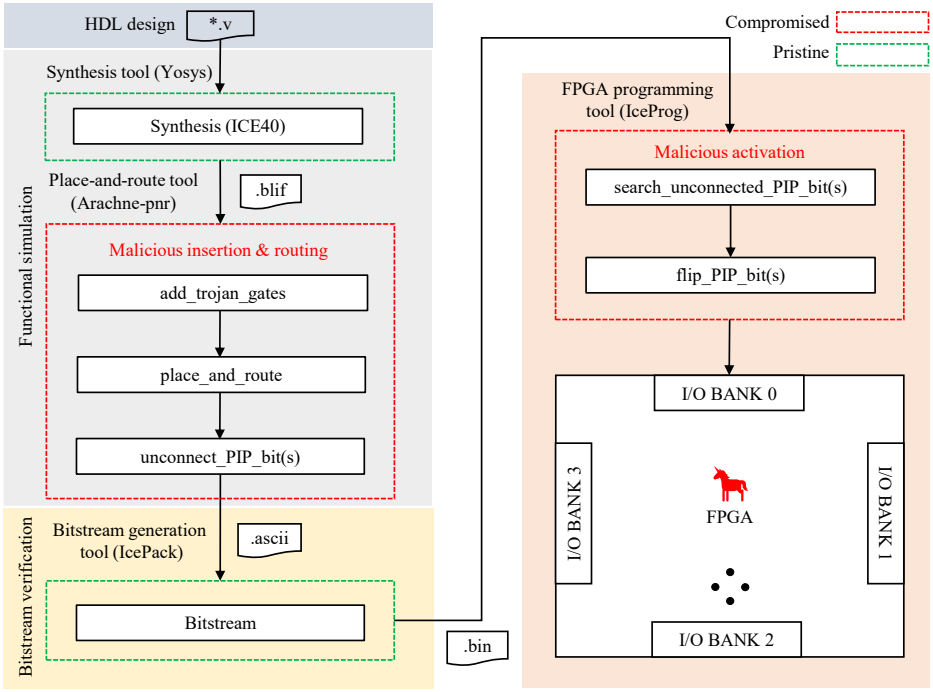

Figure 4.2: Compromised FPGA design flow for malicious insertion and activation of a Trojan.

Each of the stages of the design flow is explained in the following: The first step in our design flow, the synthesis of an hardware description language (HDL) design by a synthesis tool, is not infected, and therefore the generated netlist remains unchanged compared to the design flow using pristine EDA tools. The next step of the design flow is compromised such that the Trojan circuitry is added to the synthesized netlist when it is read by the PnR tool, e.g., barrier gates who route secret information to primary outputs, but prevent it from leaking until the Trojan is activated. After placement and routing, the connection of the Trojan to the original circuit is removed by flipping the TPIP configuration bit to "0" so that the output at this stage behaves functionally equivalent to the original design, thus avoiding detection by any functional simulation and verification methods.

Note that in contrast to compromised tools targeting the register-transfer level (RTL), the benefit of the post-synthesis Trojan insertion is that it still works if an IP core provided by a third-party to a design house

is a gate-level netlist. Even if the design is already a verified synthesized netlist, it can be infected in the next step. Furthermore, machine learning (ML) approaches based on reverse-engineering the bitstream to obtain a gate-level netlist for feature extraction mainly consider trigger nets [88–90], and thus cannot detect our Trojan circuit for two reasons: a) The information-leaking version is trigger-less, and b) in the general case, the payload of the Trojan is unconnected from its trigger in the bitstream, hence a reverse-engineered netlist would result in false negatives.

In the next step, a bitstream configuration file is generated by the unmodified bitstream generation tool which carries the Trojan payload that stays unconnected at this stage, hence evading any bitstream verification mechanism such as PCH. In the last stage of the FPGA design flow, when the bitstream is loaded onto the FPGA by the programming tool, the connection of the Trojan to the original circuit is re-established by flipping the TPIP configuration bit again to "1".

The complete flow of the proposed Trojan insertion methodology is described in Algorithm 1. Only the steps that are compromised, i.e., steps 2 and 4 that are marked as red boxes in Figure 4.2, are depicted in the algorithm. Since the Trojan is inserted after the synthesis step, therefore the malicious routing algorithm is initiated during the placement and routing step. The algorithm takes the netlist (.blif) file as an input together with the constraints file (.pcf). The output of the algorithm is a Trojan inserted bitstream that is activated only when the FPGA is being programmed. To exemplify, we target a netlist of an advanced encryption standard (AES) module that has to be modified with compromised tools to leak the secret key. The algorithm first initiates a validation step to verify if the given netlist is comprised of the AES module. If the netlist is valid, the algorithm looks for the output registers, i.e, the registers located just before the primary output, and connects a multiplexer (MUX) to each of the output registers in each iteration. This step is marked as the Trojan insertion step in Algorithm 1 line 2. The enable line of MUX is also appended in the inputs list of the circuit (.model in a blif format). In our attack, only the 8-bits of key are to be leaked, thus, as long as the condition $i<8$ is true, where i is the iteration count for inserted MUXes, the barrier gates comprising of 2×1 MUX are connected to the output registers along with the desired key bits to be leaked.

Next, when reading the constraints file, the algorithm searches for an unused input/output (I/O) pin of the device and assigns it to the enable line of MUX. The design with added barrier gates is then placed and routed by the PnR tool. Only when the PnR tool writes the design as a

Algorithm 1: Malicious routing algorithm

Input: { D_O = Original design (.blif), C = Constraints file (.pcf) }
Output: { D_T = Trojan inserted design }

```
1  if (aes_128 in D_O) then
       // *Trojan insertion Step* //
2      Identify output registers
3      for i = 0; i < 8; i = i + 1 do
4          barr_gate ← 2 × 1 MUX as LUT
5          Connect barr_gate to ith output register
6          Update netlist
7      end
8      Append enable line of barrier_gate(s) to inputs
9      Read constraints file (.pcf)
10     if (unsued I/O pin in constraints file) then
11         Assign enable line to the I/O pin
12     end
13     Update constraints file with enable line
       // *Trojan Disconnection Step* //
14     Search (T)PIP bit(s) connecting enable line to the I/O pin
15     if TPIP_bit(s) then
16         Flip TPIP_bit(s)
17     end
18     Generate text_file from placed and routed design
19     bitstream_file ← Generate bitstream file(text_file)
       // *Trojan Activation Step* //
20     if program_mode then
21         Search flipped TPIPbit(s)
22         if TPIP_bit(s) then
23             flip TPIP_bit(s)
24         end
25     end
26     Update bitstream_file with connected TPIPbit(s)
27 end
28 return D_T ← bitstream_file
```

text file, the algorithm delves for the PIP bit(s) (the TPIP) that connects the enable line to the I/O pin and flips it, i.e., Trojan disconnection step in Algorithm 1 line 14. The output of the PnR tool, e.g., text file, will have a Trojan but disconnected at this stage which is given to the bitstream generation tool to generate a bitstream configuration file. In the last step, i.e., the Trojan activation step in Algorithm 1 line 19, when the bitstream is available to program the FPGA, the algorithm searches for the flipped TPIP bit(s), and overturns it again to re-establish the connection between the enable line of barrier gates and I/O pin. Once, the connection is established, the 8-bit key can be leaked by the attacker using the specified I/O pin.

For functional Trojans, the circuit diagram of a simple design with inserted Trojan trigger and payload shown in Figure 4.3 describes the internal schematic view of the design in a bitstream and the FPGA to better understand the attack. The switch between the trigger and the payload refers to the TPIP used to disconnect and connect the trigger, thus rendering the payload inactive and the Trojan dormant. Though our malicious flow is generic and any kind of Trojans can be implemented, an intelligent attacker would be interested to get higher level attacks and much control to the design such as secret key leakage from a cryptographic circuit without external resources.

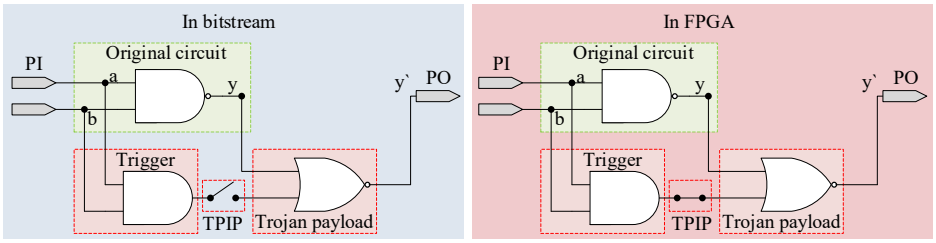

Figure 4.3: An exemplary circuit schematic of an infected design in a bitstream and FPGA.

This can be done if the enable pin of the barrier gates is maliciously routed to one of the unused I/O pins and the connection through the TPIP is removed when the tool writes the output into a bitstream. The information of the unused I/O pin can be obtained by scanning the constraints file, therefore making this pin available for activating the Trojan in the final stage. After the connection of the I/O pin is re-established by the modified programming tool and the FPGA is configured, the attacker can apply voltage to the pin to activate the Trojan circuit which leaks the key through the output pins used by the original design instead of cipher

text. The key leakage of an AES core by a Trojan payload is explained in Section 4.3.3.

4.2.2 Flow of Information Between the Compromised Tools

A prerequisite for a successful attack is that the FPGA programming tool knows the location of the TPIP in the infected design. The attacker therefore has to communicate this location from the tool that chooses it to the tool where it is used, requiring them to create a hidden communication channel between the design house and the consumer site. The attacker can realize this communication explicitly, i.e., over a new channel, or implicitly, i.e., by hiding the information among the regularly transmitted data. With explicit communication, the TPIP location is decoupled from the bitstream, and hence needs to be related within the programming tool in order to apply the correct TPIP flip for the loaded design. Depending on whether the attacker wants to target specific individual designs created at the design house, or rather infect all designs originating from there, they have to either transmit one location or create and query a database that maps design identifiers (e.g., design hashes) to TPIP locations. The former would require the attacker to closely monitor the designs that should be written in the near future, which would likely necessitate the continued presence of an agent on-premises, who could then also be leveraged to exfiltrate the information via human communication. The widespread infection of many designs, on the other hand, would generate more traffic over the hidden channels, increasing the chance of the channel being detected by the victims. However, to scale such an attack to multiple design houses and consumers, an attacker could then leverage existing botnet solutions to create a command-and-control infrastructure akin to modern software Trojans, resulting in exactly the same advantages and disadvantages as in the software world, i.e., fast and easy scalability, high degree of automation and flexibility, versus vulnerability to automated, pattern-based communication-detection methods and a complete failure of the attack in a high-security environment, where the programming tool is not connected to the internet and thus cannot query TPIP locations.

Using implicit communication instead, the attacker can only rely on the one communication that the victims cannot avoid, which is the transmission of the bitstream. By reverse engineering the bitstream format used by both parties, the attacker could leverage unused portions of the stream to hide the TPIP location in plain sight, e.g., pockets of legacy information that are no longer in use for modern devices, or information

that follow the end-of-stream and which would thus be ignored by the regular tools. This form of communication implicitly matches the TPIP location to a design and therefore does not require any method of design identification afterwards, but in order for the attacker to sustain such an attack, they would need to continue to reverse engineer bitstream formats and update their transmission code every time that a format is updated or the involved parties switch to a new format. However, since the attack itself also relies on modified binaries within the EDA suite, the attacker would have to update the compromised tools anyway in these cases.

4.3 Experimental Validation

4.3.1 Experimental Setup

We perform our experiments on a machine having an Intel Core i7-2600 CPU @ 3.40 GHz processor that includes eight cores with 8 GiB RAM and running a 64-bit Ubuntu 20.04.2 LTS operating system (OS) on it. We have installed the original versions of the EDA tools and their dependencies together with their source files from the Icestorm project [27] on our machine. The original EDA tools are used to verify the efficacy of the attack performed by the infected tool flow. Furthermore, we have installed an *iCEcube2* tool for power estimation. We use the iCE40HX-1K FPGA device, which has 1280 logic elements (LUT and FF) with five user LEDs, provided by Lattice semiconductor for evaluation.

4.3.2 Overview of an AES Core

The advanced encryption standard (AES) is a symmetric-key encryption algorithm used to encrypt the input data to make it secure during transmission and works efficiently in both hardware and software. The AES has different encryption packages (such as keys with 128, 192, and 256 bits), and a fixed block size of 128 bits, where the size of the key used to convert the plaintext into the ciphertext determine the number of transformation rounds, which are 10 for 128-bit keys, 12 for 192-bits keys, and in case of 256-bit keys, the total number of rounds are 14. The number of rounds enhances the level of difficulty to break the encryption using, for example, brute-force attacks [91]. The AES core has been getting more advanced with every next iteration, respecting its architecture with datapath ranging from 8-bits to 32-bits, besides supporting 128-bit, 192-bit, and 256-bit keys, to make compact and low power encryption solutions in modern-day devices. Recently, a compact variant of an AES core with

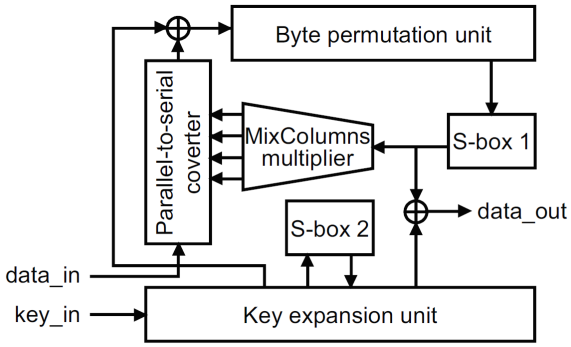

Figure 4.4: The top-level architecture of an 8-bit datapath AES-128 core. Taken from [92].

8-bit datapath and capable of encrypting the block of 128 bits of keys has been presented in [92], which supports the execution of the operation of AES rounds in parallel, in contrast to sequential operations, ultimately decreasing the cycle count and increasing the efficiency. The top-level architectural view of an AES with an 8-bit datapath, (i.e., 8 bits data width of all the registers and connections) is shown in Figure 4.4, which consists of the following components to perform the specific functions: *Byte permutation unit* for ShiftRows operation after every cipher round, *MixColumns multiplier* for matrix multiplication and AddRoundKey operation, *S-box* for key-substitution, *Key expansion unit* for expansion of key by pre-calculating the RoundKeys and to store them, and a *Parallel-to-serial converter*. Further detail about the working of the complete AES encryption process can be found in [92]. We will employ the Verilog code of an AES-128 core with an 8-bit datapath provided in [92, 93] for the assessment of our technique in the context of this thesis, which is the topic of the next section.

4.3.3 Secret Key Leakage of an AES Core

As a proof of concept, we demonstrate our attack by implementing an AES core[1] with an 8-bit data interface from [93] to an iCE40HX-1K device, iCEstick Evaluation Kit provided by Lattice semiconductor [84]. We use the open-source design flow from the project *IceStorm* [27] for iCE40 FPGAs, which consists of the *Yosys open synthesis suite* [94] for hardware synthesis, *Arachne-pnr* [86] for placement and routing, *IcePack* and *IceProg*

[1]Note that we will use the terms AES Core, AES module and AES design interchangeably in this thesis.

for bitstream generation and programming the FPGA respectively. To circumvent verification at the gate level, we did not modify *Yosys*; hence the output at this stage is a legitimate synthesized netlist *(.blif)* of the AES design synthesized for iCE40 FPGAs, i.e., using `synth_ice40` command. After that when the compromised *Arachne-pnr* reads the netlist by invoking `read_blif`, it inserts the Trojan circuit into the original circuit. The commands used in our flow are given in Listing 4.1, where the commands in rows three and five call the malicious script from the tool to infect the design.

```
1 yosys -p read_verilog example.v
2 yosys -p synth_ice40 -blif example.blif
3 arachne-pnr -d 1k -o example.asc icestick.pcf example.blif
4 icepack example.asc example.bin
5 iceprog example.bin
```

Listing 4.1: Commands used by the malicious design flow in our examples. Line 2 and 5 show that these commands are used by the compromised tools.

An example for the two-phased process of inserting and activating an I/O-triggered key-leaking Trojan based on barrier gates into an AES core is shown in Figure 4.5 and Figure 4.6, respectively. Figure 4.5 refers to Phase 1 that shows the barrier gate insertion and routing of the enable signal to an unused I/O pin, which is disconnected at a PIP, the TPIP, while writing the design output. Our barrier gates consist of eight 2×1 `MUXes` connected to the very last register before the primary output. The inputs to each MUX is the output of the register containing cipher text, and the secret key to be leaked. The key from any of the rounds can be

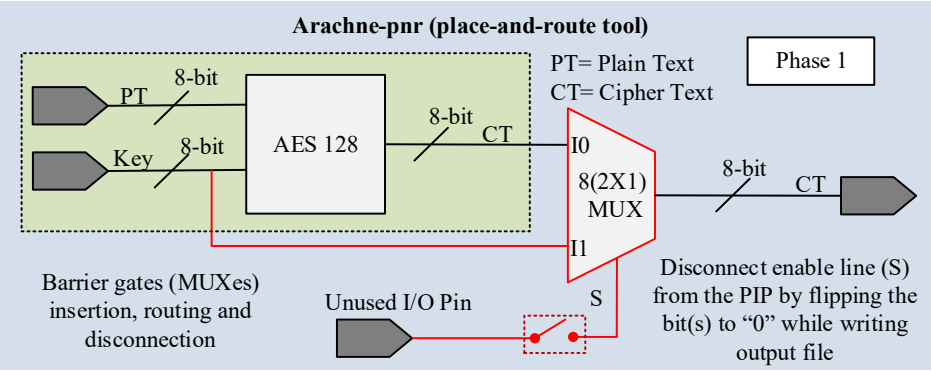

Figure 4.5: Phase 1: Trojan insertion and disconnection in AES core by the compromised PnR tool.

leaked, however for the sake of simplicity, we take the key that is used as an input to the AES module. Depending on the attacker's intentions the enable input "S" of the barrier gates in Figure 4.5 could be attached in two different ways: a) A constant "1", or b) an unused I/O pin which is added by *Arachne-pnr* when it reads the constraints file, (*.pcf)*. In the first case, after placement and routing, the TPIP connecting the enable line to the barrier gates is flipped when the output file is being written.

In the second case, the infected design is placed and routed making sure that the enable line "S" of the barrier gates is routed to the newly assigned input pin and the address of the I/O tile containing the input pin is stored. When the routed design is being written to a text file *(.ascii)*, the modified `write_text` function is invoked in the backend, which accesses the corresponding I/O tile and its connection to the barrier gates via the TPIP is removed. In the architecture of the iCE40HX-1K device, each I/O tile can have two I/O blocks, one for inputs and an other for outputs with two local tracks. Each I/O block in a tile is connected to the other I/O or logic blocks with the help of PIPs called vertical and horizontal spans in the iCE40 family. To remove the connection of the enable line from the input pin, the PIP that connects the enable signal to the I/O tile is flipped to "0". Likewise, when the FPGA programming tool, *IceProg* configures the FPGA, the corresponding TPIP bit in the bitstream, communicated by the design flow, is flipped again to make the connection of the enable line to the barrier gates which is illustrated as Phase 2 in Figure 4.6.

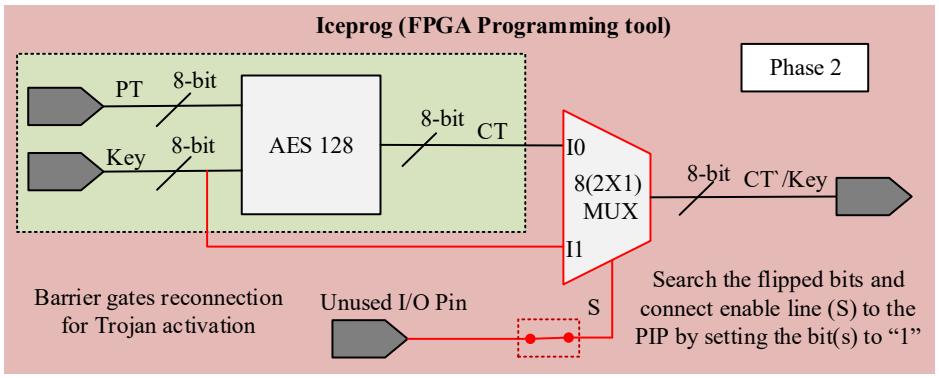

Figure 4.6: Phase 2: Trojan re-connection and activation in AES core by the compromised programming tool

In our example, the TPIP is located in I/O tile (10, 17), corresponding to the bit at address 0×4743 in the bitstream. Once the connection is made, the cyclic redundancy checksum cyclic redundancy checksum (CRC) is updated and the FPGA is configured with the infected bitstream. Since one of the inputs to the barrier gates is a cipher text bit and the other

is a secret key bit, the barrier gates act as buffers that pass the cipher text to the outputs when the enable line is disconnected or low. In the first case, the Trojan circuit will be activated and start leaking the key straight away, and can hence be accessed remotely or locally by the attacker. However, in the second case, the attacker needs physical access to the device to activate the Trojan circuit by giving voltage to the specified I/O pin used by the malicious tools, thus triggering the key leakage.

Although physical access is required, the attacker, in this case, has more control over the device, i.e., switching between the original circuit and the Trojan circuit using an input pin to leak the secret key. Therefore, the attack is more difficult to be revealed in the field as compared to the first case where the output of the implemented design will always be a malicious one. Nevertheless, in both cases, we have successfully performed the attack to leak the 8-bit secret key in our example. In order to illustrate, we have used the five available LED's on the device and connected three external LED's to read out the leaked key byte.

4.3.4 Trojan Impact

One of the important aspects of a Trojan circuit that underpins a trigger to remain stealthy is a low switching activity. The attacker intends to hide a Trojan into the areas where the transition probability of a circuit is considerably low, thereby preventing the Trojan to be detected during the design-time testing. Whereas, the defender's goal is to monitor the rarely triggered conditions by applying advanced test patterns; marking them as malicious and removing them, or increasing the switching activity of the circuit to activate even the smaller size Trojans [95]. As our Trojan is inserted after the synthesis step and also remains unconnected during the rest of the design flow, thus will have no impact on the switching activity of the design, hence cannot be caught by monitoring the rarely triggered nets or by increasing the switching activity of the circuit.

Side-channel analysis methods are usually effective for larger designs, thus to remain undetectable and to escape side-channel analysis techniques, a Trojan circuit has to be small enough, causing a minimum impact on the area, power, and delay parameters. In order to evaluate the impact of our Trojan on these parameters, we acquire the resource utilization report for original AES design and the malicious design using the command `icebox_stat` from *IceBox* tool within the project *IceStorm*.

Table 4.1 highlights the resource utilization of an original and malicious AES design implemented on an iCE40-HX1K device. The device

Table 4.1: Resource utilization of AES design in iCE40-HX1K

		Utilized by AES design	
Resource name	Available	Original	Malicious
# of LUTs (4-input)	1280	543	551
# of FFs	1280	319	319

consists of $1,280 \times$ 4-input LUTs for user design implementation together with the 64KB of an embedded random access memory (RAM). Additionally, 16 general-purpose input/output (GPI/O) pins and 5 user-LEDs are available for design analysis of different inputs and outputs. It can be observed from the Table 4.1 that the difference between the utilization of LUTs, and the flip-flops (FFs) in both designs is significantly low which proves that our Trojan circuit is small in contrast to 49 gates in [57] and 14 LUTs in [96]. The number of LUTs used by the original design is 543 out of 1280 available, whereas the number of LUTs used by the malicious design is 551 with a difference (δ_A) of only 8 LUTs, that is $\approx$ 1.4% of the total LUTs used by the original AES design, also depicted in Figure 4.7.

As the size of the Trojan decreases, the detection probability also decreases, e.g., the false-negative rate of the Trojan circuit, of size 1.7% of the original circuit, climbs from 5% to 17% for a Trojan size of 1% in [97] which is very much closer to the size of our Trojan circuit. Moreover, due to compatibility grounds, we chose to implement a smaller compact version of AES, however, for standard AES design, leaking 8 bits of a secret key using only 8 LUTs would have further low detection probability. (δ_A) is calculated by using the equation (4.1), written as follows:

$$\delta_A = \frac{Area_{MD} - Area_{OD}}{Area_{OD}} * 100 \tag{4.1}$$

Where, $Area_{OD}$ corresponds to the area utilized by the original design and $Area_{MD}$ is the area utilized by the malicious design which is calculated by using equation (4.2), written as follows:

$$Area(\%) = \frac{No.\ of\ LUTs\ used}{Total\ available\ LUTs} * 100 \tag{4.2}$$

Note that we calculate the area in terms of the number of LUTs. The Trojan does not contain any register element, hence, the flip-flops (FFs) utilization is unchanged in a malicious design. Moreover, our Trojan circuit is disconnected in a bitstream, therefore, it will have no impact on the power consumption if the placed-and-routed design is simulated to obtain the power traces for verification purposes. Furthermore, our Trojan is only triggered when the input is applied to the specific pin, which

is hidden from the user and only the attacker knows about it.

So, the runtime Trojan detection techniques relying on Electromagnetic (EM) and physical side channels [97], exploiting the fact that the Trojan is already triggered, may not be able to distinguish between the power traces of the original design and the compromised design. Also, considering the circumstances for the larger designs, it would not be viable for the user to verify all the I/O pins for Trojan activation. Similarly, there is low chance that a user may accidentally apply voltage to the pin which has not even used in the actual design, to activate the Trojan. There may be a slight increase in the power consumption due to the addition of 8 LUTs, yet lower in comparison to [57, 96], where the Trojan circuit inserted in AES-128 design contains 49 gates and 14 LUTs, respectively, which should be negligible considering the process variations and the environmental noise. Nevertheless, for a fair comparison, we first measure the power consumption for the original AES design using the power estimator tool within the *iCEcube2* design software for Lattice FPGAs [98]. Next, we implement a compromised design and use the power estimator tool to measure the power consumption. The bars at the bottom of

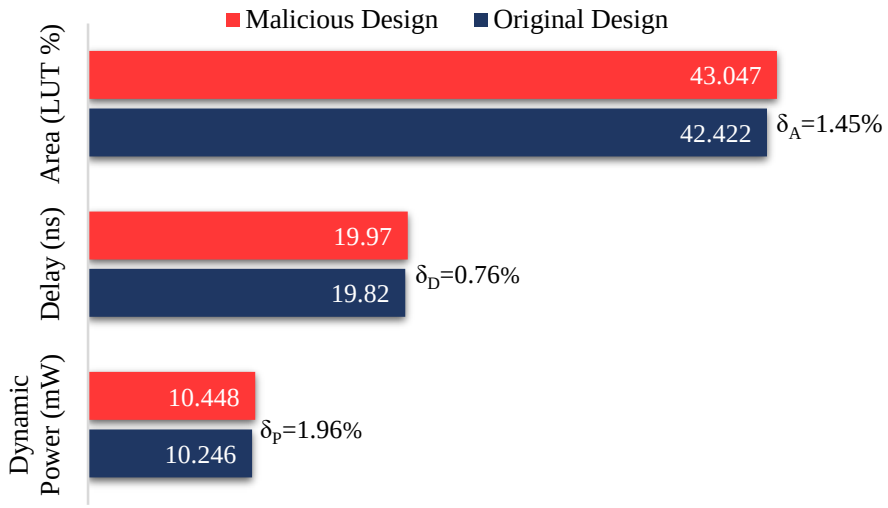

Figure 4.7: Power consumption, delay and area utilization in original and malicious AES design

Figure 4.7 show the dynamic power consumption in nano-Watt (nW) by the original and malicious AES designs. It can be seen that there is only a difference (δ_P) of $\approx$ 1.96% in power consumption by the original and compromised designs.

Where, (δ_P) is obtained by applying equation (4.3), which is written as follows:

$$\delta_P = \frac{Power_{MD} - Power_{OD}}{Power_{OD}} * 100 \tag{4.3}$$

While $Power_{MD}$ and $Power_{OD}$ refer to the dynamic power consumption in malicious and original design accordingly. Also, the dynamic power consumption of a circuit depends on the switching activity of the circuit with the clock frequency which in our case is set to 50 MHz. It is also important to note that if the barrier gates are not inserted on the critical path, it will not have any impact on the delay of the circuit. To avoid the insertion of the barrier gates on the critical path, the information of critical paths can be acquired before inserting the Trojan gates. However, in our case even if the barrier gate, i.e., one LUT is inserted on the critical path, that will have minimum effect on the timing of the circuit that could easily evade the delay-based detection techniques [99]. Even so, for the purpose of clarification, we insert a barrier gate without avoiding the critical path and measure the difference (δ_D) in delays of both the circuits, estimated in nano-second *(ns)* by the *IceTime*; a timing analysis tool within IceStorm, using the equation (4.4) which is given as follows:

$$\delta_D = \frac{Delay_{MD} - Delay_{OD}}{Delay_{OD}} * 100 \tag{4.4}$$

The total path delay in the original and malicious design is 19.82ns and 19.97ns, respectively, with the increase of only 0.15ns delay. The bars in the middle in Figure 4.7 enlighten the percentage difference (δ_D) that is $\approx$ 0.76%.

4.3.5 Demonstration Example

To assess that our barrier gates inserted and connected properly, we have synthesized the example design given in Listing 4.2 with Yosys and then gave it to the compromised PnR tool. We have read out the example design in a text file from *Arachne-pnr* before enabling the malicious back-end script that is used to disconnect the TPIP. The left part of Figure 4.8 shows this design as visualized by the ICE40 layout viewer [100].

Next, we enabled the back-end script and compiled the tool to a final compromised version. Using a synthesized netlist of an AND-gate as an example design for the compromised PnR tool, we have obtained the design depicted on the right-hand side of Figure 4.8, confirming the successful removal of the enable line after the TPIP has been flipped. To confirm the disconnection of the enable line from PIP, the text file is again

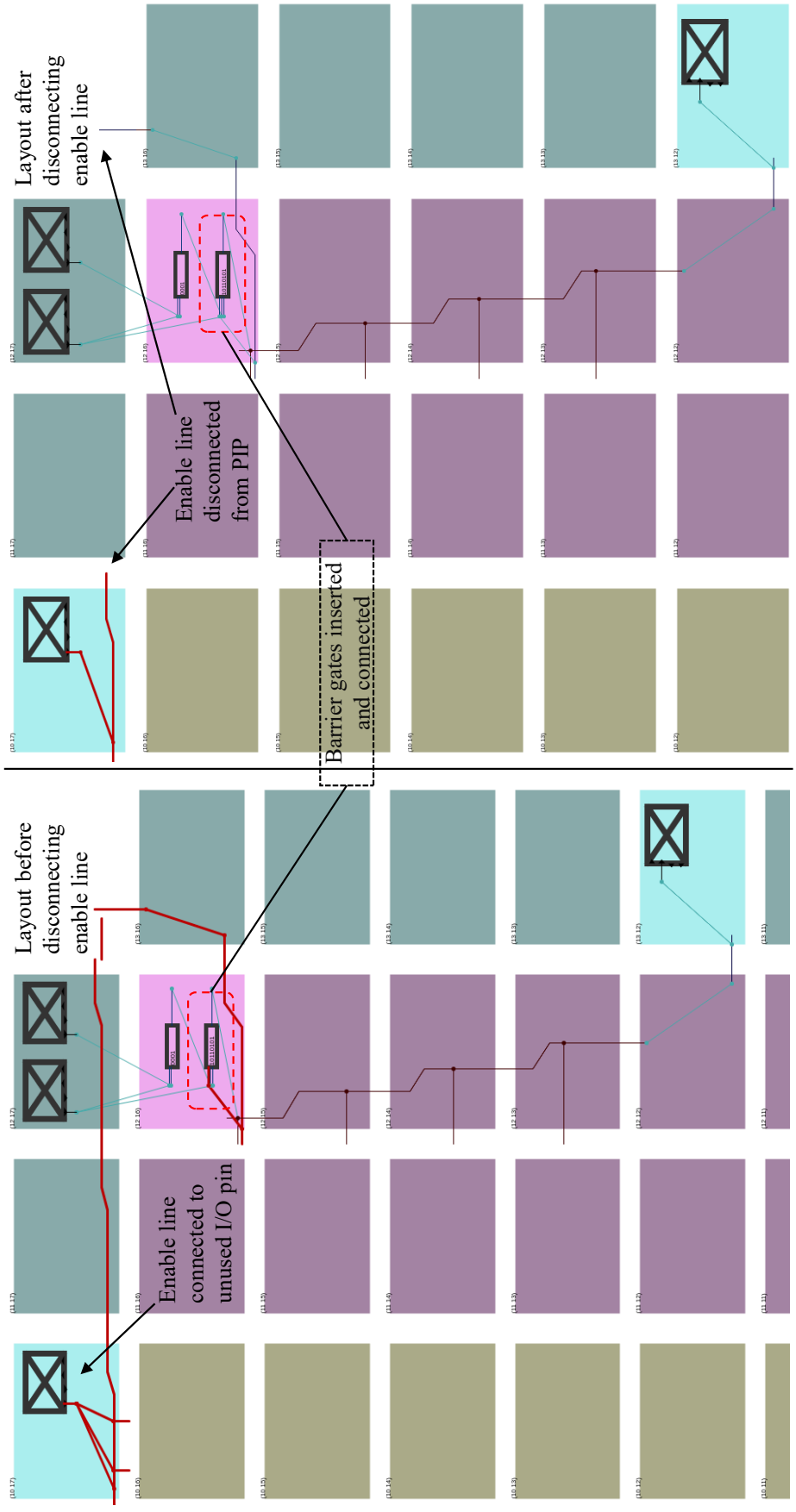

Figure 4.8: ICE40 Layout Viewer to evaluate the barrier gates insertion and disconnection by a compromised PnR tool in AND-gate design

given to the ICE40 layout viewer, see the right part of Figure 4.8. After this successful subversion of the PnR tool, we have modified the programming tool and then have programmed the FPGA with it to see the output of the design. We have verified the output with both, the original version of the programming tool and the compromised version, where the output of the former is not affected when the enable line is high while the latter then leaks the input "*b*" at the outputs.

```
1 module top (input a, b, output y);
2 assign y = a & b;
3 endmodule
```

Listing 4.2: Original Verilog code for AND-gate example

4.4 Discussion

When we talk about Trojans in FPGAs, the common question discussed is the way Trojans are inserted into the design such that it cannot be detected by conventional testing methods. In this regard, we present a hardware Trojan attack employing compromised EDA tool flow in which the Trojan is inserted into a design during the place-and-route step, remains dormant in the bitstream and is activated while loading the bitstream configuration file into the FPGA. Note that we use an open-source tool flow to present our attack, however, in principle, the attack would work for the commercial tools provided by other FPGA vendors, e.g., Xilinx, if the bitstream format is known and there is access to the code of tools. Our attack is versatile and can be implemented according to the attacker's desire and suited scenario, for example, the attack presented in [26] can also use our compromised tool flow to hide the malicious LUT Trojan even in the bitstream to evade bitstream-level detection mechanisms. Also, in literature, a lot of effort has been given to design a Trojan which could assist in leaking the secret key through side channels [24, 57, 96]. This requires huge amount of power traces to be computed to model the attack and therefore it may take days to correctly leak the secret key. Additionally an expensive setup such as digital oscilloscope is required for leakage assessment from the device. However, our attack is subtle yet effective in providing the attacker with a means to leak the secret key directly through the primary outputs at runtime, i.e., opening a covert channel. On the other hand, if we discuss the detection of sneaky Trojans, a trigger-less implementation of a Trojan in our attack renders pre-and-post-synthesis detection mechanisms based on static or dynamic trigger

characteristic ineffective [12–14]. Power side channel analysis techniques inspect and visualize the power traces obtained from the design running on the device with the golden one to detect the Trojan. In our key leakage example no logic is implemented as trigger, and the payload consists of only a few gates, i.e., eight 4×1 `LUTs`, in contrast to 49 gates in [57], hence the extra power consumption by an infected design should have negligible effect to the noise power levels as compared to the original design thus potentially evades the most of the detection techniques relying of side-channel analysis [97, 101–103].

To reveal our attack, post-configuration methods such as the readback feature of the FPGA, if enabled, can be used at the consumer side to read out the bitstream and apply verification mechanisms such as functional equivalence checking using PCH. However, `ReadBack` command is disabled in FPGAs to avoid the unauthorized readout of a design [25] for security reasons.

4.5 Chapter Conclusion

In this chapter, we have presented a novel attack that is based on the malicious routing of an inserted Trojan circuit to retain a dormant state even in the bitstream for a reconfigurable hardware device. For the information-leaking variant, the inserted Trojan circuit in our approach does not even need any trigger logic as the payload is maliciously routed to the primary outputs. In all variants, the last programmable interconnect point (PIP) connecting the payload to a trigger or an enable signal is removed during place-and-route and is established again only at the very last step when the FPGA is being programmed, thus activating the Trojan circuit. We have demonstrated our attack on an 8-bit datapath AES-128 design to successfully read out the first eight key bits of an AES module. The conventional testing and verification methods presented at the RTL or gate level so far cannot prevent or detect our proposed attack as the Trojan is injected in a post-synthesis step of the design flow, thus circumventing conventional testing and verification methods. Furthermore, we have shown that our Trojan circuit stays unconnected in the bitstream, there by evading even bitstream-level verification techniques.

Chapter 5

Conclusion

The research contributions presented in this thesis have effectively broadened the scope of hardware security in reconfigurable systems, especially, with regards to the hardware Trojan detection method employed at the bitstream level to counter the Trojans that evade standard verification methods due to their property of adapting the passive state in every stage the field-programmable gate array (FPGA) design flow and turning into the active state only in the bitstream. Ideally, an intellectual property (IP) core must be verified against malicious inclusion at each of the design stages during its development. However, due to plenty of potential attack vectors possible at each stage with distinct properties of being hidden, no single solution exists so far which could be applied to counteract the multifaceted attacks or even nullify the effect of those attacks. For example, the dynamic verification methods applied at register-transfer level (RTL) of a design (cp. Section 2.3) may detect the faults and errors perfectly, but deliberately inserted logic by an attacker could mimic the original logic of the design, i.e., having the same attribute as the original circuit or playing its part to pass the signal to the next level, may deceive the simulation-based testing and verification techniques. Also, due to the lack of a standard way to verifying the design after the RTL or gate level, i.e., placement and routing and even a bitstream level, the attacker would have a large space during these steps to insert malicious logic which would likely go undetected. The bitstream-level verification was not possible for many commercial devices due to the lack of publicly available documentation of bitstream formats which are copyrights of particular vendors. However, the availability of the complete open-source tool flow for commercial FPGAs urged us to render the feasibility of verifying the design at the bitstream level and detecting any malicious inclusion in the bitstream configuration file before loading it to the device for configuration.

We have proposed bitstream-level proof-carrying hardware (PCH) approach in Chapter 3 which provides an effective way of detecting the

Trojans in the bitstream by offering the security guarantees in the form of certificates (proofs) considering the manifold of malicious parties involved during the development and the consignment of the design. We have used *IceStorm*, an open-source design flow for Lattice FPGAs, to implement the stealthy malicious lookup table (LUT) attack and then successfully detected using a bitstream-level PCH approach (cp. Section 3.3) by integrating the *IceStorm* tool flow with the PCH tool flow for iCE40-HX1K FPGAs. We thus believe that if the bitstream format would be openly accessible, there would be even more opportunities for the reconfigurable system's end-user to verify the design against malicious inclusions, i.e., hardware Trojans. Verily, our proposed PCH approach already provides a powerful solution against the RTL and bitstream-level Trojans, for both the developer of the design and the end-user, therefore renders strong grounds to be adapted for the bitstream verification for commercial FPGAs.

The research on hardware Trojans from an attacker as well as defender's perspective presents a valuable improvement to provide security and trust in reconfigurable hardware. The defender would only be able to develop a robust system for securing a device as long as the attack is known to him. However, for unknown attacks, the device may still be susceptible under trusted conditions. So, the hardware design verification and trust methods mainly depend on both the attacker's and defender's interpretations, i.e., the defender would update the security properties of a system according to the new attack launched by the attacker. To contribute further in this direction, we have presented a new kind of hardware Trojan attack in Chapter 4 that circumvents all the conventional pre-configuration design-time and even bitstream-level verification techniques for FPGAs.

The attack is carried out in two phases: In Phase-1, the Trojan is inserted after the design synthesis step, i.e., during the placement and routing step, attached to the original circuit to get the fully placed-and-routed design which is then disconnected from the original circuit by flipping only one programmable interconnect point (PIP) bit, the Trojan PIP (TPIP), while writing the design into the text file with the help of compromised placement-and-routing tool (cp. Figure 4.5). The generated bitstream configuration file would then be carrying an unconnected Trojan along with the original design. This step thus makes sure for the successful insertion of the Trojan that would remain undetected in a bitstream verification process due to its disconnection from the original circuit via (T)PIP. In Phase 2, the connection is re-established by the compromised

programming tool when reading the bitstream of the design to configure the device. This is essentially done by flipping the corresponding TPIP bit, communicated by the design flow, again to restore the connection between the original circuit and the Trojan circuit, therefore directly activating the Trojan or making it possible for an attacker to activate the Trojan (cp. Figure 4.6). We have also demonstrated the feasibility of the attack in Section 4.3 by successfully leaking the 8-bits of a secret key of an AES-128 core, implemented on iCE40-HX1K FPGA, via an unused I/O pin.

To summarize, through the research presented in this thesis, we have been able to contribute to enhancing the security and trust-building in reconfigurable computing with respect to both, the defender's and the attacker's perspective. Our bitstream-level proof-carrying hardware (PCH) approach to detect the stealthy malicious LUT hardware Trojan is a true example of a defender's frame of mind which can be employed to verify the Lattice iCE40 FPGAs bitstreams against hardware Trojan's threat, therefore, creating an opportunity for the trust-building through the PCH certified designs running directly on the reconfigurable hardware. Furthermore, the malicious routing-based hardware Trojan signifies the attacker's point of view to carry out the attack by using the compromised design flow tools, which remains undetectable even by bitstream-verification methods thus opens a new research challenge to the hardware design and verification teams.

Chapter 6

Outlook

This chapter mainly points out some of the possible future directions to counter the attack presented in Chapter 4 by extending the work presented in Chapter 3. Moreover, some of the new attacks and their possible mitigation ideas are also discussed in the following:

1. *Enabling Post-Configuration Verification for FPGAs:* With the introduction of the malicious routing-based Trojan attack in Chapter 4, the pre-configuration verification techniques for field-programmable gate array (FPGA)'s bitstream have been jeopardized badly. This might be a good step to bring in the post-configuration verification methods for the detection of Trojans that only activate in the FPGA device. One possible countermeasure

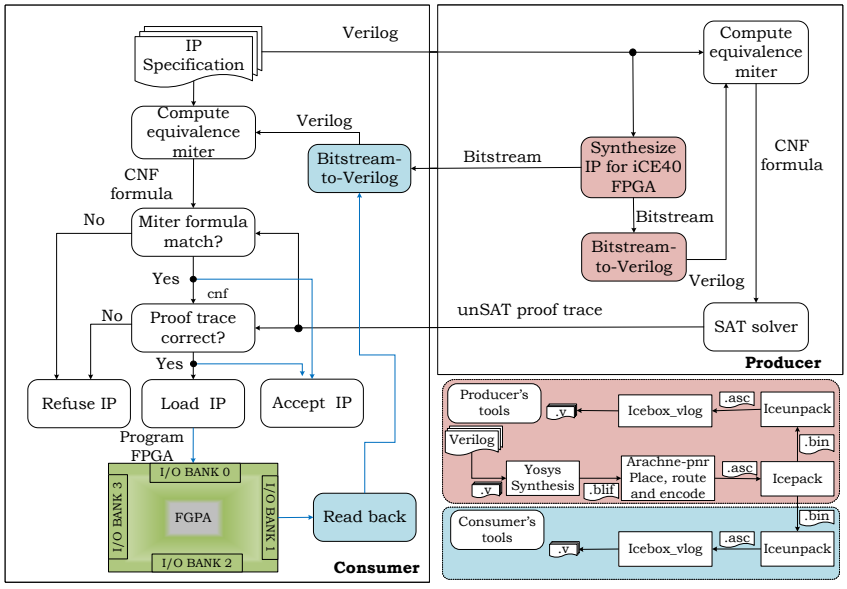

Figure 6.1: Post-configuration verification for FPGAs using PCH.

to detect such Trojans could be utilizing the read-back option in FPGAs to retrieve a complete functional bitstream file and then applying the proof-carrying hardware (PCH) as a post configuration

verification methodology, as explained in Section 3.3 and illustrated in Figure 6.1, with an additional miter formation using the retrieved bitstream. Since the Trojan would be connected to the original circuit during configuration, the miter formed by using the bitstream retrieved after configuration would not match the miter obtained by the original specification at the consumer end, consequently, the malicious routing-based Trojan attack would be revealed. However, if the Trojan is triggered externally by an attacker, i.e., by applying a voltage to the specific input pin, the probability to detect the Trojan would be low during verification, since it would be tricky to guess the input pin from a large number of available input/output (I/O) pins to which the voltage might be applied to activate the Trojan. Moreover, checking each unused I/O pin in a larger designs would demand plenty of time to expose the attack.

2. *Malicious Bitstream:* The malicious routing-based attack presented in Section 4.3 can be improved in terms of the communication between the compromised tools for sharing the TPIP locations/addresses. An attacker could essentially hide the database of TPIP locations to the free spaces available in the bitstream itself, for example, an array with a precise set of rules which consists of one TPIP for each of the outputs. The is feasible since, in most of the designs implemented on larger FPGAs, the significant area of a bitstream remains unconfigured which can be exploited by the attacker. Modification of a bitstream to hide any additional information or a circuit to its unused spaces is plausible and is shown in [75]. While programming the FPGA device, the programmer or the malicious programming tool will try to find the first TPIP from the array/-database in a bitstream that is not yet configured, if found, it will be configured to activate the Trojan. Figure 6.2 demonstrates the attack with respect to the attacker in the design house and a malicious programmer/programming tool in the SoC integration site, where the former inserts and disconnects the Trojan on certain TPIP locations and embeds the addresses of those TPIP locations into the unused spaces of the bitstream and sends to the latter which identifies the TPIP locations and activates the Trojan.

   All other intermediate locations of TPIP that are not used by the actual design are arbitrarily configured which would not have any effect on the functionality of the original circuit. Note that, if the attack is carried out with more than one TPIPs, those TPIPs in the

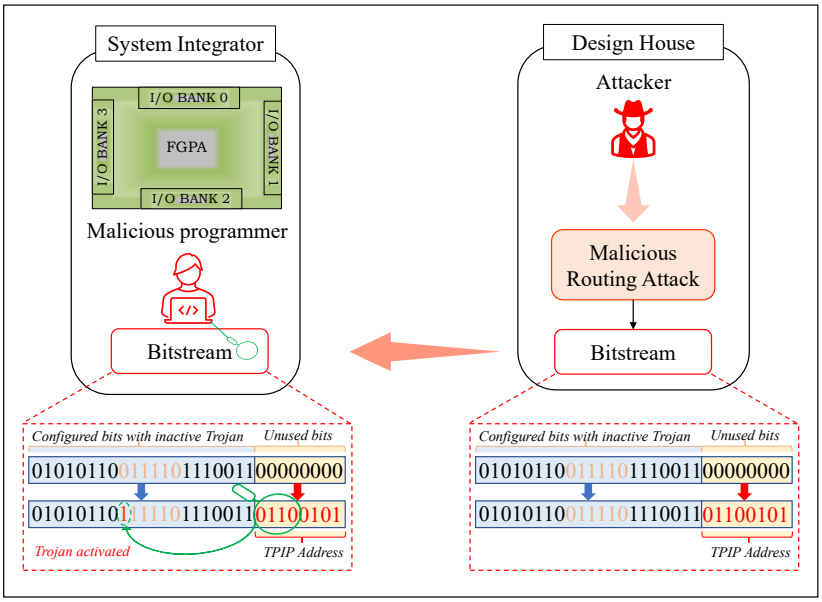

Figure 6.2: TPIP addresses embedded within the bitstream for malicious programmer.

array are left unconfigured by the attacker for facilitating the malicious programmer to find them. This would potentially exclude the communication dependencies between the compromised tools to carry out the successful attack.

3. *Malicious Programmer in Multi-tenant FPGAs:* The partial reconfiguration feature in FPGAs has allowed the users to share a single fabric among different application designs for gaining the maximum resource utilization, especially in resource starving environments, e.g., cloud computing. This property along with the normal configurations in FPGAs have preceded them towards the concept of multi-tenancy in which the FPGA is shared among various users in the cloud, this is also referred to as multi-tenant FPGA devices. Although the tenants share a single FPGA device to implement their logic, there is a logical separation provided at different regions to keep the privacy between them.

   However, such type of model, i.e., FPGA-as-a-Service, in which the resources are explicitly disclosed to cloud tenants, might raise certain security concerns. For example, different tenants despite logically isolated, share the same supply voltage that is not physically isolated, thus potentially exploitable to remote attacks such as remote power analysis attacks to leak cryptographic key [104],

and denial-of-service (DoS) attacks [105, 106]. Nevertheless, a malicious programmer or a subverted programming model, as described in Chapter 4, for single FPGAs could be applied to multi-tenant FPGAs which can provide valuable insights to the attacker through the malicious IP (receptor), by extracting random information from a victim IP in a shared environment. At this point, the programmer would know which resources will be going to be used by which IP, thus would have a power to connect the malicious IP to the isolated IP, but would not know about the internal details of the IP to carry out the full attack. However, the acquired data from a victim's IP can be processed offline to get valuable information. For instance, if the victim IP is an encryption core, the programmer could potentially attempt to read out the states of the flip-flops (FFs) that are configured in the victim IP to process the key bits that are being used. This attack would potentially be more powerful regarding the fact that it will directly provide the way of analyzing the registers state to guess the cryptographic key in contrast to the power analysis attack [104].

4. *PCH4HLS: Detection of Hardware Trojans in High-level Synthesis using PCH:* In recent years, with the development of high-level synthesis (HLS) being another design representation to RTL, the third-party IP vendors are advancing towards the provision of SystemC HLS designs as it potentially provides the flexibility to create many alternate versions of the same design. The SystemC design provided at the HLS abstraction in the FPGA design flow is generally converted automatically to its equivalent RTL in a much shorter amount of time using HLS tools. However, the acceptance of this abstraction level in hardware design flow has opened another challenge for design trust and security. A Trojan inserted into the HLS design, could go deep into the levels without being observed by the other levels, if not detected at the HLS level. One such benchmark of HLS Trojans has been presented in [107]. Unfortunately, this level of abstraction has not got much attention concerning security issues, especially hardware Trojan attacks, which are as important to address as at RTL or other levels of hierarchy in the FPGA design flow. We propose to extend a proof-carrying hardware (PCH) approach to one level above in the abstraction for the detection of HLS Trojans. The current variant of PCH, (cp. Section 3.3) takes the RTL design as

an input, however, an open-source HLS tool LegUp could be integrated into PCH flow which will convert the SystemC representation of the design to its comparable RTL. Later, the typical PCH flow can be followed to verify the design against hardware Trojans.

5. *Verilog2Bitstream Verification: A Multi-level approach for hardware Trojan detection in FPGAs:* The research presented in recent times to detect hardware Trojan in FPGAs has been mostly targeted to a certain level of abstraction in a design flow hierarchy. This is mainly due to the variations in the implementation of hardware Trojan designs and their insertion and activation mechanisms into the original design. For instance, Trojan detection at the register-transfer level (RTL) may not be applicable to the lower levels, such as gate level and bitstream level. To overcome this limitation, a multi-level Trojan detection approach can be offered which may incorporate the outcomes acquired during the previous levels until the last level, e.g., bitstream level. The detection at each level would decrease the probability of the Trojan being hidden, therefore, providing a unique and enhanced framework to verify the IP cores against Trojans from Verilog code to bitstream.

6. *MAAS: Malicious Approximate Accelerator Synthesis:* Approximate computing (AC) has emerged as an effective alternative for performance improvement in computing systems. AC targets the intrinsic error resilience present in several applications, e.g., image and signal processing, computer vision, and machine learning. Approximate computing has been largely applied to generate hardware accelerator circuits that offer substantial benefits of power consumption, area, and delay while occasionally providing erroneous yet acceptable outputs.

   Existing works in approximate circuit (AxC) synthesis provide various automated frameworks, such as the one proposed in [108], which accepts the original circuit description along with an error bound defined by the user and generates an approximate version of the circuit that adheres to the given bounds. The main target for approximation is arithmetic components, such as adders and multipliers. However, the security breaches due to inclusions of third-party's malicious arithmetic components in a circuit have not been investigated yet. An automated approximate circuit synthesis flow such as the MCTS-based framework in [108] and the CIRCA

framework [109] could be compromised either due to the addition of maligned logic or a bad employee binding a Trojan during the automated flow. The threat model illustrated in the Fig-

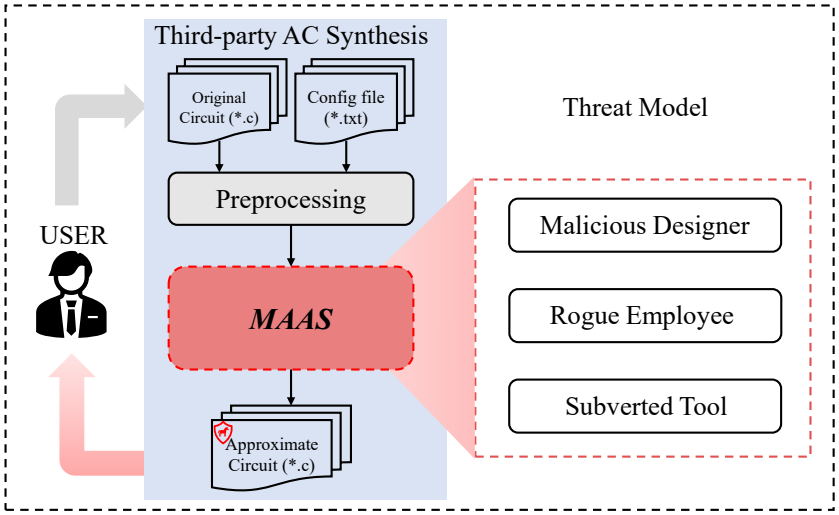

Figure 6.3: Threat model for malicious approximation.

ure 6.3 throws light on the malicious approximate accelerator synthesis (MAAS) by a third party. The user provides the design specifications (usually in SystemC) along with the required error metrics, i.e., a configuration file (in text) to an approximate accelerator synthesis provider who first runs some preprocessing and then generates the approximate versions of the design and synthesizes the circuit. In our threat model, we consider that the approximate accelerator synthesis provider is itself trusted, however, a Trojan can be inserted by one of the following entities: malicious designer, a rogue employee, or a subverted accelerator synthesis tool/framework. After the synthesis, the design is sent back to the user who runs the test vectors to verify the according to the given constraints, such as error metrics, but there is no verification performed by the user against potential hardware Trojans. To investigate the aforementioned threat model, the approximate circuit synthesis flows proposed in [108, 109] could be manipulated with malicious approximate components (Trojan), which likely fulfills the constraints such as error and performance parameters, but could lead to catastrophic results later when the Trojan is activated. Moreover, the Trojan insertion has been so far targeted at the netlist level whereas the state-of-the-art approximate circuit synthesis frameworks, i.e., [108] target higher levels of abstraction such as SystemC. Therefore, considering a higher level of abstraction to evaluate hardware Trojan

insertion in an automated AxC synthesis flow could be a valuable contribution to the AC and thus preventive measures could be proposed to alert the system-on-chip (SoC) integrator about the malicious logic before integrating it into a system.

List of Tables

Listings

List of Algorithms

List of Figures

List of Abbreviations

AC Approximate Computing 75, 76

ADAS Advanced Driver Assisted Systems 1

AES Advanced Encryption Standard 5, 8, 29, 55–59, 62

AES-128 AES With Key Size Of 128 Bits 59, 63, 67

AI Artificial Intelligence 1

ASCII American Standard Code For Information Interchange 40

ASIC Application-specific Integrated Circuit 2, 4, 9, 17, 23, 24, 29

ASICs Application-specific Integrated Circuits 2, 4, 5, 15, 24, 26, 29, 52

AxC Approximate Circuit 76

BLE Basic Logic Element 12

BLEs Basic Logic Elements 12, 13

BLIF Berkeley Logic Interchange Format 40

BRAM Block RAM 11

CAD Computer-aided Design 16

CEC Combinational Equivalence Checking 41

CLB Configurable Logic Block 13, 14

CLBs Configurable Logic Blocks 9, 11, 12, 26

CNF Conjunctive Normal Form 39, 40

COTS Commercial Off-the-shelf 3, 19

CPLDs Complex-programmable Logic Devices 10

CPU Central Processing Unit 46, 58

CRC Cyclic Redundancy Checksum 61

DARPA Defense Advanced Research Projects Agency 3

DCM Digital Clock Manager 26

DES Data Encryption Standard 27

DL Deep Learning 1

DoS Denial-of-service 18, 34, 74

DSP Digital Signal Processing 12

EDA Electronic Design Automation 2, 4, 7, 8, 15–17, 19, 23, 33, 34, 37, 43, 46, 49, 52, 53, 58, 65

EM Electromagnetic 63

EPROM Erasable Programmable Read-only Memory 11

FF Flip-flop 12, 13, 58

FFs Flip-flops 22, 46, 62, 63, 74

FPDs Field-programmable Devices 10

FPGA Field-programmable Gate Array 1, 7, 9, 10, 12, 13, 23–25, 29, 34, 40, 52, 53, 67, 69, 70, 73

FPGAs Field-programmable Gate Arrays 1, 2, 4, 8, 9, 11, 24, 25, 65, 69, 73, 74

GDSII Graphic Database System Information Interchange 23

GPI/O General-purpose Input/output 62

HDL Hardware Description Language 16, 40, 41, 53

HLS High-level Synthesis 75

I/O Input/output 11–15, 23, 50, 56, 57, 59–61, 63, 70, 73

IC Integrated Circuit 2–4, 9, 17, 19, 21

ICs Integrated Circuits 1, 3, 17, 29

IEEE Institute Of Electrical And Electronics Engineers 3

IOB Input/output Block 15

IOBs Input/output Blocks 15

IoT Internet-of-things 1, 4

IP Intellectual Property 2, 4, 5, 16, 17, 19, 26–29, 32, 37, 41, 50, 52, 54, 69, 74, 75

IT Information Technology 1

LAB Logic Array Block 13

LED Light Emitting Diode 48, 61

LSB Least Significant Bit 47

LUT Lookup Table 5, 6, 13, 34, 36, 37, 44, 47, 50, 58, 64, 65, 67, 69, 70

LUTs Lookup Tables 10, 17, 24, 26, 46, 62

MAC Media Access Controller 27

MiM Man-in-the-middle 33

ML Machine Learning 54

MPGAs Masked Programmable Gate Arrays 9

MSB Most Significant Bit 47

MUX Multiplexer 12, 13, 55, 56, 60

MUXes Multiplexers 17, 56

nW Nano-Watt 63

OS Operating System 18, 46, 58

PAL Programmable Array Logic 10

PALs Programmable Arrays Logic 10

PCB Proof-carrying Bitstream 37, 42, 44

PCC Proof-carrying Code 37

PCH Proof-carrying Hardware 6, 7, 35, 37, 38, 40, 42, 46, 47, 49–52, 54, 65, 67, 69, 70, 73, 75

PIP Programmable Interconnect Point 56, 65, 67, 70

PIPs Programmable Interconnect Points 15, 53, 60

PIs Programmable Interconnects 11, 17

PLA Programmable Logic Array 9

PLD Programmable Logic Device 9

PLDs Programmable Logic Devices 10

PnR Placement-and-routing 52, 53, 56, 60, 65

POSs Product-of-sums 31

RAM Random Access Memory 46, 58, 62

RO Ring Oscillator 29

RTL Register-transfer Level 8, 16, 22, 30, 32, 38, 54, 67, 69, 70, 75

SAT Boolean Satisfiability 39, 46

SoC System-on-chip 4, 16, 19, 29, 37, 52, 76

SOPs Sum-of-products 31

SPLDs Simple Programmable Logic Devices 10

SRAM Static Random Access Memory 11, 26

TERO Transient Effect Ring Oscillator 29

TPIP Trojan PIP 15, 53–57, 60, 70, 73, 74

UCI Unused Circuit Identification 30, 42

Author's Publications

[1] Qazi Arbab Ahmed, Tobias Wiersema, and Marco Platzner., "Proof-Carrying Hardware Versus the Stealthy Malicious LUT Hardware Trojan". In *Applied Reconfigurable Computing* (ARC) 2019. Ed. by Christian Hochberger et al. Cham: Springer International Publishing, 2019, pp. 127–136. ISBN : 978-3-030-17227-5. URL : https://doi.org/10.1007/978-3-030-17227-5_10.

[2] Qazi Arbab Ahmed, Tobias Wiersema, and Marco Platzner., "Malicious Routing: Circumventing Bitstream-level Verification for FPGAs". In: *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2021, pp. 1490–1495. URL : https://doi.org/10.23919/DATE51398.2021.9474026.

[3] Qazi Arbab Ahmed., "Hardware Trojans in Reconfigurable Computing". In: *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*, 2021, pp. 1-2. URL : https://doi.org/10.1109/VLSI-SoC53125.2021.9606974.

Bibliography

[1] S. Adee. "The Hunt For The Kill Switch". In: *IEEE Spectrum* 45.5 (2008), pp. 34–39. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2008.4505310.

[2] *Ethical Hackers Sabotage F-15 Fighter Jet, Expose Serious Vulnerabilities. [Online]*. URL: {https://www.newsweek.com/cybersecurity-vulnerability-fighter-jet-f15-defcon-hacking-tads-flight-system-hack-pentagon-1454491}.

[3] K. Xiao et al. "Hardware Trojans: Lessons Learned After One Decade of Research". In: *ACM Trans. Des. Autom. Electron. Syst.* 22.1 (May 2016), 6:1–6:23. ISSN: 1084-4309. DOI: 10.1145/2906147.

[4] M. Tehranipoor et al. "Trustworthy Hardware: Trojan Detection and Design-for-Trust Challenges". In: *Computer* 44.7 (July 2011), pp. 66–74. ISSN: 0018-9162. DOI: 10.1109/MC.2010.369.

[5] Francis Wolff et al. "Towards Trojan-Free Trusted ICs: Problem Analysis and Detection Scheme". en. In: *2008 Design, Automation and Test in Europe*. Munich, Germany: IEEE, Mar. 2008, pp. 1362–1365. ISBN: 978-3-9810801-3-1 978-3-9810801-4-8. DOI: 10.1109/DATE.2008.4484928. URL: http://ieeexplore.ieee.org/document/4484928/ (visited on 11/25/2019).

[6] 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*. 2009, pp. 166–171. DOI: 10.1109/HLDVT.2009.5340158.

[7] Swarup Bhunia et al. "Hardware Trojan Attacks: Threat Analysis and Countermeasures". en. In: *Proceedings of the IEEE* 102.8 (Aug. 2014), pp. 1229–1247. ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2014.2334493. URL: http://ieeexplore.ieee.org/document/6856140/ (visited on 11/25/2019).

[8] Samuel T. King et al. "Designing and Implementing Malicious Hardware". In: *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*. LEET'08. San Francisco, California: USENIX Association, 2008.

[9] Jie Zhang and Qiang Xu. "On hardware Trojan design and implementation at register-transfer level". en. In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. Austin, TX, USA: IEEE, June 2013, pp. 107–112. ISBN: 978-1-4799-0601-7 978-1-4799-0559-1 978-1-4799-0600-0. DOI: 10.1109/HST.2013.6581574. URL: http://ieeexplore.ieee.org/document/6581574/ (visited on 11/25/2019).

[10] C. Sturton et al. "Defeating UCI: Building Stealthy and Malicious Hardware". In: *2011 IEEE Symposium on Security and Privacy*. 2011, pp. 64–77. DOI: 10.1109/SP.2011.32.

[11] Ramesh Karri et al. "Trustworthy Hardware: Identifying and Classifying Hardware Trojans". en. In: *Computer* 43.10 (Oct. 2010), pp. 39–46. ISSN: 0018-9162. DOI: 10.1109/MC.2010.299. URL: http://ieeexplore.ieee.org/document/5604161/ (visited on 11/25/2019).

[12] M. Hicks et al. "Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically". In: *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 159–172. DOI: 10.1109/SP.2010.18.

[13] J. Zhang et al. "VeriTrust: Verification for hardware trust". In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, May 2013, pp. 1–8. DOI: 10.1145/2463209.2488808.

[14] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. "FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany: ACM, 2013, pp. 697–708. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516654. URL: http://doi.acm.org/10.1145/2508859.2516654.

[15] Intel® FPGAs. *Automotive FPGA Applications. [Online]*. URL: {https://www.intel.com/content/www/us/en/automotive/products/programmable/applications.html/}.

[16] Kaiyuan Guo et al. "[DL] A Survey of FPGA-Based Neural Network Inference Accelerators". In: *ACM Trans. Reconfigurable Technol. Syst.* 12.1 (2019). ISSN: 1936-7406. DOI: 10.1145/3289185. URL: https://doi.org/10.1145/3289185.

[17] Amazon.com Inc. *Amazon EC2 F1 Instances. [Online]*. URL: {https://aws.amazon.com/ec2/instance-types/f1/}.

[18] *Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances. [Online].* URL: {https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057} (visited on 2018).

[19] Huawei Cloud. *FPGA Accelerated Cloud Server (FACS). [Online].* URL: {https://www.huaweicloud.com/en-us/product/fcs.html} (visited on 2020).

[20] Andrew Putnam et al. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), 13–24. ISSN: 0163-5964. DOI: 10.1145/2678373.2665678. URL: https://doi.org/10.1145/2678373.2665678.

[21] Yier Jin, Nathan Kupp, and Y. Makris. "Experiences in Hardware Trojan design and implementation". In: *2009 IEEE International Workshop on Hardware-Oriented Security and Trust* (2009), pp. 50–57.

[22] Sebastian Wallat et al. "A look at the dark side of hardware reverse engineering - a case study". In: *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*. 2017, pp. 95–100. DOI: 10.1109/IVSW.2017.8031551.

[23] Seyedeh Sharareh Mirzargar and Mirjana Stojilovic. "Physical Side-Channel Attacks and Covert Communication on FPGAs: A Survey". en. In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. Barcelona, Spain: IEEE, 2019, pp. 202–210. ISBN: 978-1-72814-884-7. DOI: 10.1109/FPL.2019.00039. URL: https://ieeexplore.ieee.org/document/8892083/ (visited on 06/02/2021).

[24] Maik Ender et al. "The First Thorough Side-Channel Hardware Trojan". In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Cham: Springer International Publishing, 2017, pp. 755–780. ISBN: 978-3-319-70694-8.

[25] Stephen M. Trimberger and Jason J. Moore. "FPGA Security: Motivations, Features, and Applications". In: *Proceedings of the IEEE* 102.8 (2014). Conference Name: Proceedings of the IEEE, pp. 1248–1265. ISSN: 1558-2256. DOI: 10.1109/JPROC.2014.2331672.

[26] C. Krieg, C. Wolf, and A. Jantsch. "Malicious LUT: A stealthy FPGA Trojan injected and triggered by the design flow". In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–8. DOI: 10.1145/2966986.2967054.

[27] Clifford Wolf and Mathias Lasser. *Project IceStorm*. URL: {http://www.clifford.at/icestorm/}.

[28] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. "Architecture of field-programmable gate arrays". In: *Proceedings of the IEEE* 81.7 (1993), pp. 1013–1029. DOI: 10.1109/5.231340.

[29] Stephen D. Brown et al. "Chapter 1 - Introduction to FPGAs". In: *Field-Programmable Gate Arrays*. Boston, MA: Springer US, 1992, pp. 1–11. ISBN: 978-1-4615-3572-0. DOI: 10.1007/978-1-4615-3572-0_1. URL: https://doi.org/10.1007/978-1-4615-3572-0_1.

[30] W. Carter et al. "A user programmable reconfigurable logic array". In: 1986.

[31] Stephen M. Trimberger. "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology". In: *Proceedings of the IEEE* 103.3 (2015), pp. 318–331. DOI: 10.1109/JPROC.2015.2392104.

[32] Niccolò Battezzati, Luca Sterpone, and Massimo Violante. "Chapter 1 - Introduction". In: *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications*. New York, NY: Springer New York, 2011, pp. 1–4. ISBN: 978-1-4419-7595-9. DOI: 10.1007/978-1-4419-7595-9_1. URL: https://doi.org/10.1007/978-1-4419-7595-9_1.

[33] Katherine Compton and Scott Hauck. "Reconfigurable Computing: A Survey of Systems and Software". In: *ACM Comput. Surv.* 34.2 (2002), 171–210. ISSN: 0360-0300. DOI: 10.1145/508352.508353. URL: https://doi.org/10.1145/508352.508353.

[34] J. Greene, E. Hamdy, and S. Beal. "Antifuse field programmable gate arrays". In: *Proceedings of the IEEE* 81.7 (1993), pp. 1042–1056. DOI: 10.1109/5.231343.

[35] Xilinx Inc. *7 Series FPGAs Configurable Logic Block*. URL: {https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf}(Sept.2016).

[36] Clive Max Maxfield. "Chapter 2 - FPGA Architectures". In: *FPGAs: Instant Access*. Ed. by Clive Max Maxfield. Instant Access. Burlington: Newnes, 2008, pp. 13–48. ISBN: 978-0-7506-8974-8. DOI: https://doi.org/10.1016/B978-0-7506-8974-8.00002-8. URL: https://www.sciencedirect.com/science/article/pii/B9780750689748000028.

[37] J. Rose and S. Brown. "Flexibility of interconnection structures for field-programmable gate arrays". In: *IEEE Journal of Solid-state Circuits* 26 (1991), pp. 277–282.

[38] Vaughn Betz and Jonathan Rose. "FPGA routing architecture: Segmentation and buffering to optimize speed and density". In: 1999, pp. 59–68.

[39] Mark L. Chang. "Chapter 1 - Device Architecture". In: *Reconfigurable Computing*. Ed. by Scott Hauck and André Dehon. Systems on Silicon. Burlington: Morgan Kaufmann, 2008, pp. 3–27. DOI: https://doi.org/10.1016/B978-012370522-8.50005-4. URL: https://www.sciencedirect.com/science/article/pii/B9780123705228500054.

[40] Nisha Jacob Kabakci. "Hardware Trojans and their Security Impact on Reconfigurable System-on-Chips". PhD thesis. Munich, Germany, 2019.

[41] Jason Anderson et al. "A Placement Algorithm for FPGA Designs with Multiple I/O Standards". In: *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*. Ed. by Reiner W. Hartenstein and Herbert Grünbacher. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 211–220. ISBN: 978-3-540-44614-9.

[42] Xilinx Inc. *Vivado® Design Suite Documentation [Online]*. URL: {https://www.xilinx.com/products/design-tools/vivado.html?resultsTablePreSelect=xlnxdocumenttypes:SeeAll#documentation}.

[43] Steven A. Guccione. "Chapter 19 - Configuration Bitstream Generation". In: *Reconfigurable Computing*. Ed. by Scott Hauck and André Dehon. Systems on Silicon. Burlington: Morgan Kaufmann, 2008, pp. 401–409. DOI: https://doi.org/10.1016/B978-012370522-8.50026-1. URL: https://www.sciencedirect.com/science/article/pii/B9780123705228500261.

[44] Jason Cong and Peichen Pan. "Chapter 13 - Technology Mapping". In: *Reconfigurable Computing*. Ed. by Scott Hauck and André Dehon. Systems on Silicon. Burlington: Morgan Kaufmann, 2008, pp. 277–296. DOI: https://doi.org/10.1016/B978-012370522-8.50019-4. URL: https://www.sciencedirect.com/science/article/pii/B9780123705228500194.

[45] K. Xiao et al. “Hardware Trojans: Lessons Learned After One Decade of Research”. In: *ACM Trans. Des. Autom. Electron. Syst.* 22.1 (2016), 6:1–6:23. ISSN: 1084-4309. DOI: 10.1145/2906147.

[46] M. Tehranipoor and C.Wang. *Introduction to Hardware Security and Trust*. Springer, 2012.

[47] S. Bhunia et al. “Hardware Trojan Attacks: Threat Analysis and Countermeasures”. In: *Proceedings of the IEEE* 102.8 (2014), pp. 1229–1247. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2334493.

[48] Carl E. Landwehr et al. “A Taxonomy of Computer Program Security Flaws”. In: *ACM Comput. Surv.* 26.3 (Sept. 1994), pp. 211–254. ISSN: 0360-0300. DOI: 10.1145/185403.185412. URL: http://doi.acm.org/10.1145/185403.185412.

[49] Qazi Arbab Ahmed, Tobias Wiersema, and Marco Platzner. “Malicious Routing: Circumventing Bitstream-level Verification for FPGAs”. In: *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2021, pp. 1490–1495. DOI: 10.23919/DATE51398.2021.9474026.

[50] M. Tehranipoor et al. “Trustworthy Hardware: Trojan Detection and Design-for-Trust Challenges”. In: *Computer* 44.7 (2011), pp. 66–74. ISSN: 0018-9162. DOI: 10.1109/MC.2010.369.

[51] Bicky Shakya et al. “Benchmarking of Hardware Trojans and Maliciously Affected Circuits”. In: *Journal of Hardware and Systems Security* 1.1 (2017), pp. 85–102. ISSN: 2509-3436. DOI: 10.1007/s41635-017-0001-6. URL: http://dx.doi.org/10.1007/s41635-017-0001-6.

[52] M. Tehranipoor and F. Koushanfar. “A Survey of Hardware Trojan Taxonomy and Detection”. In: *IEEE Design Test of Computers* 27.1 (2010), pp. 10–25. ISSN: 0740-7475. DOI: 10.1109/MDT.2010.7.

[53] S. Mal-Sarkar et al. “Design and Validation for FPGA Trust under Hardware Trojan Attacks”. In: *IEEE Transactions on Multi-Scale Computing Systems* 2.3 (2016), pp. 186–198. ISSN: 2332-7766. DOI: 10.1109/TMSCS.2016.2584052.

[54] Xiaoxiao 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*. 2008, pp. 15–19. DOI: 10.1109/HST.2008.4559039.

[55] Steve Trimberger. "Trusted Design in FPGAs". In: *Proceedings of the 44th Annual Design Automation Conference*. DAC '07. San Diego, California: ACM, 2007, pp. 5–8. ISBN: 978-1-59593-627-1. DOI: 10.1145/1278480.1278483. URL: http://doi.acm.org/10.1145/1278480.1278483.

[56] Ryan Kastner and Ted Huffmire. "Threats and Challenges in Reconfigurable Hardware Security". In: *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas, NV*. 2008.

[57] Lang Lin, Wayne Burleson, and Christof Paar. "MOLES: malicious off-chip leakage enabled by side-channels". en. In: *Proceedings of the 2009 International Conference on Computer-Aided Design - ICCAD '09*. San Jose, California: ACM Press, 2009, p. 117. ISBN: 978-1-60558-800-1. DOI: 10.1145/1687399.1687425. URL: http://portal.acm.org/citation.cfm?doid=1687399.1687425 (visited on 11/25/2019).

[58] Amir Moradi et al. "On the Vulnerability of FPGA Bitstream Encryption against Power Analysis Attacks: Extracting Keys from Xilinx Virtex-II FPGAs". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA: Association for Computing Machinery, 2011, 111–124. ISBN: 9781450309486. DOI: 10.1145/2046707.2046722. URL: https://doi.org/10.1145/2046707.2046722.

[59] Amir Moradi and Tobias Schneider. "Improved Side-Channel Analysis Attacks on Xilinx Bitstream Encryption of 5, 6, and 7 Series". In: *Constructive Side-Channel Analysis and Secure Design*. Ed. by François-Xavier Standaert and Elisabeth Oswald. Cham: Springer International Publishing, 2016, pp. 71–87. ISBN: 978-3-319-43283-0.

[60] Pawel Swierczynski et al. "Physical Security Evaluation of the Bitstream Encryption Mechanism of Altera Stratix II and Stratix III FPGAs". In: *ACM Trans. Reconfigurable Technol. Syst.* 7.4 (Dec. 2014). ISSN: 1936-7406. DOI: 10.1145/2629462. URL: https://doi.org/10.1145/2629462.

[61] Gedare Bloom et al. "FPGA SoC architecture and runtime to prevent hardware Trojans from leaking secrets". In: *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2015, pp. 48–51. DOI: 10.1109/HST.2015.7140235.

[62] X. Zhang and M. Tehranipoor. "Case study: Detecting hardware Trojans in third-party digital IP cores". In: *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*. 2011, pp. 67–70. DOI: 10.1109/HST.2011.5954998.

[63] Trey Reece et al. "Stealth assessment of hardware Trojans in a microcontroller". In: *2012 IEEE 30th International Conference on Computer Design (ICCD)*. 2012, pp. 139–142. DOI: 10.1109/ICCD.2012.6378631.

[64] J. C. M. Santos and Y. Fei. "Designing and implementing a Malicious 8051 processor". In: *2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2012, pp. 63–66. DOI: 10.1109/DFT.2012.6378201.

[65] Yier Jin, Michail Maniatakos, and Yiorgos Makris. "Exposing vulnerabilities of untrusted computing platforms". In: *2012 IEEE 30th International Conference on Computer Design (ICCD)*. 2012, pp. 131–134. DOI: 10.1109/ICCD.2012.6378629.

[66] Michael Patterson et al. "A multi-faceted approach to FPGA-based Trojan circuit detection". In: *2013 IEEE 31st VLSI Test Symposium (VTS)*. 2013, pp. 1–4. DOI: 10.1109/VTS.2013.6548925.

[67] Nisha Jacob et al. "Compromising FPGA SoCs using malicious hardware blocks". In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*. Ed. by David Atienza and Giorgio Di Natale. IEEE, 2017, pp. 1122–1127. DOI: 10.23919/DATE.2017.7927157. URL: https://doi.org/10.23919/DATE.2017.7927157.

[68] Paris Kitsos and Artemios G. Voyiatzis. "FPGA Trojan Detection Using Length-Optimized Ring Oscillators". In: *2014 17th Euromicro Conference on Digital System Design*. 2014, pp. 675–678. DOI: 10.1109/DSD.2014.74.

[69] Paris Kitsos, Kyriakos Stefanidis, and Artemios G. Voyiatzis. "TERO-Based Detection of Hardware Trojans on FPGA Implementation of the AES Algorithm". In: *2016 Euromicro Conference on Digital System Design (DSD)*. 2016, pp. 678–681. DOI: 10.1109/DSD.2016.47.

[70] Jie Zhang, Feng Yuan, and Qiang Xu. "DeTrust: Defeating Hardware Trust Verification with Stealthy Implicitly-Triggered Hardware Trojans". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014,

153–166. ISBN: 9781450329576. DOI: 10.1145/2660267.2660289. URL: https://doi.org/10.1145/2660267.2660289.

[71] Masaru Oya et al. "A Score-Based Classification Method for Identifying Hardware-Trojans at Gate-Level Netlists". In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. DATE '15. Grenoble, France: EDA Consortium, 2015, 465–470. ISBN: 9783981537048.

[72] H. Salmani, M. Tehranipoor, and R. Karri. "On design vulnerability analysis and trust benchmarks development". In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 2013, pp. 471–474. DOI: 10.1109/ICCD.2013.6657085.

[73] H. Salmani. "COTD: Reference-Free Hardware Trojan Detection and Recovery Based on Controllability and Observability in Gate-Level Netlist". In: *IEEE Transactions on Information Forensics and Security* 12 (2017), pp. 338–350.

[74] George AF Seber. *Multivariate observations*. Vol. 252. John Wiley & Sons, 2009.

[75] Rajat Subhra Chakraborty et al. "Hardware Trojan Insertion by Direct Modification of FPGA Configuration Bitstream". In: *IEEE Design Test* 30.2 (2013), pp. 45–54. DOI: 10.1109/MDT.2013.2247460.

[76] Cyrus Peikari and Anton Chuvakin. *Security Warrior*. O'Reilly & Associates, Inc., 2004. ISBN: 0596005458.

[77] Adam Duncan et al. "FPGA Bitstream Security: A Day in the Life". en. In: *2019 IEEE International Test Conference (ITC)*. Washington, DC, USA: IEEE, 2019, pp. 1–10. ISBN: 978-1-72814-823-6. DOI: 10.1109/ITC44170.2019.9000145. URL: https://ieeexplore.ieee.org/document/9000145/ (visited on 08/22/2020).

[78] Behnam Khaleghi et al. "FPGA-Based Protection Scheme against Hardware Trojan Horse Insertion Using Dummy Logic". In: *IEEE Embedded Systems Letters* 7.2 (2015), pp. 46–50. DOI: 10.1109/LES.2015.2406791.

[79] Ken Thompson. "Reflections on Trusting Trust". In: *Commun. ACM* 27.8 (Aug. 1984), 761–763. ISSN: 0001-0782. DOI: 10.1145/358198.358210. URL: https://doi.org/10.1145/358198.358210.

[80] Qazi Arbab Ahmed, Tobias Wiersema, and Marco Platzner. "Proof-Carrying Hardware Versus the Stealthy Malicious LUT Hardware Trojan". In: *Applied Reconfigurable Computing*. Ed. by

Christian Hochberger et al. Cham: Springer International Publishing, 2019, pp. 127–136. ISBN: 978-3-030-17227-5. URL: https://doi.org/10.1007/978-3-030-17227-5_10.

[81] G. Necula and P. Lee. "Research on proof-carrying code for untrusted-code security". In: *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*. 1997, pp. 204–. DOI: 10.1109/SECPRI.1997.601335.

[82] S. Drzevitzky, U. Kastens, and M. Platzner. "Proof-Carrying Hardware: Towards Runtime Verification of Reconfigurable Modules". In: *2009 International Conference on Reconfigurable Computing and FPGAs*. IEEE, 2009, pp. 189–194. DOI: 10.1109/ReConFig.2009.31.

[83] Tobias Wiersema, Stephanie Drzevitzky, and Marco Platzner. "Memory Security in Reconfigurable Computers: Combining Formal Verification with Monitoring". In: *International Conference on Field-Programmable Technology (FPT 2014)*. IEEE, 2014, pp. 167–174. DOI: 10.1109/FPT.2014.7082771.

[84] *Lattice Semiconductor*. last visited on 21/11/2018. URL: {http://www.latticesemi.com/iCE40}.

[85] Tobias Isenberg et al. "Proof-Carrying Hardware via Inductive Invariants". In: *ACM Trans. Des. Autom. Electron. Syst.* 22.4 (July 2017). ISSN: 1084-4309. DOI: 10.1145/3054743. URL: https://doi.org/10.1145/3054743.

[86] Cotton Seed. *Arachne-pnr*. last visited on 15/11/2018. URL: {https://github.com/YosysHQ/arachne-pnr}.

[87] Maxime Cozzi, Jean-Marc Galliere, and Philippe Maurine. "Thermal Scans for Detecting Hardware Trojans". In: *Constructive Side-Channel Analysis and Secure Design*. Ed. by Junfeng Fan and Benedikt Gierlichs. Cham: Springer International Publishing, 2018, pp. 117–132. ISBN: 978-3-319-89641-0.

[88] Junghwan Yoon et al. "A Bitstream Reverse Engineering Tool for FPGA Hardware Trojan Detection". en. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto Canada: ACM, Oct. 2018, pp. 2318–2320. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3278487. URL: http://dl.acm.org/doi/10.1145/3243734.3278487 (visited on 01/16/2020).

[89] Tao Zhang et al. "A Comprehensive FPGA Reverse Engineering Tool-Chain: From Bitstream to RTL Code". en. In: *IEEE Access* 7 (2019), pp. 38379–38389. ISSN: 2169-3536. DOI: 10.1109/ACCESS.

2019.2901949. URL: https://ieeexplore.ieee.org/document/8653869/ (visited on 01/16/2020).

[90] Kento Hasegawa, Masao Yanagisawa, and Nozomu Togawa. "Hardware Trojans classification for gate-level netlists using multi-layer neural networks". en. In: *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. Thessaloniki, Greece: IEEE, July 2017, pp. 227–232. ISBN: 978-1-5386-0352-9. DOI: 10.1109/IOLTS.2017.8046227. URL: http://ieeexplore.ieee.org/document/8046227/ (visited on 11/25/2019).

[91] National Institute of Standards and Technology (NIST). *Advanced Encryption Standard (AES). [Online]*. URL: {https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf}.

[92] P. Hamalainen et al. "Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core". In: *9th EUROMICRO Conference on Digital System Design (DSD'06)*. 2006, pp. 577–583. DOI: 10.1109/DSD.2006.40.

[93] *8bit datapath AES. [Online]*. URL: {https://github.com/ChengluJin/8bit_datapath_AES}.

[94] Clifford Wolf. *Yosys Open SYnthesis Suite*. (last visited on 15/11/2018). URL: {http://www.clifford.at/yosys/}.

[95] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. "A Novel Technique for Improving Hardware Trojan Detection and Reducing Trojan Activation Time". en. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20.1 (2012), pp. 112–125. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2010.2093547. URL: http://ieeexplore.ieee.org/document/5678829/ (visited on 06/02/2021).

[96] Lang Lin et al. "Trojan Side-Channels: Lightweight Hardware Trojans through Side-Channel Engineering". en. In: *Cryptographic Hardware and Embedded Systems - CHES 2009*. Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 382–395. ISBN: 978-3-642-04137-2 978-3-642-04138-9. DOI: 10.1007/978-3-642-04138-9_27. URL: http://link.springer.com/10.1007/978-3-642-04138-9_27 (visited on 06/02/2021).

[97] Jiaji He et al. "Hardware Trojan Detection Through Chip-Free Electromagnetic Side-Channel Statistical Analysis". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017). Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, pp. 2939–2948. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2017.2727985.

[98] Lattice Semiconductor. *iCEcube2 Design Software. [Online]*. URL: {https://www.latticesemi.com/iCEcube2} (visited on 05/20/2021).

[99] X-T. Ngo et al. "Hardware Trojan detection by delay and electromagnetic measurements". In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015, pp. 782–787. DOI: 10.7873/DATE.2015.1103.

[100] K. Nielsen. *ICE40 Layout Viewer. [Online]*. URL: {https://github.com/knielsen/ice40_viewer} (visited on 08/10/2020).

[101] Reza M. Rad et al. "Power supply signal calibration techniques for improving detection resolution to hardware Trojans". en. In: *2008 IEEE/ACM International Conference on Computer-Aided Design*. San Jose, CA, USA: IEEE, 2008, pp. 632–639. ISBN: 978-1-4244-2819-9. DOI: 10.1109/ICCAD.2008.4681643. URL: http://ieeexplore.ieee.org/document/4681643/ (visited on 06/02/2021).

[102] D. Agrawal et al. "Trojan Detection using IC Fingerprinting". In: *2007 IEEE Symposium on Security and Privacy (SP '07)*. 2007, pp. 296–310. DOI: 10.1109/SP.2007.36.

[103] Yier Jin and Y. Makris. "Hardware Trojan detection using path delay fingerprint". In: *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*. 2008, pp. 51–57. DOI: 10.1109/HST.2008.4559049.

[104] Falk Schellenberg et al. "An inside job: Remote power analysis attacks on FPGAs". In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018, pp. 1111–1116. DOI: 10.23919/DATE.2018.8342177.

[105] Dennis R. E. Gnad, Fabian Oboril, and Mehdi B. Tahoori. "Voltage drop-based fault attacks on FPGAs using valid bitstreams". In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 2017, pp. 1–7. DOI: 10.23919/FPL.2017.8056840.

[106] Dina Mahmoud and Mirjana Stojilović. "Timing Violation Induced Faults in Multi-Tenant FPGAs". In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, pp. 1745–1750. DOI: 10.23919/DATE.2019.8715263.

[107] N. Veeranna and Benjamin Carrión Schäfer. "S3CBench: Synthesizable Security SystemC Benchmarks for High-Level Synthesis". In: *Journal of Hardware and Systems Security* 1 (2017), pp. 103–113.

[108] Muhammad Awais, Hassan Ghasemzadeh Mohammadi, and Marco Platzner. "An MCTS-based Framework for Synthesis of Approximate Circuits". In: *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. 2018, pp. 219–224. DOI: 10.1109/VLSI-SoC.2018.8645026.

[109] Linus Witschen et al. "CIRCA: Towards a modular and extensible framework for approximate circuit generation". In: *Microelectronics Reliability* 99 (2019), pp. 277–290. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2019.04.003. URL: https://www.sciencedirect.com/science/article/pii/S002627141830859X.