Technische Universität München

Lehrstuhl für Sicherheit in der Informationstechnik
an der Fakultät für Elektrotechnik und Informationstechnik

Hardware Trojans and their Security Impact on Reconfigurable System-on-Chips

Nisha Jacob Kabakci

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines **Doktor-Ingenieurs (Dr.-Ing.)** genehmigten Dissertation.

Vorsitzender der Kommission:		Prof. Dr.-Ing. Wolfgang Kellerer
Prüfer der Dissertation:		
	1.	Prof. Dr.-Ing. Georg Sigl
	2.	Prof. Dr. Sebastian Steinhorst

Die Dissertation wurde am 21.11.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 19.03.2020 angenommen.

Abstract

Manufacturers are increasingly outsourcing parts of the design and fabrication of integrated circuits (ICs) to reduce costs. Recent publications show that outsourcing of parts of the design and development process can pose a serious threat to governments and corporations, as they lose control of the development process. This facilitates the insertion of covert and undocumented functionality, also known as hardware trojans, in hardware designs. This combined with the fact that more and more devices are connected over the network would mean, a security breach in one device could not only have direct consequences but also open the door to attacks on the entire network. Thus there is growing need to protect ICs and the devices into which they are integrated from adversaries trying to maliciously modify the device to gain access to sensitive information. This work begins with an analysis of the complete development process of ICs to identify the key vulnerabilities at each stage of the IC development process. A comprehensive analysis of hardware trojan insertion possibilities and respective detection strategies is presented. The analysis also includes the estimated cost of tools and effort associated with the various hardware trojan insertion and detection schemes. The aim of this analysis is to provide a solid basis for assessing the risks and to identify the challenges associated with securing the IC development process.

Having identified the threats and challenges of malicious hardware inclusions, we show how such malicious hardware cores can be used to compromise system-on-chips (SoCs) with an integrated FPGA fabric (FPGA SoCs). FPGA SoCs are an ideal choice for applications demanding significant processing power and high flexibility, such as advanced statistical analysis of vast troves of sensor data through deep neural networks. They allow to enhance complex software systems with reconfigurable hardware accelerators that are updatable over the lifetime of the device. However, with this flexibility comes the threat of a malicious actor trying to extract and exploit intellectual property (IP). For FPGA SoCs new threats arise from the fact that the processor and FPGA are connected to the same memory bus, so that FPGA hardware designs can interfere with software routines on FPGA SoCs. An enabling factor is that integrated hardware designs are likely bought from external partners to reduce costs. There is a realistic lack of security review at the system integrators. Here, we discuss software protection mechanisms offered in conventional SoCs and techniques to circumvent them via malicious hardware blocks. As concrete examples, these vulnerabilities are exploited to demonstrate their effects on key security mechanisms, namely secure boot and secure update. From these examples we see that even when state-of-the-art software security mechanisms are implemented, this combination of high performance application processors with reconfigurable hardware creates new security threats. Hence we also present an effective mitigation technique to isolate hardware IP cores with access to memory such that they do not corrupt the rest of the system. We propose a countermeasure in the form of easy-to-review and re-usable security wrapper for hardware modules which prevents any unauthorized memory access by included hardware designs. Since embedded devices are often used in hostile environments such as autonomous cars, readout is often realistic and the user IP contained is susceptible to reverse engineering. Thus there is a need to protect the enclosed user IP from malicious modifications. Typically users have to depend on the manufacturer-provided built-in features such as bitstream/firmware encryption together with the corresponding manufacturer-provided key storage. Research has shown these manufacturer-provided built-in cores to be insecure and susceptible to side channel attacks. Hence, here we step away from manufacturer-provided decryption cores and key storage and instead use dynamic partial reconfiguration together with updatable user trusted cryptographic cores to secure user hardware cores.

Kurzfassung

Hersteller lagern zunehmend Teile des Designs und der Herstellung von integrierten Schaltungen (ICs) aus, um Kosten zu senken. Aktuelle Veröffentlichungen zeigen, dass Outsourcing hier zum Problem werden kann, da Unternehmen den Entwicklungsprozess nicht mehr vollständig kontrollieren können. Dies erleichtert Angreifern das Einfügen von Fehlern oder sogar versteckter und undokumentierter Funktionalität in Hardware Designs. Solche bösartigen Änderungen am Hardware-Design werden oft als Hardware-Trojaner bezeichnet. Dies in Verbindung mit der Tatsache, dass immer mehr Geräte über das Netzwerk verbunden sind, würde bedeuten, dass ein Sicherheitsverstoß nicht nur einzelnen Geräten schaden könnte, sondern auch die Tür für Angriffe auf das gesamte Netzwerk öffnen kann. Daher wächst die Notwendigkeit, ICs und die Geräte, in die sie integriert sind, vor Gegnern zu schützen, die versuchen, das Gerät böswillig zu verändern, um Zugang zu sensiblen Informationen zu erhalten. Diese Arbeit beginnt mit einer Analyse des gesamten Entwicklungsprozesses von ICs, um die wichtigsten Schwachstellen in jeder Phase der IC-Entwicklung zu identifizieren. Die Analyse beinhaltet auch die geschätzten Kosten für Tools und den Aufwand im Zusammenhang mit den verschiedenen Techniken zum Einfügen und Erkennen von Hardware-Trojanern. Ziel dieser Analyse ist es, eine solide Grundlage für die Bewertung der Risiken zu schaffen und die Herausforderungen zu identifizieren, die mit der Sicherung des IC-Entwicklungsprozesses verbunden sind.

Nachdem wir die Bedrohungen und Herausforderungen durch bösartige Hardwareeinschlüsse identifiziert haben, zeigen wir, wie bösartige Hardwarekerne verwendet werden können, um System-on-Chips (SoCs) mit einem integrierten FPGA (FPGA SoCs) zu kompromittieren. FPGA SoCs sind eine ideale Wahl für Anwendungen, die eine hohe Rechenleistung und Flexibilität erfordern, z.B. die statistische Analyse von riesigen Mengen an Sensordaten durch neuronale Netze. Diese Plattform ermöglicht es, komplexe Software auf dem Prozessor und Hardwarebeschleuniger auf dem FPGA auszuführen, mit dem zusätzlichen Vorteil, dass sie über die Lebensdauer des Geräts aktualisiert werden können. Jedoch erlaubt das erhöhte Maß an Flexibilität einem Angreifer, geistiges Eigentum, wie Beschleuniger zur Signalverarbeitung, zu extrahieren und auszunutzen. Für FPGA SoCs ergeben sich neue Bedrohungen daraus, dass Prozessor und FPGA an den selben Speicherbus angeschlossen sind, so dass FPGA-Hardware-Designs die Software-Routinen auf FPGA SoCs stören können. Ein Faktor ist, dass integrierte Hardware-Module wahrscheinlich von externen Partnern gekauft werden, um die Kosten zu reduzieren. In der Praxis gibt es oft einen realistischen Mangel an Sicherheitsüberprüfung bei der Systemintegration. Hier diskutieren wir Softwareschutzmechanismen, die in herkömmlichen SoCs angeboten werden, und Techniken, um sie durch bösartige Hardwareblöcke zu umgehen. Als konkrete Beispiele zeigen wir, wie diese Schwachstellen ausgenutzt werden können, um die wichtigen Sicherheitsmechanismen, Secure Boot und Secure Update, zu umgehen. Aus diesen Beispielen ersehen wir, dass selbst wenn modernste Software-Sicherheitsmechanismen implementiert werden, diese Kombination aus Anwendungsprozessoren mit rekonfigurierbarer Hardware leider zu neuen Sicherheitsbedrohungen führt. Daher stellen wir auch eine effektive Maßnahme vor, um Hardware-IP-Kerne mit Zugriff auf den Speicher zu isolieren, dass sie den Rest des Systems nicht beschädigen. Wir haben eine Gegenmaßnahme in Form eines einfach zu überprüfenden und wiederverwendbaren Wrapper-Moduls entwickelt, das den Zugriff von nicht autorisierten Benutzern durch die enthaltenen Hardware-Designs verhindert. Da eingebettete Geräte häufig in feindlichen Umgebungen, z.B. autonomen Autos, eingesetzt werden, ist ein Auslesen oft realistisch und enthaltene IP anfällig für Reverse-Engineering. Daher ist es außerdem notwendig, die integrierte Benutzer IP vor bösartigen Änderungen zu schützen. In der Regel muss der Benutzer die vom Hersteller bereitgestellten Funktionen, wie Bitstream- oder Firmware-Verschlüsselung, mit dem entsprechenden vom Hersteller bereitgestellten Schlüsselspeicher verwenden. Aktuelle Veröffentlichungen zeigen, dass diese vom Hersteller bereitgestellten Beschleuniger teilweise unsicher und anfällig für Seitenkanalangriffe sein können. Daher entfernen wir uns hier von den vom Hersteller bereitgestellten Entschlüsselungsbeschleuniger und Schlüsselspeichern und verwenden stattdessen dynamische partielle Rekonfiguration zusammen mit einem aktualisierbaren, vom Benutzer selbst entwickelten vertrauenswürdigen kryptografischen Beschleuniger, um die Benutzer Hardware-Module zu sichern.

Acknowledgement

I would like to express my sincere and heartfelt gratitude to my supervisor Prof. Dr.-Ing. Georg Sigl for having introduced me to the topic and for all the valuable feedback he has provided me over the last years. I would like to thank my mentor Dr.-Ing. Johann Heyszl for supporting my thesis at Fraunhofer AISEC.

I would also like to thank all of my colleagues at Fraunhofer AISEC, especially my colleagues from the Hardware Security Department and (former) Embedded Security Department. I am thankful to all the colleagues at the Chair of Security in Information Technology at TUM Department of Electrical and Computer Engineering for many fruitful discussions.

I would specially like to thank the following colleagues for support and guidance during the course of my PhD: Dr. -Ing. Fabrizio DeSantis, Dr. rer. nat. Julian Horsch, Dr. -Ing. Philipp Koppermann, Prof Dr.-Ing. Dominik Merli, Bodo Selmke, Dieter Schuster, Florian Unterstein, Dr. rer. nat. Michael Veltens and Philipp Zieris.

Finally, I would like to thank my family and husband for their unconditional support. I would not have be able to finish this dissertation without their help and support. I would also like to given a special shout out to Niki, Ria, Ryan, Adi and Abhi for always lifting up my spirits even during the hardest times.

Abbreviations

AES	Advanced encryption standard
AMBA	Advanced microcontroller bus architecture
ASIC	Application specific integrated circuit
ATPG	Automatic test pattern generation
BBRAM	Battery-backed random access memory
BRAM	Block random access memory
CPU	Central processing unit
CRT	Chinese remainder theorem
DDR	Double data rate
DMA	Direct memory access
DoS	Denial of service
DSP	Digital signal processing
DUT	Device under test
EDA	Electronic design automation
FIFO	First in, first out
FPGA	Field programmable gate array
FSBL	First stage bootloader
GCM	Galois counter mode
GMAC	Galois message authentication code
HDL	Hardware description language
HMAC	Hashed message authentication code
IC	Integrated circuit
ICAP	Internal configuration access port

IOMMU	Input output memory management unit
IoT	Internet of things
IP	Intellectual property
IV	Initialization vector
JTAG	Joint test action group
LR-AES	Leakage resilient advanced encryption standard
LR-PRF	Leakage resilient pseudo random function
LUT	Lookup table
MMU	Memory management unit
NVM	Non-volatile memory
OFB	Output feedback mode
OS	Operating system
P&R	Placement and routing
PR	Partial reconfiguration
PRNG	Pseudo random number generator
PUF	Physical unclonable function
RAM	Random access memory
RNG	Random number generator
RO	Ring oscillator
RTL	Register transfer level
SCA	Side channel analysis
SDRAM	Synchronous dynamic random access memory
SMMU	System memory management unit
SoC	System-on-chip
SRAM	Static random access memory
TEE	Trusted execution environment
TLB	Translation lookaside buffer
TLS	Transport layer security

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Electronic devices control all aspects of our daily life helping us create and collaborate with one another. With our reliance on electronic devices steadily growing, the risk of malicious activities, e.g. cloning and illegal use of credit cards, counterfeiting of hardware components, hijacking cars, are also on the rise. The rise in the number of internet of things (IoT) and cyber-physical systems is also causing a shift from traditionally isolated devices as previously seen in industrial and home automation applications, towards fully connected devices. Apart from the harm done to a single isolated device, there is also now the possibility that malicious activities can be scaled such that the entire network is affected. For instance, researchers have shown how one compromised smart lightbulb can be used to affect other smart lightbulbs in the vicinity [ROSW16] thereby potentially causing a city-wide blackout. Thus as the scope of connected devices expand, the need for solid information security is inevitable.

Information security aims to prevent data from being illegally accessed, modified or used. The main objectives of information security are, *confidentiality, integrity, authentication and non-repudiation.* Confidentiality protects information from unauthorized entities through mathematical algorithms that render data unintelligible. Integrity assures that the information has not been altered illegitimately. Authentication assures the information is being exchanged between authorized entities instead of masqueraders. Non-repudiation prevent entities from denying a previous commitment or action.

This thesis deals with the impact of malicious hardware inclusions in reconfigurable system-on-chips (SoCs), as they are increasingly being used in safety and security critical applications, together with corresponding countermeasures to protect the system. The rest of the chapter is organized as follows: Section 1.1 provides the motivation for this thesis. Section 1.2 and Section 1.3 states the objectives and contributions of this thesis respectively.

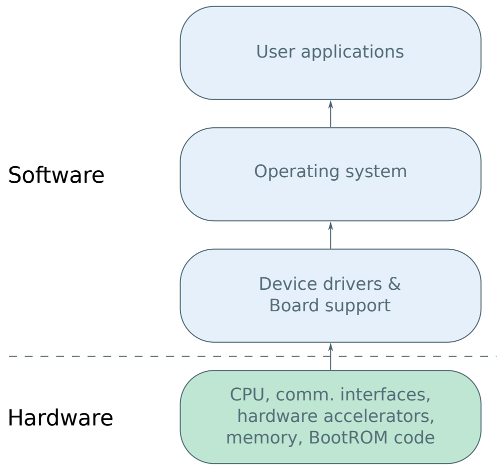


Figure 1.1: Abstraction layers of a typical embedded system

1.1 Motivation

Across all domains, embedded systems are mostly built on high-volume SoCs. SoCs are integrated circuits (ICs) that consist of a mix of both analog and digital components on a single chip capable of executing a complex software system including a rich operating system (OS). Figure 1.1 shows the different abstraction layers in such systems. For the hardware level, the common practice among embedded system designers is to use commercially available off-the-shelf SoC chips. The designer then customizes the software that runs on the hardware to meet the requirements of the end user application.

As SoC and application specific integrated circuit (ASIC) designs in general are steadily becoming more complex, companies are not always able to perform all the design in house or afford the fabrication equipment for small feature sizes. Globalization has made it easier to design and fabricate ICs by outsourcing parts of the design and fabrication to other companies around the world. In general, SoCs typically run a mixture of open-source and closed-source intellectual property (IP) and include network communication interfaces. Even large ASIC SoCs for short-lived consumer-grade routers/modems are designed with outsourced hardware modules, which means that the following threat also applies to ASICs in such cases. A factor for the outsourcing is the cost associated with the design, development and fabrication of ICs. Outsourcing allow designers to save costs and time but they also start to lose control over their design processes. This opens the issue of trust in ICs and devices into which they are integrated as adversaries can maliciously manipulate the design of ICs, without the knowledge of the designer.

Malicious manipulations of ICs have been gaining a lot of attention in the last few years by both, researchers and governmental organizations. Various publications and projects undertaken by governments and defense organizations [Ade08, For05, Mar11] are indicators of the growing fear of malicious hardware. Based on recent trends [JJC13, BS13, GN13, PLS13], we see that it is not only the outsourcing of fabrication that is prone to malicious manipulations, but the entire IC development chain. The different types of malicious manipulations in an IC include those that weaken the system, give unauthorized access to the system, and directly modify the functionality of the system. Weakening refers to manipulations that compromise the system's performance or reliability without directly manipulating the IC design or endangering the data integrity, e.g. causing early aging in ICs by using higher temperature during the IC fabrication [SWR+10]. Unauthorized accesses can be used for debugging purposes [SW12a], but can also be fatal and maliciously used to insert, modify or leak information of the IC design. Modifying functionality through the malicious insertion, deletion or modification of the original IC design is commonly referred to as *hardware trojans.* They are built to covertly gain access to secure devices and their data without affecting its normal operation. Such manipulations pose a threat not only to security-sensitive devices used for military or financial systems, but also consumer appliances.

A trojan is composed of two parts: *trigger* and *payload.* The trigger is the activation signal, while the payload is the main trojan functionality. Trojans are known to be dormant until they are triggered. Some of the typical characteristics of trojans are (i) not changing the physical form and number of inputs and outputs of the original IC, (ii) being at least 3-4 orders of magnitude smaller than the original circuit, (iii) remaining undetected during regular test phases, and (iv) operating covertly during the normal IC operation [ABK+07].

The advantage of hardware trojans is that access to a whole series or a batch of chips can be gained by manipulating the design or fabrication of an IC. The manipulations inserted during design or fabrication can be exploited later when the device is deployed with lower efforts. However, the cost of inserting malicious functionality can vary significantly depending on the design stage at which it is included.

With the rapid development in IoT applications, the need for secure, flexible and performance-driven embedded systems is growing. Vast amounts of sensor data need to be processed instantly within embedded systems for extraction of relevant information and fast reaction times. Devices are moving away from centralised cloud-based data computation and towards edge computing nodes. For instance autonomous driving requires real time object recognition with low latency to detect upcoming obstacles. A failure could have direct and fatal consequences. Thus, there is a need for high performance computing nodes directly on-site.

FPGA SoCs are considered to be valuable platforms in this regard. FPGA SoCs integrate a hard-core processor system and a field programmable gate array (FPGA) fabric on the same die. The hard-core processor can be used to run complex software, while the reconfigurable logic can be used to replace slow software functions, e.g. cryptographic functions with efficient and high performance custom hardware accelerators. Systems built using such devices (e.g. public transportation infrastructure [Lei15]) are able to support significant computational capabilities through hardware acceleration while both, software as well as hardware can be updated in the field. Thus, giving system architects more control over the design and performance. Designers are more flexible and can directly add new hardware accelerators, which they can download from an IP store. This is simply cheaper and provides a faster time-to-market. Open-source software is used for similar reasons. For example, the elastic compute cloud (EC2) from Amazon web services includes an instance with integrated FPGAs (EC2 F1 instance) where hardware cores can be used from a dedicated IP market place [Jef16]. As FPGA SoCs get more versatile, they will likely become a popular choice for embedded systems in the future. A trend that is also evident from the FPGA vendors who are continually investing in new FPGA SoCs to make them more flexible and powerful [Alt15, Xil17, Mic18b, Int19, Xil19].

Configurable hardware within FPGA SoCs is typically connected to high-bandwidth memory buses of the central processing unit (CPU) which means that hardware blocks may possibly access memory regions that are access-managed by the software system. This has severe consequences for the system, because these hardware blocks may easily corrupt the security of the entire system. We have in the past seen similar attack vectors in the PC world which were successful. For instance, external high-speed interfaces have been misused to directly access internal memory [Gam11, Rus13]. Fortunately, countermeasures like input output memory management unit (IOMMU) have also been developed against such attacks [AMD11]. IOMMU handles the memory management of peripherals with direct access to memory and thereby prevents unauthorized memory accesses. However, such threats are now possible from a new direction, i.e. through the integrated hardware.

If hardware is sourced from elsewhere, it is hence questionable whether the embedded system designers/integrators will be able to properly review the hardware code to check for unwanted functionality or vulernabilites. In many cases, hardware blocks are even delivered as a synthesized netlist which more or less restricts a proper review. The problem is that malicious functionality could be part of such hardware modules. Extensive tests must be conducted to ensure the integrity of IP cores [BH10]. However, thorough testing of IP cores is a time consuming and complex process. Typically, the IP vendors provide a testbench along with the IP core which can be used to test the core. System designers integrating the IP core would use the provided testbenches to verify the functionality of the IP core. These testbenches do not cover all the possible test cases. As a result, finding a small stealthy malicious function by solely running these testbenches is extremely difficult.

1.2 Objectives

The objectives of this work can be summarized into three main categories:

1. Comprehensive understanding of the current threats of malicious hardware modules to embedded systems
2. Evaluate the practicality and scope of malicious hardware on the overall system security
3. Develop and evaluate countermeasures to protect systems against malicious hardware modules

1.3 Contributions

This work begins with a comprehensive study of the threat of hardware trojans and different detection schemes currently known. The feasibility of malicious insertions and the associated costs are derived for each stage of the IC development and production. The different entities capable of inserting malicious functionality are also analyzed. This analysis allows us to judge the threat level that is essential for future researchers and design houses to develop efficient techniques to protect the integrity of their products. Based on the outcomes of the vulnerability analysis different threat vectors are derived. This resulted in a contribution, *Hardware Trojans: current challenges and approaches* to the *IET Computers & Digital Techniques* journal in 2014 [JMHS14].

Having identified the vulnerabilities in the IC development chain, they are exploited to demonstrate their implications using real world examples. In that, hidden functionality in third party hardware cores is derived to be a very important threat vector. Hidden functionality in a third party hardware core is used to circumvent memory protection and corrupt the software update process running on the processing system. For demonstration purposes, a malicious hardware core is designed for the Xilinx Zynq-7000 FPGA SoC platform which is capable of finding and replacing the public key in memory which is used for the authentication of system updates. The core scans the memory at run-time for this public key. Once the key has been located, it is replaced by a pre-defined key known to the attacker. This allows an attacker to later maliciously update and alter the system. Thus, a malicious hardware core is used to access unauthorized regions of memory and alter its contents. This resulted in a contribution, *Compromising FPGA SoCs using malicious hardware blocks* at the *Design, Automation & Test in Europe Conference & Exhibition, DATE* conference in 2017 [JRZ+17].

Secure boot, one of the most important and basic security features of embedded systems, is analyzed to understand how malicious hardware blocks in FPGA SoCs can compromise it. A proof of concept is implemented on the Xilinx Zynq-7000 FPGA SoC device. We also show why newer embedded systems which include IOMMUs will likely be susceptible to such attacks. The proof of concept system includes a conventional software stack along with an additional hardware block for the FPGA. The included unwanted hardware functionality overwrites parameters of the second stage boot loader, U-boot, during secure boot so that an unauthorized kernel image is retrieved and booted over the network instead of the local authorized image even though all previous boot stages are properly verified prior to that. This resulted in a contribution, *How to Break Secure Boot on FPGA SoCs through Malicious Hardware* at the *Cryptographic Hardware and Embedded Systems - CHES* conference in 2017 [JHZ+17].

Since FPGA SoCs may contain a mix of hardware cores from various sources whose trust level may be hard to determine, there is need to prevent hardware cores from interfering or corrupting another process on the system. Here, a security wrapper for hardware cores with access to memory is developed. It is a simple hardware wrapper for the AXI bus interface which is easy to wrap around all outsourced hardware cores with memory access and is configured from software. This wrapper can be easily integrated by the system designers to protect the system against all unauthorized memory accesses through hardware cores. It can be seen as a stripped-down IOMMU which instead works directly after configuration (without requiring the software to explicitly enable or configure it, which is usually done after boot in the OS) and has a smaller set of functionality, thus, trusted code base. This contribution was part of the paper, *How to Break Secure Boot on FPGA SoCs through Malicious Hardware* presented at *Cryptographic Hardware and Embedded Systems - CHES* conference in 2017 [JHZ+17].

As such embedded systems contain sensitive user IPs and are deployed in hostile environments, thus their configurations need to be protected. We develop a security enhanced configuration process using alternative side-channel hardened cryptographic accelerators for the authentication and decryption of hardware configurations instead of the manufacturer-provided built-in cores. Dynamic partial reconfiguration of FPGAs is used together with custom user cryptographic cores to securely configure a system. A central contribution of this work is *a solution to retrofit FPGA SoCs* with side-channel hardened cryptographic cores to enhance design security. Further, by using FPGA SoCs we also have the added benefit that the hardened cryptographic cores are updatable, which enables them to be updated to fit new standards or fix bugs. Parts of this work resulted in the following contributions, *Securing FPGA SoC configurations independent of their manufacturers* to the *International System-on-Chip Conference, SOCC* in 2017 [JWH+17] and *SCA Secure and Updatable Crypto Engines for FPGA SoC Bitstream Decryption. At the Workshop on Attacks and Solutions in Hardware Security, ASHES* in 2019 [UJH+19].

1.4 Organization

The rest of the work is organized as follows:
The thesis begins with an introduction to FPGAs and FPGA SoCs in Chapter 2. Along with this, the relevant background on the conventional security mechanisms in SoCs is also provided in Chapter 2. We then go on to understand the current threats and challenges with regard to hardware trojans during IC development and FPGA-based development in general. Chapter 3 provides a study on the current threats and challenges in hardware trojans design and detection techniques. The next chapters exploit the threats identified from this analysis and evaluate their feasibility in real world scenarios. A practical realisation of the attack targeting secure software update through the hardware on FPGA SoCs is outlined and implemented in Chapter 4. Chapter 5 describes the implementation of the attack on secure boot using a malicious hardware core in the FPGA fabric of an FPGA SoC. Following the attacks, two countermeasures are presented. Chapter 6 provides a description of the design and implementation of the security enhanced AXI-wrapper in order to isolate malicious cores with access to memory. Chapter 7 presents the scheme for secure configuration of FPGA SoCs using user cryptographic cores instead of manufacturer-provided closed-source cryptographic cores. Finally Chapter 8 draws the conclusion of this work.

Chapter 2

Background

This chapter provides an introduction to the security in conventional SoC-based embedded systems. In particular, the security at different abstraction levels of the system will be presented with a focus on the memory management schemes. Also provided in this chapter is an introduction to FPGA systems. Finally, as a basis for the attacks to be discussed in later chapters, the new contemporary embedded platform FPGA SoC is also introduced in this chapter.

The chapter is organized as follows: Section 2.1 provides an overview of security in embedded systems. Section 2.2 describes different memory isolation schemes available in today's SoCs. Section 2.3 and Section 2.4 provide an introduction to FPGAs and FPGA SoCs respectively.

2.1 Security of Embedded Systems

Popular security mechanisms for embedded systems are described in this section. This is necessary to understand how embedded systems have been evolving over the years and how the different security issues have been addressed in traditional embedded systems. The security mechanisms can be divided into three layers, hardware, system start-up and run-time security, built on top of each other as addressed below.

2.1.1 Hardware Security

Hardware security is the lowest level of embedded system security and is the critical building block for securing later stages. With the rapid increase of the IoT infrastructure and connected devices, embedded systems are now within the physical reach of potential adversaries. A read-out of the contained IP is often possible thereby making them also vulnerable to attacks like reverse engineering or malicious modification. To counteract this, many SoCs for embedded systems include hardware cryptographic accelerators with a dedicated secure memory for keys and/or certificates [Xil14]. These accelerators are typically used to protect the user IPs while also accelerating the boot process. Manufacturers now also allow users to freely use these hardware accelerators for their respective use-cases [Xil17]. The secure key storage helps to refrain from storing such information in external memories.

Manufacturer-provided hard-core cryptographic accelerators, however, have in the past shown to be vulnerable to side channel attacks [MBKP11, SW12a, SW12b, SMOP15, MS16] where physical properties of the device such as timing [Koc96], power [KJJ99] and electromagnetic radiation [QS01] are observed and exploited. The physical properties are usually observed when the device performs critical functions like cryptographic operations or checks, in order to retrieve the secret key or other sensitive information from the device. As an effective countermeasure against such attacks, newer devices include dedicated side-channel protection for their cryptographic accelerators [Mic15, Xil18b].

In addition to hardened hardware cryptographic accelerators for design security, hardware security mechanisms also include protection against physical tampering. Invasive attacks like inserting transient faults using a laser during a cryptographic operation can be used to alter the outputs or probing attacks [Noh12] can be used to read out the contents of a chip. Tamper detection schemes are effective countermeasures against invasive attacks which remove the device casing or the package of the chip to extract the underlying contents. Devices can be protected against physical tampering by using tamper-proof casings with light sensors, wire meshes and reactive membranes to detect break-ins. Chip-internal tamper detection sensors typically monitor clock and voltage inputs to prevent fault-injection attacks [BCN$^+$06, OSS17]. In case of a tamper event, critical data is cleared and the system is shut down.

Protection of debug interfaces is another important aspect of hardware security, especially in case of physically accessible devices, so that read-out and/or corruption of data and software via debug interfaces is prevented in the field [GBMB16]. This can be achieved by using one-time programmable fuses to permanently break the link to the debug interface. However often manufacturers prefer to have access to the device via the debug interface to facilitate failure analysis. In such cases, password protected interfaces can be used such that only authorized entities have access to the device via the debug interface.

2.1.2 Secure System Startup

SoCs and other embedded platforms like FPGAs and FPGA SoCs usually have very little to no on-chip persistent memory where the binaries can be stored. Typically, the binaries are stored in plain in an external non-volatile memory (NVM) and transferred to the device at power-up over an insecure channel. This transfer can be intercepted by attackers with access to the device, like in a man-in-the-middle attack, to read out the contents of the NVM or alternatively the NVM chip could be desoldered and with minimal effort its entire contents can be read out.

Formerly, vendors of FPGA devices have relied on proprietary configuration bitstream[1] formats which are intended to make reverse engineering of the underlying code difficult or even impossible. However, this has not really been the case, researchers have been able to successfully reverse engineer these proprietary bitstream formats [NR08, BSH12, DWZZ13, TZ13, Skr13, Sym19], thereby revealing the underlying code. Furthermore, these proprietary formats do not deter an adversary from simply cloning the device. For software binaries, tools like IDAPro [Hex] are readily available which are capable of decoding software binaries to reveal the underlying code. From this we learn, hardware configuration bitstreams and software binaries stored in external NVMs need to be protected using strong cryptographic techniques and securely loaded to the device when the device is powered-up.

One of the most important security mechanisms is a secure boot process. In this process, all executed code is verified for integrity and authenticity using cryptographic means before execution. This means that right after the system startup, the running software can be trusted, which is the required foundation for all later security mechanisms. For a secure boot, a chain-of-trust is established which starts from the very first code that is executed from within the CPU internal hardwired ROM (also including respective keying material), commonly called the hardware root-of-trust [Dic14].

Using similar functions as secure boot, a secure update process allows updates in-the-field from authentic sources by checking authenticity and integrity as well as decrypting confidential data using cryptographic methods. Both require a secure storage for the respective key material and optionally also secure hardware accelerators for the cryptographic functions.

[1]Configuration bitstream is a binary file containing information on how the FPGA must be programmed to realize the implemented functionality.

2.1.3 Runtime Security

After the device startup, all assets such as cryptographic keys, processed data and control functions must be protected against run-time attacks on the software. This only makes sense if the running software is trusted from the start (i.e. secure boot). Software, however, is never flawless and could contain bugs. Software exploits capitalize on weaknesses in the implementation, e.g. pointers going out of bounds or dangling pointers [SPWS13] in order to carry out attacks like code corruption and control flow hijacking. Over the last years many such bugs have been published, e.g. the heartbleed bug. The heartbleed bug [Cod14] exploits a bug in the OpenSSL library allowing a remote adversary to steal secrets or even impersonate users.

A lot of effort has been spent to develop robust countermeasures for run-time protection. Isolation of software parts on embedded systems is an important topic in IT security as it can prevent successful attacks in one part of the software from corrupting other parts. Memory isolation can be achieved through the OS or a hypervisor with the support of hardware features like memory management units (MMUs), IOMMUs, and trusted execution environments (TEEs). A description of these memory isolation techniques is provided in the subsequent section.

2.2 Memory Isolation and Partitioning in SoCs

In this section we provide a description of the key hardware mechanisms available in current SoCs for memory isolation and partitioning.

2.2.1 Memory Management Unit (MMU)

An MMU is a hardware unit that handles memory separation for the OS. The MMU is responsible for translating a virtual memory address to a physical memory addresses. Virtual memory addressing enables processes to address more memory than the physically available memory. MMUs divide the virtual memory space into several pages. Each process has its own virtual address space from which they are executed, thus giving the impression that each program has its own large contiguous memory space as depicted in Figure 2.1.

Figure 2.1 shows a simplified view of the address translation between the CPU and the physical memory using the MMU. The MMU typically consists of a translation lookaside buffer (TLB) which holds a cache of the recent

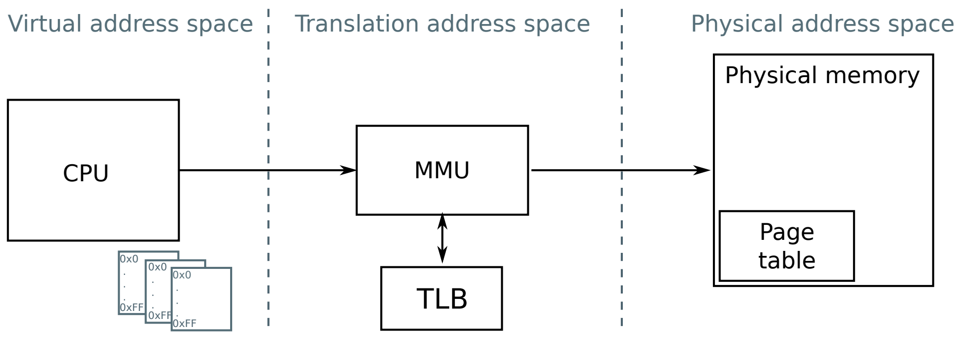


Figure 2.1: Memory management unit

translation mappings, while all the mappings between virtual pages and physical pages in the main memory are in the page tables. If an address needs to be translated, the MMU first checks the TLB if the mapping is cached else the address mapping is looked up in the page table entries.

In addition to memory translation, an MMU also assigns different attributes, such as cacheable, read-write or read-only, for each page of the memory. Memory restrictions are maintained and managed using page tables. Processes in supervisory mode (i.e., higher privileges) are allowed to make changes to the translation table. By managing and restricting the access of programs in the user levels, unauthorized software access from other user level programs are not permitted. Processes typically have access to their dedicated virtual address space and also the shared memory space. This way, the MMU isolates processes and prevents one corrupted process from accessing the memory of other processes.

MMUs, while restricting the access of processes under the CPU from unauthorized sections of the memory, do not protect against unauthorized memory access from other peripherals on the SoC with direct memory access (DMA). Other techniques would be necessary to isolate such peripherals as described in the subsequent section.

2.2.2 Input/Output Memory Management Unit (IOMMU)

SoCs typically also integrate peripherals that have direct access to memory, this allows them to continuously read/write from memory without interruption, hence they are commonly used for peripherals requiring high-throughput like high speed communication cores. These peripherals, however, are outside the scope of the MMU and thus allowing these peripherals to transfer data back and forth with the main memory without the supervision

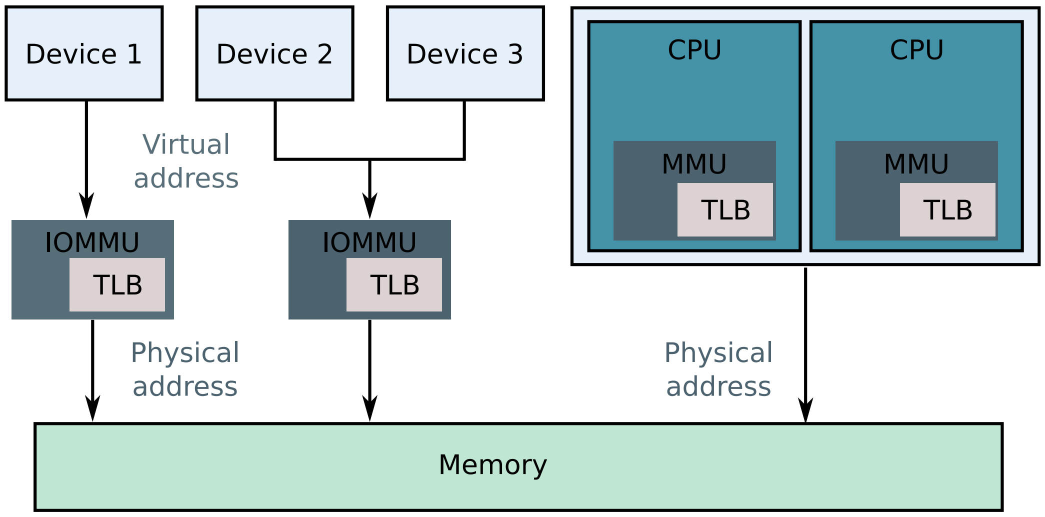


Figure 2.2: I/O memory management unit

of the OS which controls the MMU.

This, however, can be handled by another protection mechanism known as input output memory management units (IOMMUs). IOMMUs work similar to an MMU but for peripherals on the SoC that have direct access to memory. Newer SoCs come with an IOMMU [AMD11] which is called system memory management unit (SMMU) in ARM SoCs. An IOMMU performs both, memory translation as well as memory protection for the peripherals on the SoC. The IOMMU also handles the remapping of memory and I/O in the case of virtualized systems, thus allowing guest operating systems to use the underlying hardware resources.

Figure 2.2 shows how IOMMUs are used in a SoC. Traffic from the peripheral devices to the memory is managed via the IOMMU, while traffic in the opposite direction from the CPU to the peripheral device must be managed by other techniques like the MMU of the CPU. Similar to an MMU, translation tables are used by an IOMMU, to manage the access of the different peripherals to the main memory. The IOMMU looks up the TLB of the corresponding peripheral device on every access. The translation tables are maintained by the OS or (in case of virtualized system) the virtual machine monitor. IOMMUs provide no indication to the underlying peripheral about a failed or unauthorized translation [AMD11]. If multiple devices/peripherals with direct access to memory are present in a SoC, an individual IOMMU can be placed in front of each of the peripherals as shown in Figure 2.2. Typically a source identifier of the device is sent to the IOMMU with every DMA transaction. The IOMMU looks up the access rights permitted for the device

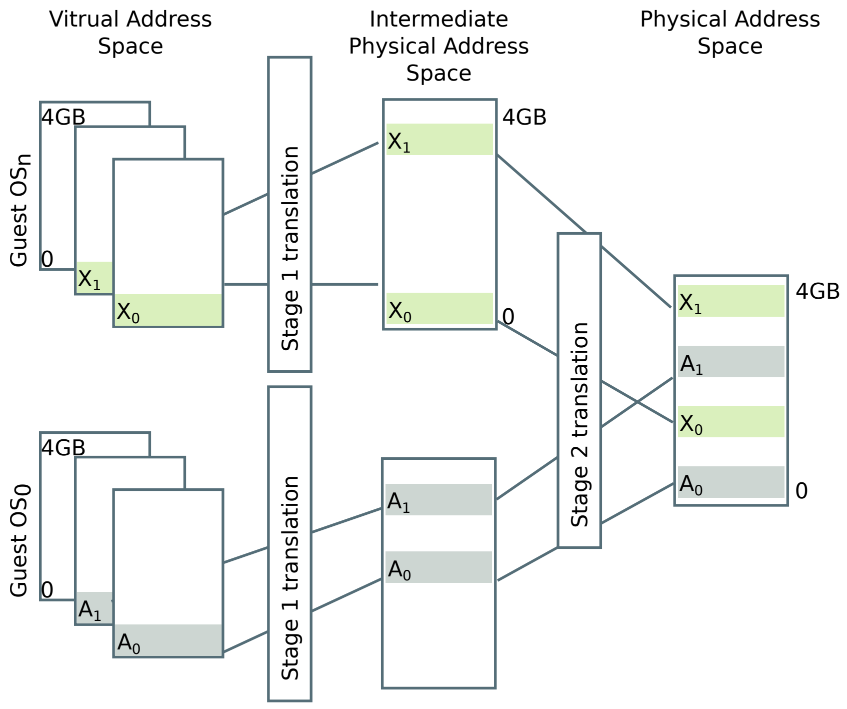


Figure 2.3: I/O memory management unit address translation

before allowing the transaction to go through.

The address translation can be split into two stages to facilitate virtualization of peripherals to the guest OS with each stage maintaining a translation table as shown in Figure 2.3. During Stage-1 the virtual address is converted to an intermediate physical address and in Stage-2 the intermediate physical address is translated to the physical address. Stage-1 is used to provide isolation i.e., DMA isolation within the OS, while Stage-2 is intended for the virtualization of the peripheral with direct memory access to the guest virtual machines. The IOMMU maintains a page table entry to translate a guest physical address to a system physical address, while the guest OS maintains a device translation table to translate a guest virtual address to a guest physical address.

Different strategies can be adopted for the usage of IOMMUs to derive a suitable balance between performance and protection using an IOMMU [WRC08]. The trade-off between performance and protection is derived based on the management of the page table entries.

- Single use mapping: The IOMMU is configured once for each I/O transaction. After each transaction the IOMMU mapping of the corresponding transaction is invalidated which in the case of a virtualized system is handled by the virtual machine monitor.

- Shared mapping: The mapping among different devices with the same and valid physical memory page is shared. This strategy checks if an IOMMU mapping already exists for the given memory page and reuses it. Here the overhead with creating, maintaining and destroying page table entries is reduced as they do not need to be updated after every access. This scheme is typically used for applications that repeat the same message. When no outstanding transactions remain, the mapping is invalidated.

- Persistent mapping: This allows mappings to stay valid even after all transactions have been completed, thereby further reducing the load with creation and destruction of IOMMU mappings compared to the shared mapping strategy.

- Direct mapping: This strategy can be used to reduce the runtime overhead of managing and updating the IOMMU mappings by reusing them. Here the physical address space can be directly mapped to the guest OS, thus there is a one-to-one mapping between the IOMMU mapping and the physical address space of the guest OS.

2.2.3 Trusted Execution Environment (TEE)

Trusted execution environment (TEE) [Ste14] is a secure and integrity protected enclave that provides tamper resistance to applications running within the enclave. Isolation of parts of the software and data can be hardware enforced through features like the ARM TrustZone [ARM08] and Intel SGX [McK15]. The security requirements of the TEE are as follows [SAB15]:

- Data of one partition is not allowed to access data of other partitions

- Shared resources cannot be used to derive information of other partitions

- Applications cannot gain access to the secure partition unless explicitly provided by the management unit

- Errors from one world cannot propagate to the other partitions

The system is divided into two words: an isolated secure world and a normal world. This can be enforced through hardware protection, e.g. ARM TrustZone uses a single bit signal (ARPROT/AWPROT) to enforce and check the privilege of the processes for each transaction while a security monitor manages the switching between the two worlds. The secure world applications run alongside the normal OS. Typically the secure world will contain very minimal applications and interfaces, like cryptographic accelerators and cryptographic keys, while everything else is placed in the normal world. A standard operating system is generally placed in the normal world. Within each world further privilege levels, like user-space applications and kernel-space applications, can be enforced similar to a regular OS.

Peripherals and memory associated with the secure world are completely isolated from the normal world. The memory and peripherals associated with the normal world do not have direct access to the secure world. Thus, even if the normal execution environment is corrupted, the sensitive information in the secure world of the TrustZone will not be compromised.

2.3 Field Programmable Gate Arrays

A field programmable gate array (FPGA) is an integrated circuit (IC) which unlike traditional ICs is customizable post fabrication and through its entire life-cycle. An FPGA fabric consists of a magnitude of individual building blocks that can be interconnected via a surrounding routing network. Together they can be used to implement any arbitrary configurable digital hardware.

FPGAs have a low non-recurring cost but higher per-unit cost. Thus historically, FPGAs have been used for fast prototyping of consumer goods and low-volume markets such as defence and aerospace. In addition to fast prototyping, FPGAs are widely used because of their reconfigurability and inherent parallelism (several hardware cores can be operated simultaneously). The high up-front costs together with stringent requirement in terms of power had made them unsuitable for high volume consumer market for a long time. This trend has, however, changed over time and FPGAs are now also increasingly being used in various industrial and consumer products [Jef16, IFI15, Lei15, Alt13]. The high computational power and possibility of in-field hardware updates make them very attractive for a wide range of applications like signal processing and machine learning.

Based on the programming technology used, FPGAs can be classified into three broad classes:

1. **SRAM-based**: The FPGA fabric is made up of static random access memory (SRAM) cells. SRAM cells are volatile static bistable memory cells used to store bits of configuration data. Contrary to dynamic

RAM (DRAM) cells which need to be periodically refreshed, SRAM cells retain the data as long as power is supplied.

This is the most widely adopted technology by commercial FPGA vendors. Examples of commercial SRAM-based FPGAs include Xilinx Virtex, Kintex and Artix series of FPGAs and Intel's (formerly Altera) Stratix, Cyclone and Aria series of FPGAs.

2. **Flash-based**: These are based on flash technology which are non-volatile by nature. Hence they do not require external power to retain the configuration data. They are made up of an array of floating gate transistors commonly referred to as flash cells.

   Flash-based FPGA products trail behind SRAM-based FPGAs in terms of density, performance and on-chip features such as processor core, high speed I/O. This is mainly attributed to the larger structures of the flash cells and hence resulting in a lower performance. Over the last years this trend has changed, flash-based FPGAs are now catching up with their SRAM-based counterparts in terms of technology sizes. Microsemi's Igloo and ProASIC series of FPGAs are examples of flash-based FPGAs.

3. **Antifuse-based**: These FPGAs are also non-volatile by nature but unlike the flash-based FPGAs, antifuse FPGAs can be programmed only once. They are made up of an array of antifuses which are one-time programmable. As the name already suggests, antifuses functionally differ from fuses. Antifuse start with a high resistance and form a permanent conductive path when the fuse is burned. In contrast fuses are initially conducting, this conductive path is permanently broken after a high current is passed through, i.e. when the fuses are burned.

   The primary advantage here is that they are non-volatile. The Axcelerator and MX series of FPGAs from Microsemi are based on the antifuse technology.

This work will focus mostly on SRAM-based FPGAs as they are the most widely used. Figure 2.4 shows the structure of an FPGA with its basic building blocks. The basic blocks of an FPGA are:

- Configuration logic blocks
- I/O blocks
- Block random access memory (BRAM)
- Special purpose blocks

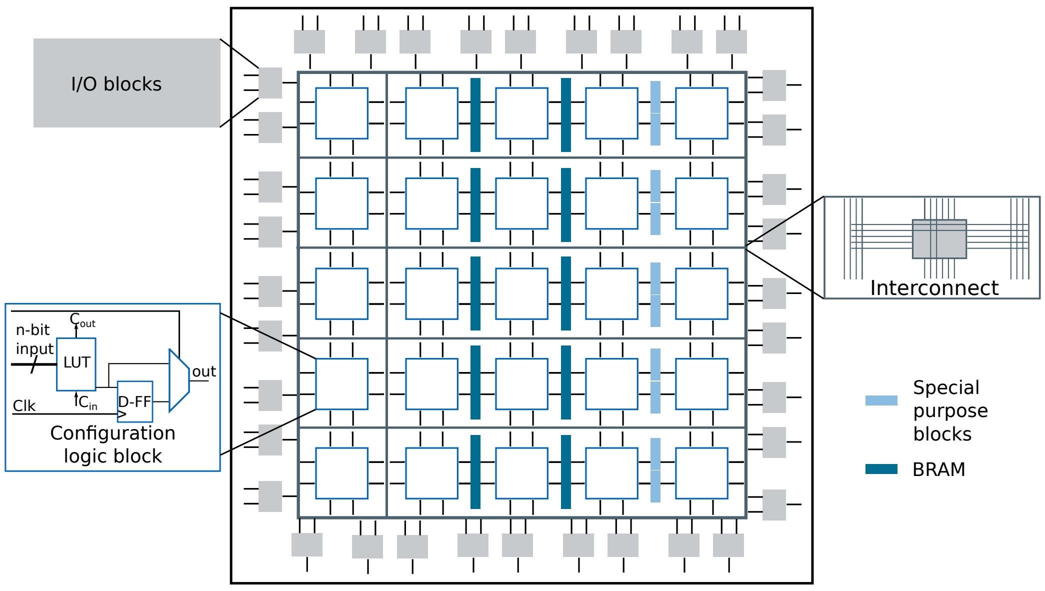
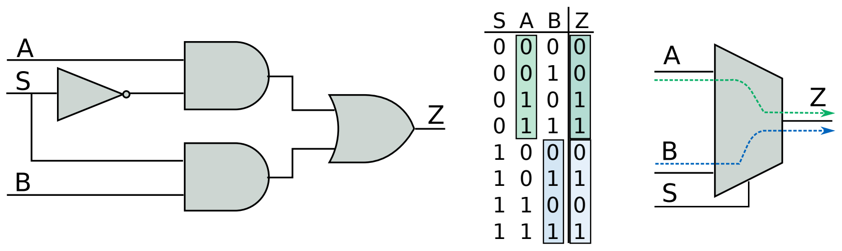


Figure 2.4: Structure of an FPGA

These basic blocks are all connected by a configurable interconnect to implement arbitrary hardware circuits and are described in more detail in the following.

2.3.1 Configuration Logic Blocks

A configuration logic block is the fundamental component of an FPGA and is used to implement logic and provide storage. The configurable blocks are usually connected in an array to implement arbitrary logic functions. One configuration logic block is made up of smaller logic cells often referred to as *slice or logic elements*. They typically consist of n-bit lookup tables (LUTs), D-flipflops (D-FF), carry logic and multiplexers as shown in Figure 2.4. Essentially, a LUT is a truth table built using SRAM bits which defines how the combinatorial logic behaves. LUTs takes an n-bit input and produces a 1-bit output. Current FPGAs offer 4 or 6-input LUTs. Several LUTs are used to implement any m-bit input boolean function. An m-bit input function is implemented, using 2^m SRAM bits and m:1 multiplexer. Figure 2.5 shows an example of a multiplexer starting from the circuit diagram, the corresponding truth table and the logic symbol. The output from the LUT can either be registered or unregistered. The multiplexer decides if the output for the logic block is the output from the flip-flop or the output from the LUT as depicted in Figure 2.4. The carry chain (C_{in},C_{out}) is used to implement efficient arithmetic operations like adders and counters since it allows logic elements



S	A	B	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Figure 2.5: Example of a multiplexer

to communicate with their neighbours more efficiently when compared of the slower global routing network.

2.3.2 Routing

As FPGAs can be used to implement any arbitrary circuit, the routing network must be flexible. The programmable logic blocks are connected to each other and the I/O blocks via a programmable routing network to provide the desired functionality. Routing is the most important process, about 90% of the FPGA configuration bitstream is devoted to the interconnect [BR99]. The routing network is made up of a combination of various types of elements, e.g. short wires for local connection and long wires for distant connections.

The routing architecture of each platform defines the length of the routing wires, location of the routing switches and how wires are connected together. It also defines which wires adjacent to input/output can connect to the logic pin and the width and spacing between the routing wires. The overall performance of the FPGA is directly dependent on the arrangement and distribution of these resources. Based on their distribution there are two routing styles: *island* [Xila] and *hierarchical* [DeH99].

Island style is the most commonly used routing technique in commercial FPGAs. The routing consists of an evenly distributed horizontal and vertical routing network. Connection blocks are used to connect the logic blocks to the routing network. The output pins can connect to the routing network through a pass transistor or multiplexer. Figure 2.6 shows a simple island style routing consisting of various routing resources like long wires (wires that connect the I/O of a logic block across multiple switch boxes), short wires (wires that connect the I/O of a logic block to the adjacent switch box). The connection between the logic blocks and wires are achieved using programmable connection switches. The multi-length wires are used to balance the area and delay of the routing.

Hierarchical routing, often also referred to as tree-based architecture, di-

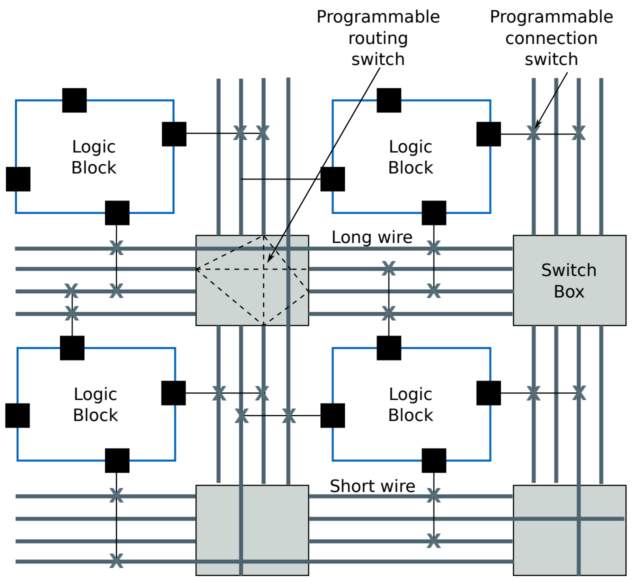


Figure 2.6: Island style routing

vides the FPGA into clusters and exploits the locality of connections. Each cluster is connected to the neighbouring clusters. Thus forming a hierarchical structure between the different logic blocks. The lowest level of connection is between logic blocks within a cluster via wire segments. The connection between different clusters requires traversing through one or more hierarchical levels. Figure 2.7 shows an example of a hierarchical routing architecture. This routing architecture has been used in a number of FPGAs from Intel, notably in the Flex 10K family [Alt03].

2.3.3 I/O Blocks

I/O blocks are used to interface the FPGA to external peripherals. They are placed on the periphery of the FPGA fabric as can be seen in Figure 2.4 and are also connected to the programmable routing interconnect. Each block consists of a group of basic elements like, bidirectional buffers, registers and multiplexers, which are connected together to provide input/output functionality. They are configurable to support various standards and features such as configuring the voltage, pull-up resistors, tri-state buffers and double data rate resisters for double data rate (DDR) synchronous dynamic random access memory (SDRAM) interfacing.

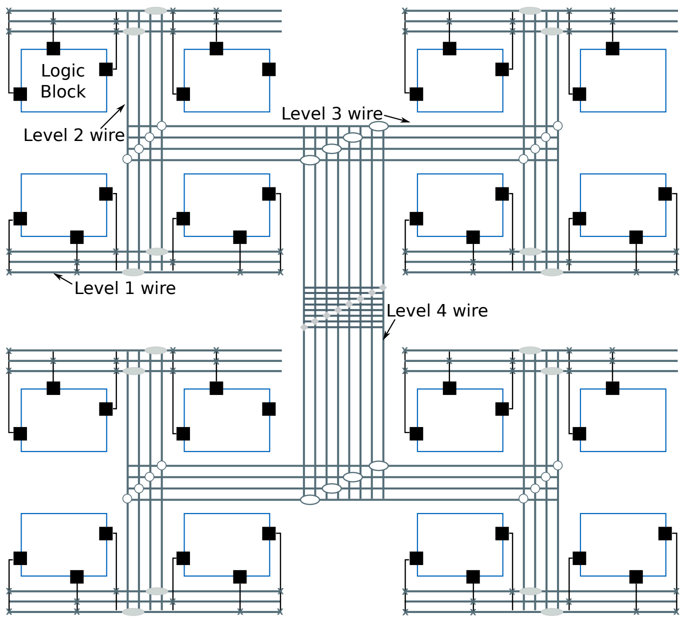


Figure 2.7: Hierarchical style routing

2.3.4 Block RAM

Block random access memories (BRAMs) are special blocks in an FPGA fabric designed to implement large memories. They are made up of several SRAM cells and are used to store data on-chip. Each BRAM can store several kilobits of data depending on the architecture of the FPGA (e.g. in the Xilinx 7 series each BRAM can store up to 36 Kb). Several of these BRAMs can be cascaded together to create a larger memory, e.g. up to 360 Mb in high-end devices [Xil16b].

BRAMs can be configured as a simple single port memory or dual port with one port to read and one to write. Additionally they can be configured as true dual port RAMs where each port can be used to both read/write and run on different clocks. Apart from functioning as a traditional memory, BRAMs can also be configured to operate as first in, first out (FIFO) buffers, read-only-memories or stacks.

2.3.5 Special Purpose Elements

In addition to the standard building blocks described above, FPGAs may also have a limited number of special purpose elements to perform dedicated functionality. The special elements include multipliers for dedicated block optimized multiplications, digital signal processing (DSP) blocks, phase locked loops, clock buffers for enabling/disabling global and regional clock domains. Another common block is the joint test action group (JTAG) controller which is used to provide in-system debugging. JTAG is also often used to load and configure the FPGA.

2.3.6 FPGA Design Flow

The entire FPGA design flow starting from the source files and until the device is deployed in the field will be described in this section. The development chain of ASICs will be described in Chapter 3. Figure 2.8 shows the design flow of FPGAs, it can be divided into three main stages, design, compile and runtime as described below:

Design Time

As with any development, the design flow begins with a specification, this determines the functionality that needs to be implemented. Here the protocols, algorithms, and interfaces necessary for implementing the desired functionality are determined.

Next, the functionality is implemented using a hardware description language (HDL) of choice, like VHDL, verilog, or system verilog. The design can then be simulated to verify the design functionality with respect to the specification. For this a testbench is developed with a variety of test vectors to check for bugs or flaws in the implementation. Following this an FPGA is selected based on the resource requirements of the design.

Compile Time

The design files from the previous step are synthesized. Typically FPGA vendors provide a dedicated electronic design automation (EDA) tool for their devices. For instance the integrated synthesis environment (ISE) is an EDA tool from Xilinx for the design, simulation and synthesis of hardware designs for older Xilinx FPGAs [Xil12]. ISE was superseded by the Vivado design suite for the newer Xilinx FPGAs and FPGA SoCs [Xil18a]. The EDA tools automate the process of mapping the instantiated functionality to the corresponding resources available on the device.

During synthesis the HDL files are translated in terms gates and flip-flops. Here the design is mapped to the various resources like slices, I/O blocks and BRAMs, available on the device. This translated circuit description is known as *netlist*. The design can once again be verified using post-synthesis simulation to verify that it still functions as specified.

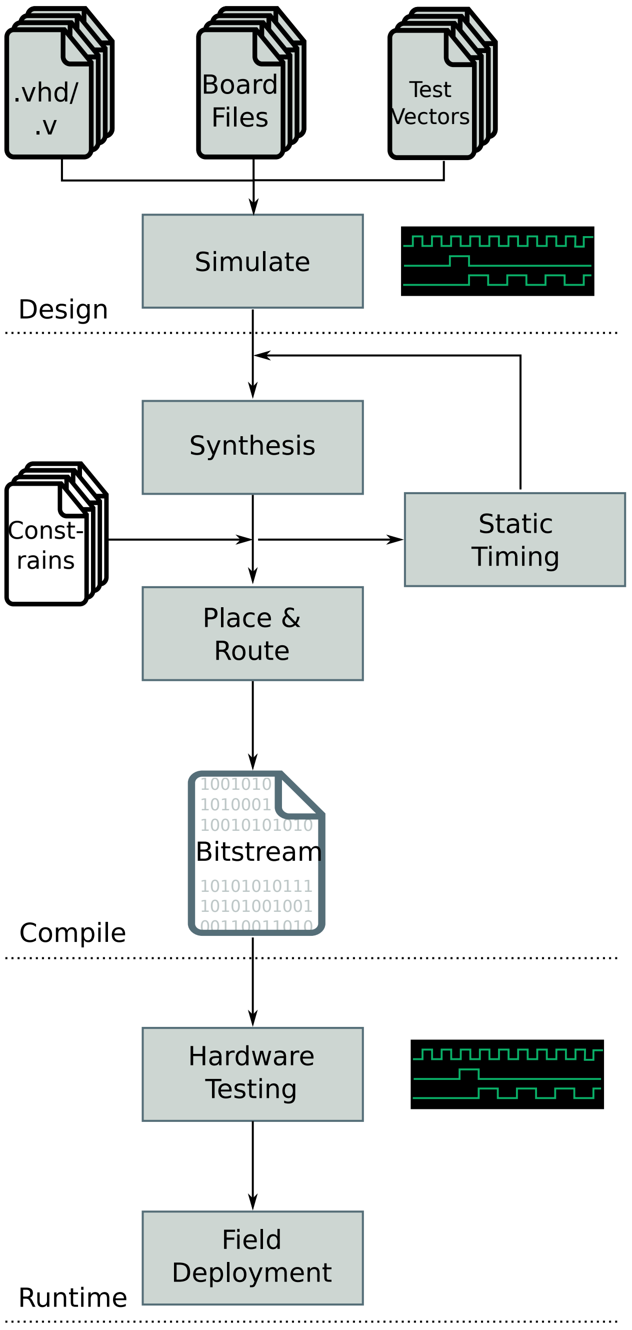


Figure 2.8: FPGA design flow

Next, the design is placed and routed. During this process the different logic blocks generated in the netlist are interconnected. The design can again be verified using a functional simulation and static timing analysis. The design can be optimized for throughput or area by providing constrains for the placement and routing of the different modules of the design.

After placement and routing, the design is converted to a *bitstream*. A bitstream is a binary sequence of bits that make up the configuration data, i.e. the implemented design, of the FPGA. This binary sequence is usually proprietary and changes from one FPGA family to the next.

Run Time

The FPGA design can now be tested on the real hardware. For this the generated bitstream is loaded to the device. Due to the volatile nature of SRAM-based FPGAs, the bitstream needs to be loaded to the FPGA at each power-up. The bitstream is transferred from the external NVM on each power-up. In contrast, for flash-based FPGAs, this process is done only once for each bitstream. The bitstream can be loaded through a standard configuration or debug interface such as JTAG from a host PC. Alternatively, the bitstream can even be loaded from an external NVM like, an SD card or even remotely over the network.

Once the FPGA is configured with the corresponding bitstream, the design can be tested by running built-in self tests or by monitoring activity on the external ports of the FPGA, using a logic analyzer or an oscilloscope. Additional debug logic can also be included in the design during the synthesis to monitor selected nets in the design. In system debug cores such *Chipscope* and *Integrated Logic Analyser (ILA)* from Xilinx [Xil16a] or SignalTap from Intel [Alt11] can be used to monitor the selected nets in the design. *Care must however be taken to not include these debug cores in the final system design as they could be exploited by an adversary to gain information on the system design.*

After the design has been thoroughly tested and debugged, the FPGA can be deployed in the field. It can then be updated either fully or partially during its lifetime to meet new application needs.

2.4 Introduction to FPGA SoCs

Powerful System-on-Chips (SoCs) which include configurable hardware in the form of an FPGA on the same chip are increasingly popular platforms for embedded systems because they offer high performance and high flexibility in software as well as hardware. The tight coupling between the CPU and FPGA provides a high performance at lower power and costs when compared to using the two independently. As embedded systems are often deployed in the field for a long lifetime, they require updates from a functional as well as from a security perspective. This has been simplified with FPGA SoCs, as here both the hardware and software can be updated over the lifetime of the device. Further, the updatability option also allow designers to adapt and update their design to keep up with changing standards without having to roll out new hardware. All these advantages are persuading designers from various fields like industrial automation, automotive and aerospace to switch to FPGA SoCs.

Figure 2.9 depicts a general overview of FPGA SoCs. They consist of two main building blocks:

- Hard core application processor[2] with dedicated peripherals and a
- FPGA fabric for integrating custom hardware.

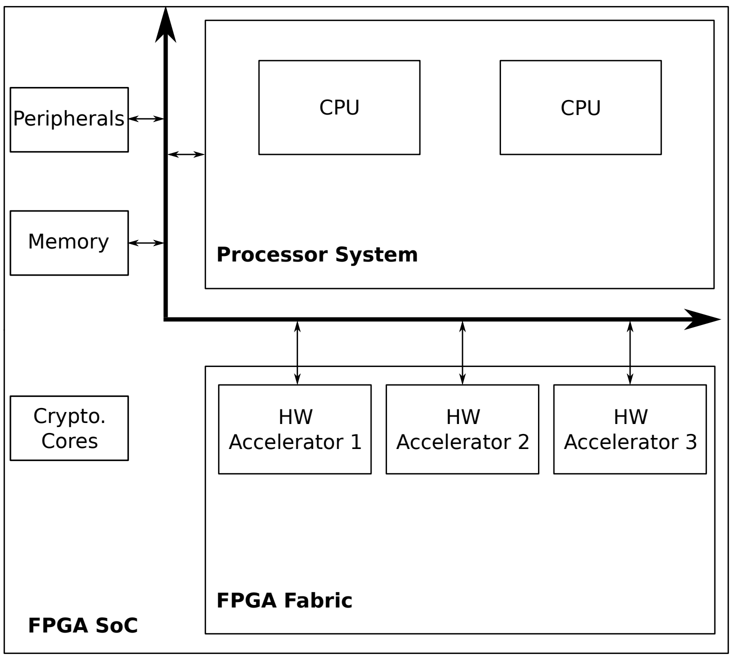


Figure 2.9: General architecture of FPGA SoCs

In addition to these two building blocks, FPGA SoCs usually include a small on-chip memory for storing sensitive data, memory controllers to access

[2]Hard core refers to cores fixed in silicon.

the external memory, a DMA controller for high speed data transfers, I/O, and various communication interfaces. Furthermore, the processing system may contain various levels of caches to enhance the performance of the system. The application processor may also include hardware-based protection mechanisms like MMU, IOMMU and an isolated execution environment, e.g. TrustZone, to enable run-time security for software applications running on the processor. An FPGA SoC may also contain cryptographic cores to decrypt and authenticate FPGA bitstreams and software applications running on the processor. However, these cryptographic cores are often not accessible by the user design for any other purpose. The different hardware modules in the FPGA fabric and peripherals are connected using either proprietary or standard bus systems like the ARM's advanced microcontroller bus architecture (AMBA) [ARM11], IBM's CoreConnect [IBM99] and Opencore's Wishbone [Ope10].

The design flow for FPGA SoCs is similar to traditional FPGAs but in addition to the bitstream generation, software drivers and applications are developed. The boot process of these contemporary platforms is different from traditional FPGAs. Depending on the vendor of the FPGA SoC different configuration processes are allowed:

- Processor boots first and configures the FPGA.
- FPGA is configured first and then configures the processor.
- Both FPGA and processor boot independently.

At present, there are three vendors manufacturing FPGA SoCs, namely Xilinx (e.g. Zynq-7000 and Zynq UltraScale+) [Xil17, Xil14], Intel (e.g. Cyclone V, Arria V, Arria 10 and Stratix 10) [Int17] and Microsemi (e.g. SmartFusion, SmartFusion2, PolarFire SoC) [Mic18a]. In this work, the Zynq-7000 FPGA SoC from Xilinx was chosen since it is a popular choice for contemporary embedded system designs. Also, the insights will be generalizable to later models as well as devices from different manufacturers.

Xilinx Zynq-7000 FPGA SoC

The Xilinx Zynq-7000 is an FPGA SoC consisting of a dual core ARM Cortex A9 CPU and a Xilinx 7-series FPGA fabric on the same die. The processing system includes an MMU and a two-level cache. Each processor has a private self-contained 32 KB level one data and instruction cache and a shared 512 KB level two cache with automatic cache coherency between the processor cores. A small on-chip RAM of 256 KB is available for the storage of sensitive information or code. A larger external DDR SDRAM can be accessed via the memory controller. The main memory bus is an ARM AMBA AXI bus system. The processor, hardware modules in the FPGA fabric and other on-chip peripherals communicate with each other over this bus system. External communication is supported through CAN, I2C, Ethernet and USB interfaces. The device also includes an advanced encryption standard (AES) and hashed message authentication code (HMAC) hard-core hardware accelerators which are used during boot for the decryption and authenticity verification of images and configuration files to be run on the Zynq-7000. The ARM trusted execution environment known as TrustZone is available for runtime security of the software system.

The boot process in the Zynq-7000 follows the principle that the processing system is the master and is responsible to load the boot image. The boot image of the Zynq-7000 consists of the unmodifiable BootROM code, first stage bootloader (FSBL), the bitstream for the FPGA fabric, second stage bootloader and finally the OS or the bare metal application. The Zynq-7000 supports both secure and non-secure boot process. A detailed description of this will be provided in Chapter 5.

Chapter 3

Study about the Threats of Hardware Trojans

Today, as businesses outsource parts of the design and fabrication process of ICs, a new threat is rising through the inclusion of malicious functionality in the hardware design, commonly known as *hardware trojans*. In this chapter, the issue of hardware trojans is comprehensively analyzed starting with the different possibilities to include trojans in the development and life cycle of ICs. The different attack vectors and entities capable of inserting trojans at each development stage are identified. For each, the cost, time and design knowledge required by an adversary to insert a trojan at the corresponding development stage is also analyzed. This comprehensive analysis is used to estimate the difficulty for an adversary to insert a trojan.

In general, the threats outlined here apply to both ASICs as well as FPGAs as they both outsource parts of the design and development process. The threats of malicious inclusions in an FPGA are twofold: (i) the threat of malicious inclusions during the fabrication of the FPGA (which is similar to the threats on an ASIC), and (ii) the threat of malicious inclusion during the development of an FPGA-based design, e.g. in the form of a corrupted IP or by reverse engineering the bitstream to insert malicious functionality. The latter is also the main difference between malicious inclusions in ASICs and FPGAs, as FPGA designs do not have a mask. Thus instead of reversing the mask, an attacker would have to reverse the bitstream.

After we have identified the different threats affecting the IC development and life cycle, techniques to detect and prevent trojans are studied. A comprehensive survey of different hardware trojan detection techniques published in literature is provided. Continuing with a similar structure as the trojan design and insertion stage, the detection schemes are classified based on the IC development stage where they can be used to detect and/or prevent trojans. For each technique, the costs in terms of tools, time, manpower, and knowledge required to carry out the detection technique are estimated.

The goal of this chapter is to provide the reader a holistic view about hardware trojans design and detection techniques together with the costs associated with them.

The rest of the chapter is organized as follows: Section 3.1 outlines the related work. Section 3.2 provides an overview of the IC development chain. In Section 3.3, the threats of hardware trojan design and insertion are discussed. In Section 3.4, we describe the detection techniques that can be applied at each stage of IC development. Finally, we summarize this chapter in Section 3.5

The contents of this chapter were part of the publication titled *Hardware trojans: current challenges and approaches*, in the journal *IET Computers & Digital Techniques* [JMHS14].

3.1 Related Work

An overview into the topic of hardware trojans was provided by Tehranipoor et al.[TSZ+11] [KRRT10]. They proposed a similar taxonomy for trojan insertion, i.e. classify published trojan designs based on the IC development stage. In contrast, in this work in addition to providing a comprehensive study of the publicly known hardware trojan design and detection schemes, the threat and feasibility of trojan insertion and detection at each development stage is evaluated. This in our opinion is essential to understand the risks associated with hardware trojans.

3.2 IC Development Chain Overview

The IC development chain can be divided into four main phases functional design, physical design, test program development and fabrication. The IC development process is illustrated on the vertical axis in Figure 3.1 along with the FPGA development chain marked in blue (a detailed description of an FPGA design flow is provided in Section 2.3.6). Through the course of this chapter the threats of trojan insertion at each development stage and the corresponding detection techniques that can be applied will be discussed.

3.2.1 Functional Design

During this stage, the functional behavior, interfaces and protocols are specified. First, a high level device specification is described, followed by the

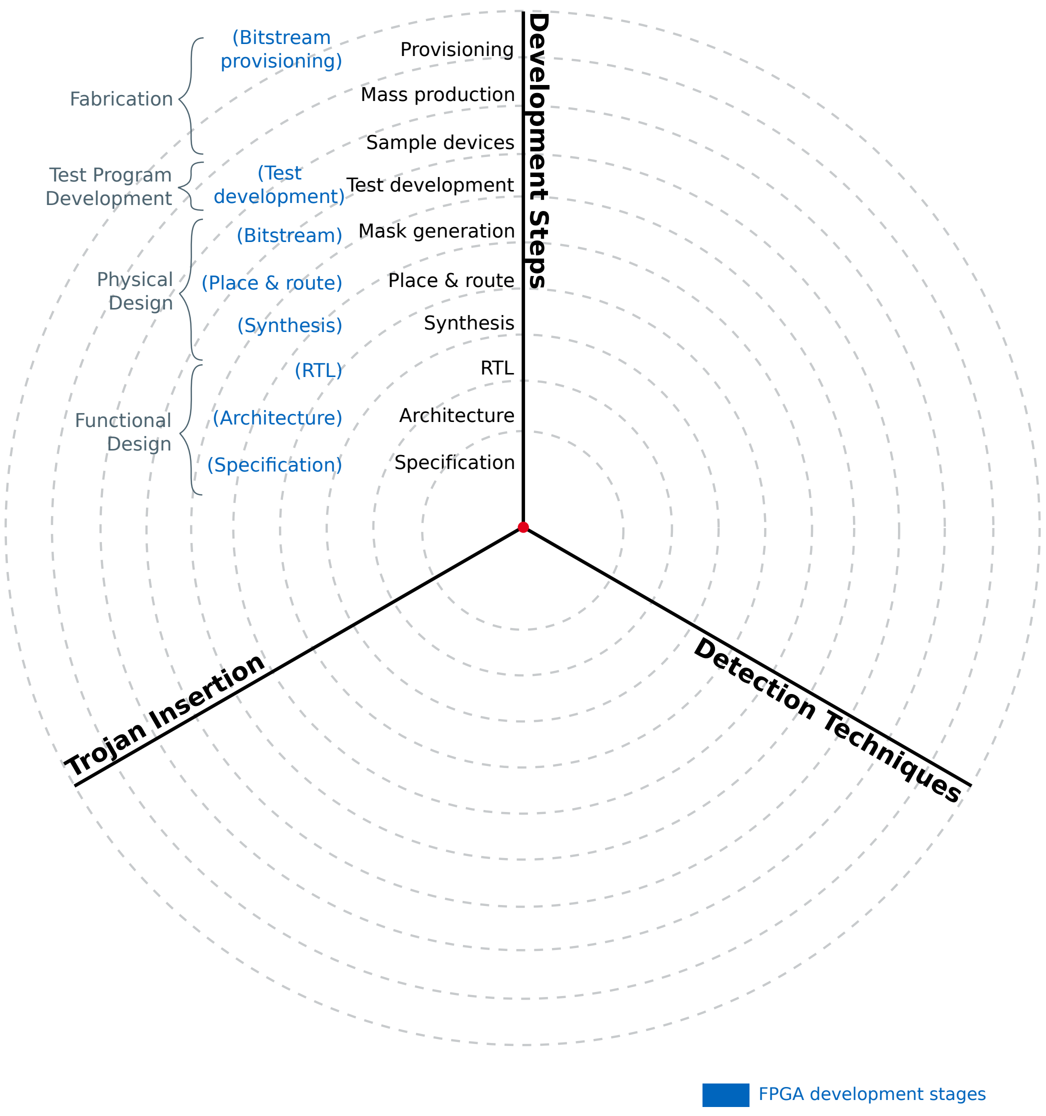


Figure 3.1: IC development chain

device architecture. Often they are described in plain text leaving room for inconsistencies. In some cases, however, the specification and architecture may be formally defined using architectural description languages. Next, the specification is translated into logic functions which are described in the register transfer level (RTL). Hardware description languages (HDLs) such as Verilog, VHDL or Systemverilog can be used for this. Furthermore, third party RTL IP blocks, like processor cores, cryptographic accelerators, are integrated into the design at this stage. Some big foundries have their own mask shop but others rely on third party mask shops. This stage of the development is similar for both FPGAs and ASICs as also shown in Figure 3.1.

3.2.2 Physical Design

Once the logical functionality is described, the RTL description is synthesized to lower abstraction levels and converted to the technology specific netlist using EDA tools. Next, the placement and routing (P&R) is performed. The gates of the netlist are placed on a 2-D floorplan and then routed depending on the design constraints like timing, area and power. Here designers are reliant on the technology libraries provided by the foundries that are ultimately in charge of the fabrication. Also during this stage, third party hard macros such as analog macros and memories are integrated into the design. Once it is complete, the mask layout is generated. The mask set is later sent to the foundry which produces the final IC. The graphic database system version II (GDSII) is a standard used in the industry to store the mask layout of a design. Some big foundries have their own mask shop but others rely on third party mask shops.

In case of FPGAs, no mask layout is generated following the P&R, instead a bitstream of the implemented design is generated. During the P&R stage, placed and routed third party IP cores, e.g. the Xilinx Security Monitor (SecMon) [Xil15], can be integrated to the design. Although mature open-source tools are available for the synthesis, the designers are reliant on vendor provided tools for P&R and bitstream generation as each FPGA family has their own proprietary bitstream format.

3.2.3 Test Program Development

In this phase, the test programs are developed, which are later used to configure and test the IC. For instance, boot loaders, keys, unique identifiers and seeds for pseudo random number generators (PRNGs) are defined. Furthermore, scan chains used to detect manufacturing faults or built-in-self-tests used to test the memory and functional tests for the IC are designed during this phase. Electrical or optical fuses to be blown are defined by the test program.

Similarly for FPGA-based systems, test programs are developed which are then used to test the generated bitstream. In addition, different configuration settings, e.g. eFuse setting for bitstream encryption and configuration interfaces and keys that need to programmed into device are defined.

3.2.4 Fabrication

Prior to the mass production, sample devices are fabricated to validate the desired functionality. Once the sample devices are fully characterized and tested, the mass production of the device is carried out. Firmware for embedded devices is downloaded during the provisioning stage and each device is then validated using the test programs developed in the previous stage. Finally the ICs are packaged and shipped for integration into the end product.

As for FPGA-based systems, only the final bitstream provisioning step is done where the bitstream is loaded to the NVM in case of SRAM-based FPGAs and programmed into the device in case flash-based FPGAs. The previously defined configuration settings are also burnt into the device during this step. Following which the FPGA is ready to be deployed in the field.

3.3 Hardware Trojan Design and Insertion

In this section, the vulnerabilities and the types of trojans that can be inserted in each stage of the IC development are described as shown in the *Trojan Insertion* branch of Figure 3.2.

Hardware trojan designs can be described based on their physical properties, activation and action characteristics [WTP08].

Physical properties: Hardware trojans can be divided with respect to the following attributes.

- Type: Describes whether there has been addition/deletion of gates or modification of wires.
- Size: The number of components in the trojan circuitry.
- Structure: Refers to whether or not modifications are made to the original layout.
- Distribution: Describes whether the trigger and payload circuit are placed together or distributed.

Activation characteristics: Describes the characteristics of the trigger signal.

- External signal: This way the attackers can turn on the trojan at a time of their choice.
- Internal signal: Refers to trojans that are always active or conditionally activated by a sensor value or an internal signal pattern.

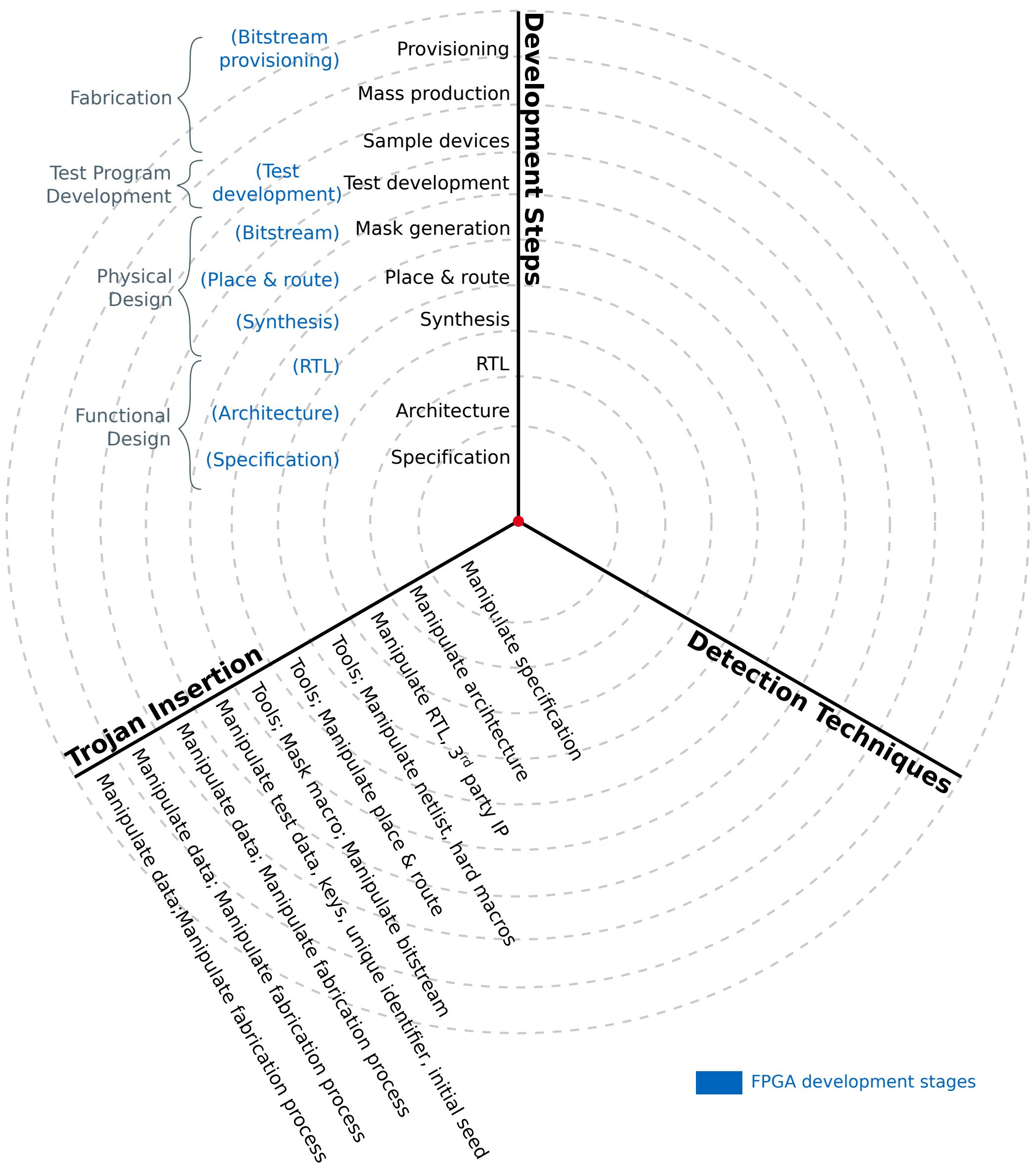


Figure 3.2: Threats of trojan insertion in the IC development chain

Action characteristics: Describes the characteristics of the payload.

- Modify functionality: Trojans that alter the original functionality of the IC.
- Transmit information: Trojans that leak information.
- Denial of service (DoS): Trojans that temporarily or permanently deny or interrupt services of the device.

Table 3.1: Selection of important hardware trojan survey

Paper	Possible Insertion Phase	Action Characteristics			External Trigger	Internal Trigger	
		Modify Functionality	Transmit Information	DoS		Always On	Condition Based
Jin et al. [JM08]	Functional Design			●		●	
Agrawal et al. [ABK+07]	Functional Design	●					●
King et al. [KTC+08]	Functional Design	●			●		
Lin et al. [LKG+09]	Functional Design		●			●	
Kutzner et al. [KPS13]	Functional Design		●				●
Chakraborty et al. [CSPN13]	Physical Design			●		●	
Muehlberghuber et al. [MGK+13]	Physical Design			●	●		
Bhasin et al. [BDG+13]	Physical Design	●			●		
Swierczynski et al. [SFK+17]	Physical Design	●				●	
Ender et al. [EGMP17]	Physical Design		●				●
Becker et al. [BRPB13]	Fabrication	●				●	
Yang et al. [YHD+16]	Fabrication	●					●

Having understood the different characteristics of a hardware trojan, the threats of malicious modifications are evaluated with respect to the IC development chain as shown in Figure 3.2. For each of the four development phases we classify the entities capable of inserting trojans, e.g. a malicious insider or a third party delivering IP and describe their capabilities. Along with the entities capable of inserting trojans, publicly known trojan designs corresponding to each development stage are described. A number of open source hardware trojan design benchmarks are available on *trust-hub.org*. Here however, we do not list all of those benchmark designs, instead we describe the different types of trojans and provide examples for the same. Table 3.1 provides a summary of the relevant hardware trojan designs together with their corresponding characteristics like activation and action characteristics.

3.3.1 Functional Design

Looking back at the IC development chain in Section 3.2, an adversary can manipulate the functional design by altering the specification and architecture or by including malicious third party IP cores. There can be several entities influencing these stages, for instance, a malicious insider in the design house or even a third party like a competitor, research institution, or governmental organization. Below we describe the capabilities of these entities:

* *Malicious insider in the design house.* A malicious insider with unrestricted access to the functional design can modify the specification, so as to leave room for insertion of a trojan during a later design phase. An insider can also manipulate the RTL and insert malicious code.

* *Third party.* Third parties have different opportunities to influence the specification. A third party contractor who develops parts of the specification or IP cores for the design house can include malicious elements. Governmental organizations can force design houses to include malicious elements or include weaknesses in the functional design through laws [BBG13]. Third parties can also directly influence the standard committees and the standards itself [ARS13]. The designers would be obliged to comply with the standards. Standards are not considered as trojans but they can impair the system just as much as a trojan.

There have been several proposed trojan designs that can be inserted in the RTL design to cause DoS, modify functionality and transmit information out of the device.

DoS trojans can be built to deny services of the device temporarily or permanently. For instance, a counter-based trojan can be used to increase the power consumption of the circuit [JM08] or as a time-bomb which shuts down the system when a particular count is reached [ABK$^+$07]. The advantage of these trojans is their small size due to which their effects are normally camouflaged by the circuit noise, hence making their detection very difficult.

Comparator-based trojans can be used to modify the functionality or insert faults into the system. Two or more signal values can be monitored to conditionally trigger a trojan that alters the value of signals of interest to execute undefined or out-of-specification functions. Agarwal et al. [ABK$^+$07] use a comparator-based trojan to insert a fault during the Chinese remainder theorem (CRT) inversion step of an RSA signature computation which in turn facilitates the recovery of the secret exponent. King et al. [KTC$^+$08] use hardware-software co-design to insert a trojan in the Illinois Malicious Processors. They design a hardware that supports post-fabrication trojan insertion through malicious software by sending a specific sequence of bytes. Once the trigger sequence is received, the memory privileges are disabled giving the software access to restricted memory regions. To achieve this, they modify the data cache of the processor to add an additional state and circuitry in the memory management unit. Kutzner et al. [KPS13] design a trojan that transmits the last round key of AES instead of the cipher text, thus transmitting sensitive information. This trojan is conditionally triggered by a 64-bit trigger sequence. These types of trojans can be detected using functional tests only if the correct trigger sequence is provided, which might not necessarily be the case during traditional post-fabrication tests as trojans are designed to have rare trigger conditions. Hence, code review or additional dedicated tests that amplify the effects of rare-conditions would be necessary to detect such trojans.

Trojans can also be designed to actively leak data such as secret keys from the IC. Previous research has shown how secret keys of block ciphers can be transmitted using power side-channel [LKG$^+$09] and hidden in the process variations that occur in the amplitude and frequency of wireless transmission [JM10]. These types of trojans cannot be detected using conventional post-fabrication functional testing techniques.

3.3.2 Physical Design

The different vulnerabilities and entities capable of inserting trojans at this stage of the IC development are enlisted below:

* *Malicious insider in the design house.* A malicious insider can easily modify the design files and insert macros during the design synthesis. At this stage, designers heavily rely on the design tools, which are generally closed-source, to develop their design. An adversary could maliciously modify the design tools or even alter the P&R process so as to make room for the trojan circuity to be inserted during a later design stage, e.g. mask generation or fabrication process.

* *Third party.* Design houses buy third party IP cores, which may include malicious manipulations. Third party IP providers may not have a very controlled development chain. Also, semiconductor companies use third party technology libraries and EDA tools for many critical design stages. As these entail trade secrets, the sources of these libraries and tools are usually not publicly available. A malicious entry in these tools can directly affect the hardware design of many designers. Design houses may outsource the mask generation to a third party company. This gives third party companies the opportunity to maliciously include mask macros in the GDSII.
  As for FPGA-based designs, an adversary would have to reverse engineer the bitstream to include malicious functionality instead of reversing the GDSII layout.

In contrast to trojan insertion during earlier design stages, trojan insertion at this stage requires more resources and effort, e.g. to reverse engineer masks and bitstreams. Examples of trojan designs from literature that can be inserted during this stage include modification of the mask layout and in the case of FPGAs, modification of the bitstream.

Muehlberghuber et al. [MGK$^+$13] and Bhasin et al. [BDG$^+$13] show how the GDSII mask layout can be modified to insert a DoS trojan with a kill switch and a trojan that inserts a fault into the AES block cipher to enable differential fault attacks, respectively.

Works from literature also show how trojans can be inserted in the bitstream of FPGA-based systems. Chakraborty et al. [CSPN13], Swierczynski et al. [SFK$^+$17] and Fyrbiak et al. [FWS$^+$17] reverse engineer the bitstream of FPGAs to insert malicious functionality to cause early aging of the FPGA and weaken the security of cryptographic cores within the FPGA. An easy and already widely deployed mitigation technique against reverse engineering of bitstreams is bitstream encryption and integrity checking. Bitstream encryption makes it hard for an adversary to reverse engineer the bitstream as first the encryption key would need to be obtained only following which can the bitstream be reverse engineered. While bitstream integrity check prevents an adversary from malicious modifying the bitstream to include additional functionality.

Ender et al. [EGMP17] design a trojan by modifying the placement and routing of the FPGA design to cause the design to function maliciously at higher clock rates. The trojan was then used to leak side-channel information of a side-channel protected symmetrical cryptographic core.

3.3.3 Test Program Development

Test programs can be manipulated by a malicious insider or a third party. Below these vulnerabilities are discussed:

* *Malicious insider in the design house.* The malicious insider may modify the keys or add additional keys during the test development. They can also modify the keys and identifiers to be programmed into the device. Initial seeds of PRNGs can be manipulated and set to a number that is known to have certain weaknesses. The debug interface can also be left open, which can later be used to exploit the device.

* *Third party.* A third party providing IP cores to a design house can provide malicious test programs. This could lead to malicious functions in the design going undetected as the provided testbenches do not cover the corner cases.

3.3.4 Fabrication

Malicious manipulations at the time of fabrication can be due to malicious personnel in the foundry or a third party influencing the fabrication process as described below.

* *Malicious personnel in the foundry.* The personnel with access to the fabrication process and tools can insert a trojan into the IC by modifying the dopant level or the mask layout provided to them either during the sample or mass production.

* *Third party.* In the fabrication stage, again governments can regulate the production through laws. In addition, foundries use third party tools that can be manipulated.

Becker et al. show how the dopant levels can be modified during fabrication [BRPB13] to create a potent trojan. They design an always-on trojan in the sub transistor level by modifying the circuit type. This is done by modifying the polarity of the dopants, so that the transistors always have a constant output and thus modify the functionality. This modification does not require any additional gates. As a proof-of-concept, they stimulate this trojan in an Intel Ivy Bridge random number generator (RNG). They observe that their trojan infected circuit passes the built-in-self-tests. They also find that, by having an entropy of at least 32 bits, the trojan RNG even passes the NIST test suite. This is because the RNG uses AES, which is a strong post-processing scheme. This kind of trojan may even go unnoticed during traditional reverse engineering. Sophisticated and expensive equipment would be required to detect such a trojan.

Yang et al [YHD$^+$16], designed a stealthy analog trojan that can be inserted during fabrication without changing the original design by leveraging the empty spaces after placement and routing of the design. The trojan circuitry consists of capacitors that draws charge from neighbouring circuity to change the output of the flip-flop controlling the privilege bit of a processor. By controlling the output of this flip-flop an attacker would be able to execute a privilege escalation attack. The size and design of this trojan make it extremely difficult to spot the trojan using only traditional post-fabrication testing. Invasive techniques like reverse engineering would be required to detect the changes made to the design.

3.3.5 Difficulty of Trojan Insertion

Having understood the threats, vulnerabilities and entities capable of including malicious functionality together with a survey of previously published tro-

Table 3.2: Difficulty of trojan insertion

Development Stage	Cost	Time Constraints	Design Knowledge	**Insertion Difficulty**
Functional Design				
Specifications	Low	Low	Low	**Low**
Architecture	Low	Low	Low	**Low**
RTL	Low	Low	Medium	**Low**
Physical Design				
Synthesis and P&R	Medium	Low	Medium	**Medium**
Tools	Medium	Low	Medium	**Medium**
Mask	High	High	High	**High**
Test Program Development				
Test development	Low	Low	High	**Medium**
Fabrication				
Fabrication	High	High	High	**High**

jan designs, we next examine the cost of such inclusions. Here we combine the knowledge about the IC development cycle from Section 3.2 and threats of trojan insertion Section 3.3 to estimate the difficulty to insert trojans.

In Table 3.2, we derive the difficulty of trojan insertion by adversaries at each stage of the IC development. The difficulty of trojan insertion is derived based on the estimated cost, time and design knowledge needed by an adversary. Cost refers to the manpower, budget and tools required by an adversary. Time constraints refers to the time an adversary has between stages to insert a trojan. Design knowledge is the insight the adversary would need to have about the design to insert a stealthy trojan. This helps us identify the critical stages of the IC development cycle and thereafter find appropriate countermeasures to harden these stages against the threat of malicious inclusions.

In our opinion, trojan insertion in the functional design stage is relatively easy and has a low cost. Furthermore, hardware accelerators from various vendors are integrated into the design during this phase. The adversary has more freedom, as there is no strict time constrains and there is no need to perform complex reverse engineering. Manipulations at this level will be carried over to the next stages. Without a very deep insight into the design, malicious functionality in third party IP can be disastrous for the whole system security as will be shown in Chapter 4 and Chapter 5. Thus, there is low difficulty for trojan insertion which can be seen in Table 3.2 where it is marked in bold.

Trojan insertion during the physical design stage has a strong impact. The attacker, however, would need to have a deeper understanding of the functional design and tools. But the adversary is not bound by strict time constraints. Thus, combining all these factors we think that this stage poses as a medium difficulty for trojan insertion.

Inserting trojans in the test program generation can greatly impact the integrity of the device. These manipulations can be carried out at a relatively low cost as it does not require many resources. This stage is bound by low time constraints, but the adversary should have high knowledge of the design. Hence, insertions at this stage pose as a medium difficulty.

Trojan insertion during fabrication process is the most difficult and most expensive as the adversary has very limited information about the design. To insert a complex trojan, the adversary is required to perform reverse engineering of the mask layout in very limited time due to strict time constraints in the supply chain management. Also, the mask creation is the most expensive part of IC production. Masks can cost anything between 20K-2M USD for small feature sizes. This makes it harder for organizations with a limited budget to insert a trojan during fabrication. However, it is still possible for organizations with big budgets and manpower. It would also be necessary that the attacker has a high design knowledge so that he understands the design well and is able to successfully insert a trojan. Even small mistakes could lead to the failure of the design. In summary, the difficulty of trojan insertion at this stage is high.

3.3.6 Outlook for Hardware Trojan Threats

Having seen the current threats of hardware trojans, we look at some future threats affecting the integrity of ICs. Based on our analysis, the main and impeding threats affecting the integrity of ICs are:

1. Trojans insertion during early design stages i.e., inserting trojans directly into the RTL design of the IC. In particular, the integration of untrusted third party IP pose as a big challenge for the future. As not all IP vendors may have a very strict and controlled design phase.

2. Trojans insertion during the design of test programs. For instance additional keys with root access could be loaded or wrong fuses could be programmed. Third party IP vendors may provide test benches that do not cover or overlook critical test vectors. They could also provide malicious firmware for their products so as to leak critical information from the system.

3. Malicious IP cores in FPGA SoCs. Here the threat lies in the fact that in FPGA SoCs several resources are shared between the SoC and FPGA. Thus a corrupted hardware core can affect the integrity of the entire system.

3.4 Hardware Trojan Detection

In the previous section the different trojan types at each stage of an ASIC and FPGA development (Trojan Insertion branch of Figure 3.3) were outlined. Now we try to identify how trojans can be detected at different stages (Detection Techniques branch of Figure 3.3). Similar to Section 3.3, we provide a survey of previously proposed detection techniques with respect to the development chain. Table 3.3 provides a summary of all the trojan detection techniques together with their characteristics like the type of reference model used and design stage where they can be applied. This does not include organizational measures which can be applied in order to avoid trojan insertion.

3.4.1 Functional Design

The objective at this stage would be to obtain a trojan-free core root-of-trust which can be used as a reference during later development stages, a golden reference model so to say. Trojan-free designs can be obtained only if the specification is established as the root-of-trust. With this, a chain-of-trust must be built in all the following development stages. Malicious inclusions or modifications at this phase can be detected by thorough review.

The architecture level can be verified using formal verification. Formal verification of the architecture is possible only if the specification is formally defined. Detection at the RTL level can be done using formal verification techniques provided the architecture is formally defined.

Banga et al. [BH10] propose a technique to detect trojans in third party RTL using automatic test pattern generation (ATPG) tools and equivalence checking. They first eliminate easy-to-detect signals and then use ATPG tools to detect the hard-to-excite or rarely activated signals, since trojans are known to have rare triggers [DAR06, WPBC08]. Finally, the suspicious signals are compared against the specification provided by the IP provider using equivalence checking to find any malicious inclusions. Zhang et al. [ZT11a] use coverage metrics like line, toggle, statement and finite state machine to detect parts of code that have not been executed. From the synthesized RTL, all redundant code is removed to reduce the number of suspicious signals using formal verification and ATPG tools. They also propose to insert scan chains in the netlist and ATPG patterns to check for untestable stuck-at faults. Later, the gates with stuck-at faults are removed from the original design. If any suspicious signal still remains in the design, it is considered to be malicious. Hicks et al. [HFK$^+$10] search the HDL code of a design for unused logic. A dataflow graph of the code is generated, with signals and

Table 3.3: Survey of published trojan detection techniques

Paper	Detection Technique	Reference Model		
		Golden IC	Design Simulation	Specification
Functional design				
Banga et al. [BH10]	Formal Verification			●
Zhang et al. [ZT11a]	Formal Verification		●	
Hicks et al. [HFK+10]	Formal Verification			●
Zhang et al. [ZYW+15]	Formal Verification			●
Rajendran et al. [RDVK16]	Formal Verification			●
Waksman et al. [WSS13]	Functional Analysis		●	
Fyrbiak et al. [FWS+17]	Functional Analysis		●	
Physical design				
Potkonjak [Pot10]	Functional Analysis			●
Fabrication design				
Bhasin et al. [BDG+13]	Optical Inspection		●	
Zhou et al. [ZAZ+15]	Optical Inspection		●	
Jha et al. [JJ08]	Functional Analysis	●		
Chakraborty et al. [CPB08]	Functional Analysis		●	
Chakraborty et al. [CWP+09]	Functional Analysis		●	
Agrawal et al. [ABK+07]	SCA-Power	●		
Banga et al. [BCFH08]	SCA-Power	●		
Banga et al. [BH09]	SCA-Power	●		
Potkonjak et al. [PNNM09]	SCA-Power		●	
Alkabani et al. [AK09]	SCA-Current		●	
Wang et al. [WSTP08]	SCA-Current	●		
Rad et al. [RWTP08]	SCA-Current	●		
Jin et al. [JM08]	SCA-Delay	●		
Li et al [LL08]	SCA-Delay		●	
Cha et al. [CG13]	SCA-Delay	●		
Li et al. [LDT12]	SCA-Delay		●	
Balasch et al. [BGV15]	SCA-EM	●		
He et al. [HZGJ17]	SCA-EM		●	
Run-time				
Salmani et al. [STP09]	On-chip Sensors	●		
Salmani et al. [ST12]	On-chip Sensors	●		
Zhang et al. [ZT11b]	On-chip Sensors	●		
Narasimhan et al. [NYW+12]	On-chip Sensors		●	
Bao et al. [BFS15]	On-chip Sensors		●	

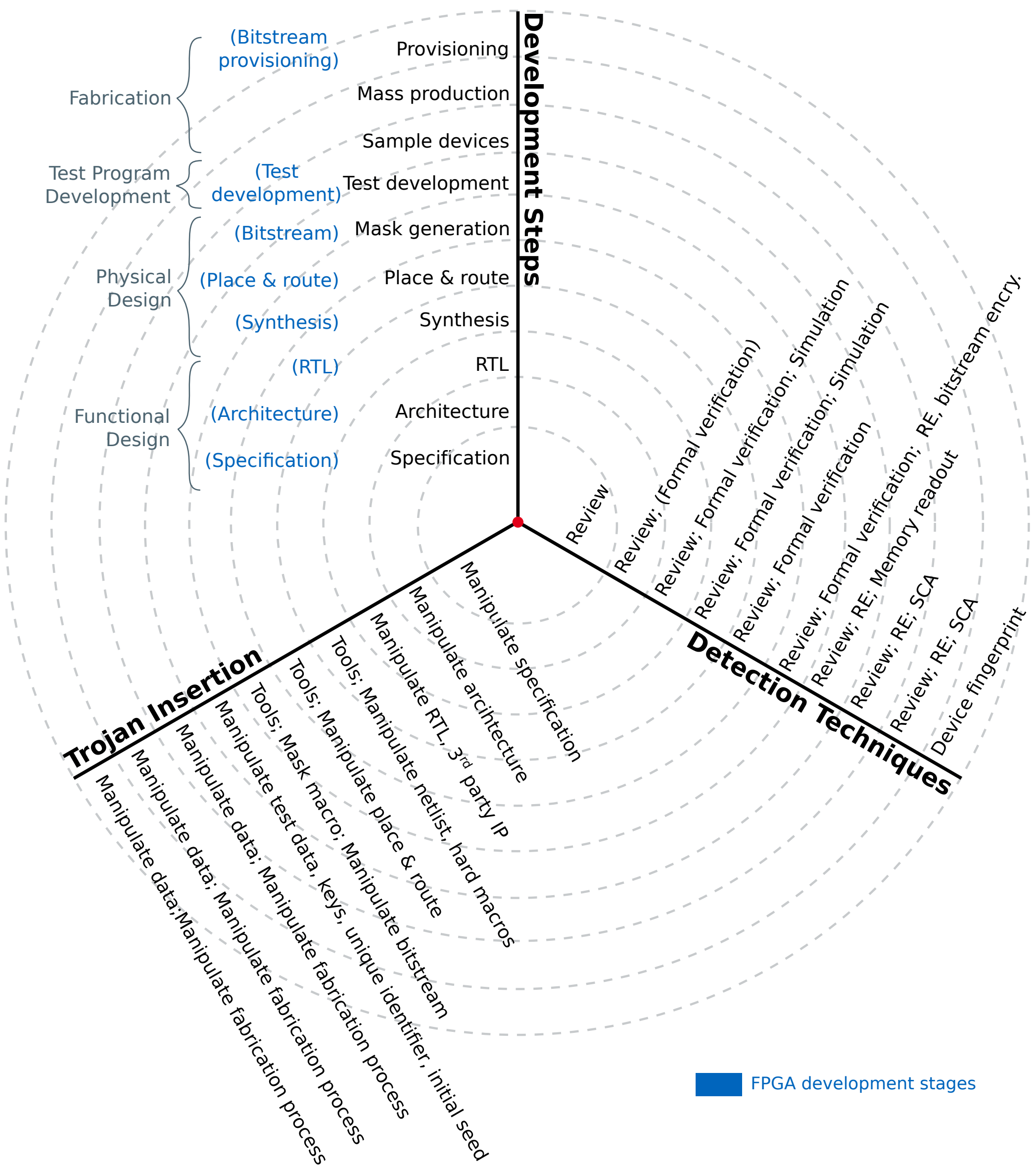

Development Steps
Provisioning
Mass production
Sample devices
Test development
Mask generation
Place & route
Synthesis
RTL
Architecture
Specification
Fabrication
(Bitstream provisioning)
Test Program Development
(Test development)
Physical Design
(Bitstream)
(Place & route)
(Synthesis)
Functional Design
(RTL)
(Architecture)
(Specification)
Detection Techniques
Review
Review; (Formal verification)
Review; Formal verification; Simulation
Review; Formal verification; Simulation
Review; Formal verification
Review; Formal verification; RE, bitstream encry.
Review; RE; Memory readout
Review; RE; SCA
Review; RE; SCA
Device fingerprint
Trojan Insertion
Manipulate specification
Manipulate architecture
Manipulate RTL, 3rd party IP
Manipulate netlist, hard macros
Tools; Manipulate place & route
Tools; Mask macro; Manipulate bitstream
Tools; Manipulate test data, keys, unique identifier, initial seed
Manipulate data; Manipulate fabrication process
Manipulate data; Manipulate fabrication process
Manipulate data;Manipulate fabrication process
FPGA development stages

Figure 3.3: Hardware trojan detection schemes

state elements as nodes while the dataflow between nodes is represented as edges. Subsequently, source-sink signal pairs are derived. By applying high coverage verification tests, any intermediate logic between a source-sink pair that does not affect the output is marked as suspicious. A similar detection technique was also proposed by Zhang et al. [ZYW+15], who build a detection tool, VeriTrust, that tests code with low coverage to detect trojans that add additional functionality. However, this tool was shown to be insensitive to trojans that are always on and island trojans i.e., trojans whose effects are not propagated to the output of the device like a network of ring oscillators that simply consume more power. Rajendran et al. [RDVK16] use formal verification using SAT solvers. They also use bound checks where the design is verified to see if all states are reached at the specified time, if not it could be due to the presence of a trojan. This technique targets trojans that leak information and use digital inputs for the trigger. Waksman et al. [WSS13] developed a tool, FANCI which uses boolean functional analysis to detect nearly unused circuits in third party IP cores. For this, they construct truth tables for all output wires with its corresponding input wires. Wires with probability below the control value are flagged as suspicious. Once, the wires are flagged as suspicious, code review is done to determine whether the suspicious wires are malicious. Fyrbiak et al. [FWS+17] build on FANCI to develop a static analysis tool ANGEL which uses boolean functional analysis and graph neighbourhood analysis i.e., instead of considering just a single gate like FANCI, the influence of the neighbouring gates are also considered.

The above proposed techniques can be used to reduce the design space and amplify the effects of the malicious inclusions in the design at relatively lower costs. A more robust technique at this stage for trojan detection is code review. This however, is a more time consuming process and requires the system designers to have a strong knowledge in hardware design. Simulation of the RTL can be used for evaluating code coverage, i.e. to check if all parts of the design have been tested or not.

3.4.2 Physical Design

The different trojan detection techniques that can be deployed in this phase include review, formal verification and functional analysis of the design. Formal verification techniques can be used to verify the results after synthesis and P&R. Designers at this stage are reliant on third party EDA tools for the development of their design, the EDA tools must be reverse engineered to check for correctness. An alternative could be to use multiple tools and compare the generated results with each other. Potkonjak [Pot10] developed a technique to design trusted ICs using untrusted EDA tools. This is achieved by specifying all the design aspects at every level of abstraction and accounting for all resources at each design stage thus leaving no room for the EDA tool to insert untrusted functions. For instance, they propose scheduling all operations, having no dont-care states in the finite state machine and using all the functional units in all the control steps. Due to the complexity of the tasks like resource allocation and scheduling, designers are still reliant on the untrusted EDA tools to perform these tasks following which the designer can check the generated design for correctness. This technique is not scalable for large designs, due to the high overhead is terms of time and resources required to check for correctness.

After the mask generation process, the design should be extracted and reverse engineered to guarantee that no malicious components have been included or malicious modifications of the design have not occurred during the mask generation.

3.4.3 Test Program Development

The next stage of the IC development is the test pattern development. All memory contents and fuse bits that have been programmed in the test program can be read out from the sample devices. However, this will not be possible, if the test program has switched the chip into the operating mode which does not permit reading all memory contents. Meaning the design flow would need to adapted to include checks before switching the chip into the operating mode.

Authentication and integrity checks can be performed inside a chip when configurations are downloaded [Mic15]. Configurations can be internally hashed and the hash can be output for verification [Mic13]. This prevents an adversary from maliciously reading out the configuration. However, this requires cryptographic modules to be present in the tested chip. Thus ensuring that only authentic configurations are loaded on the chip.

3.4.4 Fabrication

The different analysis methods used for hardware trojan detection in the fabrication stage are optical inspection, functional analysis, side channel analysis (SCA) and device fingerprinting. Below we describe each of these techniques together with examples from literature to show their capabilities and associated costs.

We begin with optical inspection as a way to detect trojans. Chips are reverse engineered, each layer of the IC is photographed and thoroughly examined for any addition, deletion, or modification of the design. This technique offers a high degree of assurance. It is also effective in detecting minor changes like inclusion and deletion of transistors. The main drawback of this method is the cost of the initial set up. For effective analysis, one requires dedicated equipment, which costs around USD 500K[3] and the whole process is very time consuming. Another drawback is that all the chips need

[3]http://www.siliconfareast.com/

to be destructively tested, thus rendering them useless. Reverse engineering even simple circuits like sensors can cost around USD 15,000[4].

Due to the high costs and destructive nature of reverse engineering, this technique is a commonly proposed method to derive a golden chip [ABK+07]. A golden chip is an IC that is assumed to be trojan-free. The golden chip is used as a reference chip against which different chip characteristics, like power consumption and delay, of all other chips can be compared to determine the presence of malicious functionality.

Optical imaging is another often proposed technique as a trade off between the high cost and accuracy achieved using reverse engineering. This technique has lower costs and also a lower accuracy in comparison with reverse engineering. Works using optical imaging include comparison of the post-fabrication images of an IC to the GDSII layout, i.e. layout versus image comparison [BDG+13] and deriving a optical fingerprint of a IC by distinguishing between the fill cells and the actual logic [ZAZ+15]. These techniques are only effective if the trojan has some connections in the higher metal layers, if the trojan is hidden in the lower layers it might go unnoticed.

Functional analysis is a conventional post-fabrication IC testing tool. It has a low cost but the tests are dedicated to verify the proper functionality of the IC. Trojans, however, are designed to be stealthy and evade the conventional testing phase [WPBC08]. Hence, conventional functional testing tools are not effective against trojan detection.

Researchers over the years have proposed various techniques to improve the conventional post-fabrication functional tests, to enhance the effects of the malicious functionality and thereby increase the probability of trojan detection post-fabrication. Jha et al. [JJ08] perform functional analysis of a device under test (DUT). First, a unique signature of the circuit is constructed. Once the fingerprint is obtained, a random set of input patterns are applied to the DUT. The DUT is treated as a black box. The probability distribution of the output is unique for each functionally different circuit. If the probability of logic 1 at an output is different for the DUT, then it indicates the presence of a trojan. When a trojan is detected, the fingerprint of the input pattern is returned. Such a set up works best on trojans that modify functionality. In this method, only the implemented circuit is tested for the presence of a trojan. A trojan could still be inserted in an unused part of the IC.

Chakraborty et al. [CPB08] include a special key-protected test mode into their design. The test triggers a set of rare events within the modules under test and generates a signature of the critical modules. This generated

[4]http://www.chipworks.com/

signature is sent to the external GPIOs of the chip. Such a technique works for small circuits, as generating multiple signatures for a complex design would be difficult and thus is not a very scalable technique.

Chakraborty et al. [CWP+09] developed a reduced test pattern that excites multiple rarely activated nodes. First, a golden netlist is simulated to identify nodes with low signal probability. Next, they generate a set of test patterns targeting the node with low signal probability. Their goal is to derive the minimal set of test patterns that should increase trojan coverage probability. Although this approach may be helpful in triggering a trojan, the effects of the trojan must be propagated to one of the primary output signals of the design to be detected. It can still be a useful approach when combined with side-channel techniques because it can increase the trojan activity.

SCA is the measurement of physical properties, such as power, electromagnetic radiation and delay, along with some statistical analysis while the device performs some critical function, e.g. while the trojan is operational. SCA has a low initial cost in comparison to techniques like the previously described optical detection techniques. The goal here is to derive a side-channel fingerprint from a set of sample devices, the fingerprint is later compared with all other chips after fabrication. If the fingerprint of the tested chips is different from that of the sample devices, they are deemed to have additional undocumented functionality. The main challenge with this technique is the noise caused by process and environmental variations. SCA was first used for trojan detection by Agrawal et al. [ABK+07] where they generate a power fingerprint from a set of randomly selected chips. Later, these chips are destructively tested, i.e., reverse engineered, to make sure that they are trojan free. All the other chips are compared against this fingerprint. Results were simulated in SPICE with different percentages of process variations. However, they show that small trojans could not be detected successfully due to the process variations. In the case of conditional trojans, only the trigger circuit is active at all times. The trigger circuit can be very small and composed of very few gates, which may go unnoticed by conventional SCA due to the high noise levels. Hence, detecting those trojans is most effective when the trigger circuit is not smaller than 3-4 orders of magnitude of the entire circuit [ABK+07].

In order to enhance the effects of the trojan activity Banga et al. [BCFH08] and Rad et al [RWTP08] developed a technique where they first divide the circuit into partitions and then measure the power consumption in each partition. They generate test vectors in a way to minimize the hamming distance between adjacent regions. The activity in one partition is increased while reducing the activity of the remaining, in order to maximize the influence of the trojan on the power consumption. Thereby reducing the impact of the process variations.

Further improvements were made to this technique, with a focus on methods to efficiently increase the activity on the chip, in an attempt to trigger the malicious functionality [BH09]. This is done by dividing the circuit into levels and inverting the voltage supply of alternate levels so as to cause switching in rarely activated gates. They then go on to use power profiling to detect any suspicious behaviour in the circuit.

In addition to power-analysis, researchers have also shown how other side channels like EM [BGV15, HZGJ17], transient current [WSTP08] and path delay [JM08] can be used to detect trojans. For instance, Wang et al. [WSTP08] show how trojans can be detected using a multi-supply transient current integration technique. It is done by measuring the current on multiple ports of the chips. Random sets of input patterns are applied to increase the switching activity. The current consumption from each port is integrated and compared with the worst case charges of the golden chip. The authors assume that the trojan is inserted only in a selected number of dies. The golden chip is retrieved by exhaustively testing several randomly chosen dies. Alkabani et al. [AK09] also propose a technique to measure the gate level characteristics of a circuit. They measure the static current leakage on the output port and compare it to the nominal leakage values of the gate. The nominal values are got by simulating the netlist with high coverage input vectors. They simulate a model of this technique and use a consistency metric to measure the difference between the real and estimated scaling factors of the gate leakage.

Measurement of path delay is also a widely proposed side-channel for trojan detection. Jin et al. [JM08] use path delay fingerprinting for trojan detection. First, a set of high coverage input patterns are applied to collect the path delay fingerprint of the golden chip. Later, the path delay of the DUT is compared to this fingerprint. Cha et al. [CG13] improve this technique by measuring the path delay along the shortest path as trojans are typically designed to be stealthy and not affect primary outputs. The advantage of this technique is that there is a higher influence of the trojan on the delay. And thus, the effect of the nominal delays and process variations is minimized.

Later works have also shown how a combination of multiple side-channels can be used to detect trojans. For instance, Potkonjak et al. [PNNM09] measure gate level characteristics like the delay and power together with statistical analysis to detect trojans. In summary, we see that SCA is effective for trojan detection if the overall switching of the circuit is kept low so as to highlight the influence of the trojan in the side-channel being measured.

In general, a skillful adversary would design a trojan to have a rare trigger so that they are not detected during the regular IC testing [DAR06, WPBC08]. Hence, by eliminating low probability paths, the probability of trojan activation can be increased. This also increases the probability of detection using other detection schemes like SCA, as the switching activity of the trojan circuitry is increased. A newer detection technique modifies the original circuit at design time by inserting additional circuitry or reordering existing circuitry to enhance detection by other techniques, like power, delay or logical testing, later at run-time. For instance, inserting dummy scan flip-flops in paths with low transition probability [STP09] or reordering the scan cells chains [ST12] have been shown to increase the probability of rare occurring path, consequently also increasing the probability of trojan detection using SCA. In another approach, Zhang et al. [ZT11b] inserted a ring oscillator (RO) network. The ROs were distributed all over the IC to detect abnormal switching activity. A golden IC is used to generate a fingerprint of the RO frequencies and then the DUT is compared against this fingerprint. Switching due to trojan activity affects the frequency of the RO network. As a result, the localized effects of the trojan are captured from the closest RO. Similar detection techniques have been presented using different on-chip sensors such as a network of current sensors to measure on-chip transient current [NYW$^+$12], on-chip temperature sensors to detect increased activity [BFS15], delay sensors to measure register-to-register path delays [LDT12, LL08]. On-chip trojan detection sensors while helping to increase the probability of trojan detection, comes at the cost of additional area overhead and a drop in the overall performance.

Chakraborty et al. [CB09] proposed to use design obfuscation to increase the complexity of reverse engineering for the attacker. They did this by adding additional states to the design. Such a technique increases the complexity of the attacker to insert a ingenious trojan in the design. But this involves adding additional circuity and thus reducing the performance of the whole design.

Following the fabrication of the chip, a unique response or fingerprint from each IC could be read out after the mass production, which can then be verified by the design house. This could for instance be achieved through the use of a physical unclonable function (PUF). PUFs can be used post-fabrication to guarantee that no manipulations or replacements have occurred during the shipment of an IC and also throughout the life cycle of the IC. PUFs exploit the manufacturing differences in ICs to generate a unique-per-device fingerprint. This unique-per-device fingerprint for a device is derived after fabrication during a so called enrollment phase. After the IC is deployed, a verifier can verify that the fingerprint has not changed. Any modifications

Table 3.4: Difficulty of trojan detection

Development Stage	Cost	Time Required	Technique Accuracy	**Detection Difficulty**
Functional Design				
Specification	Low	Low	Medium	**Low**
Architecture	Low	Low	Medium	**Low**
RTL	Low	Medium	High	**Low**
Physical Design				
Synthesis and P&R	Medium	Medium	High	**Medium**
Tools	High	High	Medium	**High**
Mask	High	High	High	**High**
Test Program Development				
Test development	Medium	Medium	High	**Medium**
Fabrication				
Fabrication	High	High	High	**High**

to the device would also alter the response. Thus preventing the insertion or replacement of ICs by malicious clones. This might be especially helpful for FPGA-based designs to prevent an adversary from modifying the bitstream after the system is deployed. Owen et al. [JHC$^+$18] developed a self-authenticating PUF that checks the PUF response at ever power-up. A difference in the PUF response is attributed to a malicious modification of the bitstream, in which case the system does not power-up. However care must be taken while integrating PUFs in the design, properties of the PUF like entropy, temperature effects, environmental effects and aging must be thoroughly studied before deployment.

3.4.5 Difficulty of Trojan Detection

Having described all the detection schemes currently publicly known, in this section we assess the difficulty of trojan detection using these published schemes. For this, we again go back to the IC development chain, and estimate which detection techniques are most suitable for each IC development stage and the estimated cost for these schemes.

From our analysis, we see that current publicly known detection schemes known mostly focus on post fabrication detection techniques. Table 3.3 shows that most of the techniques use a golden chip as their reference model. Detection techniques like optical inspection or exhaustive testing could be used to derive the golden chip. The DUT is compared against the golden chip but this is based on the assumption that such a golden chip exists. Another reference model that can be used is a design simulation. Assuming to have a trusted specification and design phase, one can use the simulation of a pre-fabrication design for comparison.

In Table 3.4, we derive the difficulty of trojan detection for each IC development stage based on the cost, time and accuracy using a particular detection technique. Cost is estimated based on the budget and tools required to set up the detection technique. Time required is the time needed for trojan detection and finally accuracy defines the reliability of the technique. Whenever there are several detection techniques, we use the one with the highest accuracy for our evaluation. Through this evaluation we try to determine the practicability of trojan detection at each stage of development. A low detection difficulty indicates that the technique is suitable for trojan detection while a high detection difficulty indicates that the technique is not optimal for trojan detection due to factors like associated cost and resources.

Trojan detection during the functional design stage can be carried out at a relatively low cost and time. Thorough review of the specification and architecture provides a medium accuracy for trojan detection. Code review and simulation of the RTL has a high accuracy of trojan detection at relatively low costs. Thus, in our opinion the difficulty of trojan detection at this stage is low.

During the physical design stage, an evaluator would require higher resources and more time for trojan detection as they might have to reverse engineer complex layouts and tools in this stage. Techniques like formal analysis and reverse engineering provide a high accuracy, but are expensive and time consuming. Hence, we conclude that there is a medium to high difficulty of trojan detection.

Trojan detection at the test program development stage has a medium cost and time due to the complexity of the design. Techniques like configuration readback and integrity checks provide a high accuracy level. Thus, there is a low to medium difficulty of trojan detection at this stage.

Current research mostly focuses on trojan detection during fabrication. The most accurate technique here is reverse engineering, which is, however, very expensive and requires a lot of time in order to perform a thorough analysis. But this still does not guarantee the integrity of all the other fabricated ICs. For instance, detecting changes in the dopant level can be very expensive, as one requires sophisticated tools to conduct reverse engineering [BRPB13]. Thus, this stage has a high difficulty with regard to trojan detection.

3.4.6 Outlook for Hardware Trojan Detection

Through this work we see that the cost and difficulty of trojan detection increases with each stage of IC development. The detection techniques that are most relevant in our opinion are code review of third party IP. This correlates with our earlier results which had shown that the early design stages of the IC development are most vulnerable to trojans threats, hence we need to protect these stages. Code review together with a high coverage simulation can be used to review the third party IP cores that will be integrated into the design during the functional design stage. This technique offers a high level of confidence for trojan detection and has medium overall cost.

3.4.7 Hardware Trojan Prevention

Although a variety of detection methods have been introduced, most are still not resilient against process variation and are highly dependent on trojan size and type. Several prevention methods were also proposed in the literature. Prevention methods can be categorized into:

1) Layout-filler: Here the unused portions of the circuit are filled with dummy logic [KAABS15] or standard cells [XFT14]. However, it is possible that an attacker is able to identify dummy logic from functional logic, a trojan can then be inserted by replacing the dummy logic.

2) Logic encoding: Also known as logic obfuscation, this method adds additional gates to the design in order to encode its functionality. By providing the correct key or sequence of inputs, the correct functionality is executed else incorrect outputs are produced [RPSK12, RZZ$^+$15]. Thus upon retrieving the key, an attacker would be able to insert malicious functionality.

3) IC camouflaging: Another technique to prevent reverse engineering of an IC to insert trojans is to camouflage the design. Similar to logic encoding, the design functionality is hidden. The difference is that unlike logic encoding where key gates are inserted, IC camouflaging synthesizes cells that look alike but have different functionality [LSM$^+$16, RSSK13].

4) Split manufacturing: To eliminate the threat from an untrusted foundry, split manufacturing was proposed where part of the design is fabricated by an untrusted high-end foundry while the other part is done by a trusted low-end foundry [JGS$^+$14, RSK13]. Thus leaving no foundry with full access to the design layout. An attacker from the untrusted foundry will not be able to reverse engineer the design from the partial mask-layout, and thus making is very hard to insert a meaningful trojan.

In general all these techniques harden the effort required by an attacker to insert a trojan by modifying the original design to include additional non-functional states.

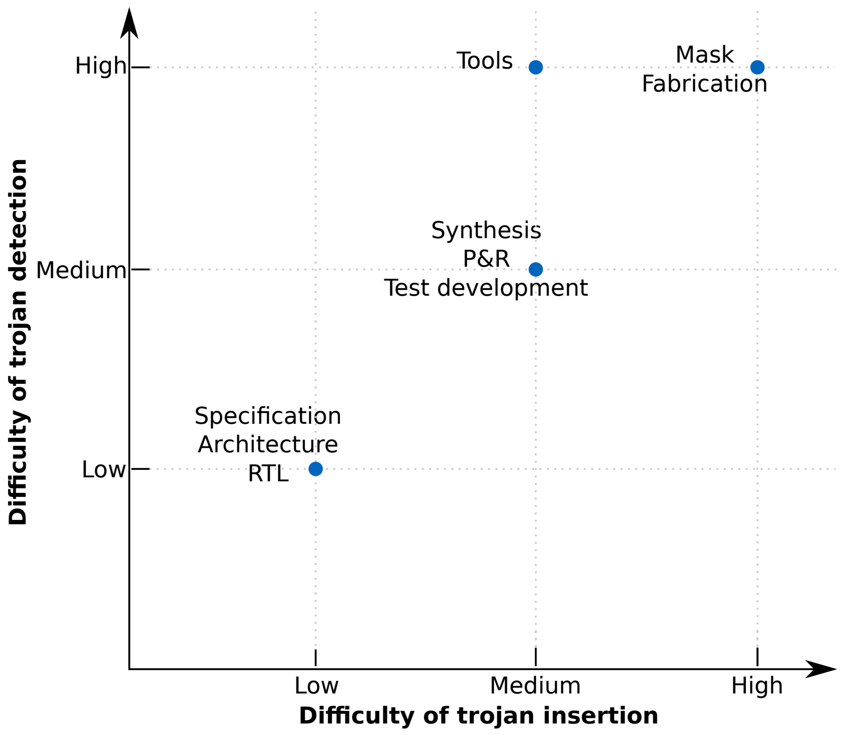


Figure 3.4: Feasibility of trojan insertion and detection

3.5 Summary

In this section, we analyze the difficulty of trojan insertion at each IC development stage against the difficulty of trojan detection as also summarized in Figure 3.4. The cost and effort of trojan insertion increases with each design step. Similarly, the cost of trojan detection increases as the complexity of the design increases with each development stage. Also with the increase in complexity, the time required for reliable trojan detection also increases and thus the difficulty of the trojan detection also increases.

From our analysis in Table 3.2 and Table 3.4, we see that the difficulty of trojan insertion during the functional design and test program development stage are low and also the cost for trojan detection is low. Further, the difficulty of trojan detection is low during the functional design stage and medium during the test program development stage. On the other hand, the difficulty of trojan insertion and detection is high during the fabrication stage. As here an adversary would need high budget and high design knowledge to insert a trojan. Trojan detection after fabrication can be expensive and time consuming as each fabricated device would need to be thoroughly tested.

Looking back at Figure 3.1 and Figure 3.4, as we move from the origin outwards, the cost of both trojan insertion and trojan detection increases with every circle, i.e. with every design stage. We identify that it easier for an adversary to make malicious manipulation during the early design phases than during fabrication. And it is also easier for the evaluators to detect trojans during the early design stages than post fabrication. Hence, we conclude that future research must focus more on securing the early stages of development against trojan insertion. Especially in the cases where IPs are sourced and integrated from third party vendors. Also, more effort should be spent on developing efficient detection techniques.

We use the observations made from this analysis as the foundation for the rest of this work. We exploit the findings and present two attacks that target the early design stage in Chapter 4 and Chapter 5. We also present two countermeasures that isolate malicious hardware core with access to memory in Chapter 6 and securely update an FPGA without having to rely on manufacturer-provided closed-source IPs in Chapter 7.

Chapter 4

Hardware Threats to Secure Software Update Processes on FPGA SoCs

Modern FPGA system-on-chips (SoCs) combine high performance application processors with reconfigurable hardware. This allows to enhance complex software systems with reconfigurable hardware accelerators. However, this combination gives rise to new security threats which can bypass state-of-the-art software protection mechanisms. Attacks on the software are now possible through the reconfigurable hardware as these cores share resources with the processor and may contain unwanted functionality.

In this chapter, we show how malicious functionality in a hardware module in the FPGA accesses and replaces critical memory sections. We carry-out a proof of concept on the Xilinx Zynq-7000 device which runs a Linux OS and a software application that verifies system updates. The malicious hardware core replaces the public key used to verify system updates, thus, allowing an attacker to maliciously update the FPGA SoC.

The chapter begins with a description of the software update process and secure updates on FPGA SoCs in Section 4.1. This is followed by the description of the attack vector in Section 4.2, related work in Section 4.3 and the practical demonstration on the Xilinx Zynq-7000 in Section 4.4. In Section 4.5 and Section 4.6 the results and different countermeasures are discussed, respectively. Finally, Section 4.7, provides a summary of this chapter.

The work presented in this chapter is part of the publication: *Compromising FPGA SoCs using Malicious Hardware Blocks. In Design Automation and Test in Europe, DATE 2017, Lausanne, Switzerland, March 2017* [JRZ+17].

4.1 Secure Software Updates on FPGA SoCs

Software update refers to the process of providing updates to the application or firmware running on the device. This is a critical functionality for software development as it is used to fix bugs or add new features to a running system that is already deployed in the field.

Updates to a system may be provided over-the-air, e.g. via wireless cellular service. For devices which are not connected to the internet or those that have no over-the-air update capability, the updates are downloaded from a remote server and transferred over external NVMs such as SD-cards or USBs. In the automotive domain this is commonly done by driving the car to the dealership where a technician downloads the updates and runs the update on the vehicle.

A common requirement for updates irrespective of how the updates are transferred to the device is the secure transmission and loading of the updates on the corresponding device. It is very important to verify the authenticity of the updates before they are loaded on the target device. In the absence of any verification, non-authentic software may be allowed to maliciously update the device. Signature verification and encryption can be used to verify the authenticity and maintain confidentiality of the updates. Common transport layer security (TLS) libraries such as OpenSSL or embedded device specific libraries such as mbedTLS or WolfSSL can be used to encrypt and authenticate updates.

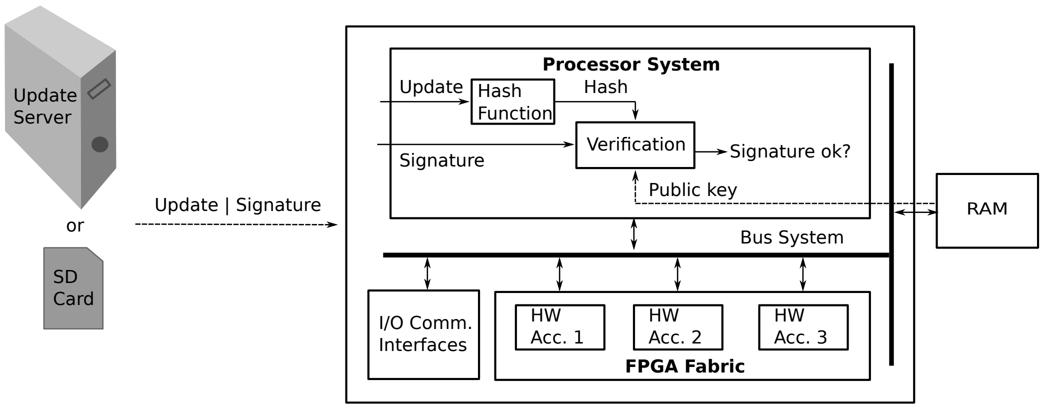


Figure 4.1: Secure update on FPGA SoCs

Figure 4.1 illustrates the process for verification of secure updates. It consists of a server from which the update can be downloaded and an FPGA SoC performing the signature verification process. On the server side, the hash of the update is computed and signed using the server's private key.

The update along with the signature is delivered to the FPGA SoC.

On the embedded system, an application with the help of a hardware accelerator (if available) or a software library, computes the hash of the update. Next, the signature of the update, the recomputed hash and the public key stored in the FPGA SoC are passed to the verification process. If the signature verification process is passed, the update is accepted by the system. If not, the update will be dropped and not processed.

4.2 Attacking the Secure Software Update Process on FPGA SoCs

FPGA SoCs share several resources between the FPGA fabric and processor, e.g. the on-chip memory, the DDR memory and the bus system. A novel attack vector for FPGA SoCs is the inclusion of hardware modules in the reconfigurable logic that maliciously exploit resources shared between the processor and the FPGA fabric. In order to reduce costs, hardware modules may be sourced from third parties. The problem is that malicious functionality could be part of such bought modules as already discussed in Chapter 3. Then they are part of a valid FPGA configuration file without the integrators knowledge of the malicious parts. As an analogy, think of flash-light applications for mobile phones that require internet and location access. They may contain unwanted functionality that is not supposed to be there. But a naïve user would download and use the application from the application store as if it were a legitimate application. Similarly, a hardware module in the FPGA fabric of the FPGA SoC with unwanted and undocumented functionality can for instance exploit the shared bus system by sniffing or corrupting the data being sent on the bus. In this work, we show how the shared memory can be used to retrieve or manipulate sensitive data, while they are loaded or stored in the random access memory (RAM) before being used for computation by the application processor.

Shared RAM memory is chosen as our target to show the attack since both SoC and FPGA fabric can access and are able to modify its contents. Also, the programs running on the application processor will be loaded to the RAM before their execution, this makes shared memory an ideal resource to corrupt. For instance, the memory can be scanned for sensitive data like keys that are used during secure update. Furthermore, the memory can be scanned for flags or jump addresses, which could be manipulated to modify the control flow. Once the memory location of interest is identified, the next step is to manipulate it in a meaningful way while not causing a partial or complete system failure. The manipulation needs to be stealthy and not alert the system about a manipulation. The attacker can replace the memory with his own code/data and, thus, modify the functionality of the system, e.g. modify data in order to bypass an authentication phase. Alternatively, an attacker could also leak some sensitive information like keys used for data encryption.

In this chapter, we describe a proof of concept of a malicious hardware module on an FPGA SoC that compromises an embedded system running an operating system and an application to verify secure remote updates. The goal of the attacker is to manipulate the signature verification process using hardware assistance. We built a custom hardware module that will be included in the FPGA fabric of the FPGA SoC. In addition to its regular and documented functions, the hardware module also includes hidden functionality. The hardware module scans the memory for the public key used for authentication and then replaces this key with a predefined key known to the attacker. By replacing this key with another key known to an attacker, the attacker is subsequently able to maliciously update the FPGA SoC.

The public key is identified by scanning for the header of *pem* formatted keys [All06]. Alternatively, the mathematical properties of asymmetric keys can be exploited to identify keys in the RAM, e.g. measuring the entropy since non-key data has lower entropy than key data [SS99]. In embedded systems, with limited resources, this key may be declared statically within the program binary to avoid the overhead of file handling. A keyfile may also be accessed using a file handler and then, its contents are copied to a buffer allocated in the stack or heap depending on the implementation. In both cases, the key will be loaded into the RAM, although at different locations.

4.3 Related Work

Previous publications have shown the threats of malicious functionality on the overall system security. Gonzalvo et al. [GBMB15] and Li et al. [LDP15] carry out similar attacks. Gonzalvo et al. have shown how the privileges of a user application running on a SoC can be escalated. They use the JTAG interface to scan the memory for a system function call and then overwrite it to grant the user application root privileges. The advantage with our set-up is that an attacker does not need physical access to the device and does not have to rely on an open JTAG port. The malicious functionality can be delivered in a third party IP core or it can be delivered during the life-time of the system in form of an update of the hardware module.

Li et al. developed a hardware module to trace the memory accesses of software applications running on the Xilinx Zynq-7000. They modify the entire architecture to pass also the memory traffic to and from the processor through the malicious memory tracing core in the FPGA instead of directly passing the traffic to the memory controller. Additionally, for the memory tracing tool, the Xilinx in system debug core, Integrated Logic Analyzer (ILA core), is used, which further adds the hardware costs. Leaving memory debugging in the final design not only causes a drop in the performance but also debug cores could be exploited by a malicious entity to gain insight/modify the functionality.

In contrast our work does not rely on any debug functionality to be present in the final design and also make no changes to the architecture of the standard software stack. We use a hardware core in the FPGA fabric of a Xilinx Zynq-7000 FPGA SoC with access to memory, which in addition to its regular and documented functionality scans the memory for the header of the public key used to sign system updates. Upon locating the public key in the memory, it is replaced by a key known to the attacker, thus allowing an attacker to maliciously update the system.

4.4 Practical Demonstration on Xilinx Zynq

We implemented a proof of concept on the Xilinx Zynq-7000 FPGA SoC [Xil14]. We use the Zedboard Rev. C, which is a development board for the Zynq.

4.4.1 Software

A realistic embedded system which runs a Linux operating system on the application processor is implemented. We use *Linux-xlnx* [Xil16c], which is a BusyBox Linux distribution for the Zynq devices from Xilinx. We also developed device drivers for the hardware module in the FPGA fabric, which the applications running on the OS can use to communicate with the hardware module. Currently, there is one core in the FPGA fabric that can be configured and used by applications running on the OS. The OS runs a software application that authenticates firmware updates using the elliptic curve digital signature algorithm. The FPGA SoC needs to verify the authenticity of updates. We ported the mbed TLS library, which is an open source TLS library for embedded devices [ARM], to our application processor to verify the signature and guarantee the authenticity of an incoming update. We assume a signed update is delivered to the FPGA SoC using an SD-Card or USB device.

4.4.2 Hardware Block Including Malicious Functionality

We implemented an AXI-Lite hardware core as a place holder for a cryptographic accelerator. This core exhibits the characteristics of a cryptographic hash core which would be used to calculate the hash of the update and would also be used as an accelerator for run-time integrity monitoring. For simplicity, we implement a 32-bit XOR operation as a place holder function.

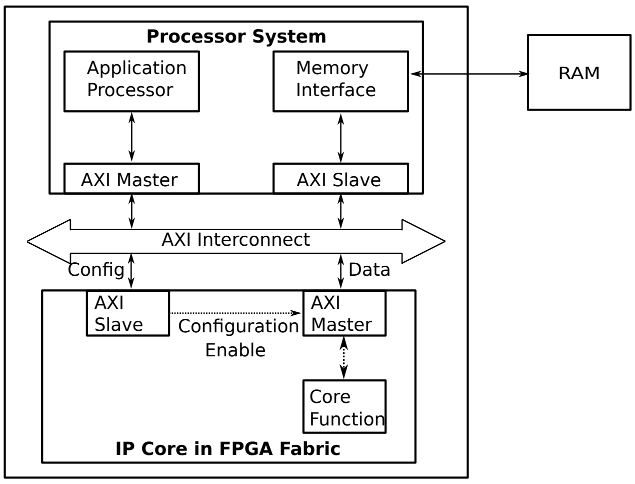


Figure 4.2: Design of IP core

A hardware accelerator for hash computations often needs to process large amounts of data. Thus, the core would likely be designed to receive configuration information from the processor and then operate autonomously which means it operates outside the control of the processor. This interface design is to our knowledge representative and commonly used in practice [Bar15, Ens13, Bar16]. The DMA-like interface reduces the load on the processor, as it is not involved in moving the data to and from the memory. The processor passes the source address, destination address and the length of the message as configuration information to the hardware core followed by an enable signal. Upon the completion of the computation, a done flag is set. A similar setup is described in a white paper from Freescale [Fre08]. In accordance to common practise, we designed our hardware module to have two ports as shown in Figure 4.2. An AXI-lite slave interface through which the enable and configuration parameters for the hardware core are passed from the processor. An AXI-lite master interface through which the core reads data from the memory, performs its core function based on the configuration parameters passed from the processor and writes the results back to the memory.

Besides the desired and documented functionalities, the hardware core performs the following hidden functions:

1. Scan the RAM memory in the background.
2. Identify the header of a `pem` formatted public key (`-----BEGIN PUBLIC KEY-----`).
3. Overwrite the stored public key with a key known to the attacker.

These hidden functions are performed in succession to regular and documented operations of the core. The key scanning operation is executed after every normal read operation of the core. Starting from the base address of the user memory, the memory is incrementally scanned with every regular read operation. This is feasible, if the hardware core is performing run-time integrity checks or remote attestation, the whole memory needs to be verified to show that the system is running unaltered software. As a result, the key scan operation will also scan the whole memory and the key header can be located in the memory.

As the keys will normally be loaded to the user space memory, it would be sufficient to only scan this section of the memory. If a standard Linux kernel is used the user space address mapping can be found publicly. In the cases, where this is not public knowledge, the whole memory would need to be scanned for the key. Once the key header pattern is identified, the key is overwritten with a new key hard coded in the bitstream or binary. The overwriting of the key is performed before the core performs its documented write operation. Currently, the malicious scanning is only performed when the hardware core is being used by the processor. By performing the hidden functionality in between the regular functions, the hidden functions can be disguised as legitimate operations of the hardware core. Thus, making their detection harder.

4.5 Results

The described implementation has been tested and *as an important result we were able to successfully modify the public key at run-time using a malicious hardware core in the FPGA SoC.*

Here we provide the implementation results of our prototype implementation for the Xilinx Zynq-7000 FPGA SoC. Our malicious functionality requires 38 additional flip-flops and 138 additional LUTs for scanning and storing the new public key. A SHA-384/512 IP core for the same FPGA family without the AXI interface requires 2526 LUTs (the interfaces require additional resources) [Hel16]. While a high throughput AES-galois counter mode (GCM) for the same FPGA family with an interface similar to our set-up requires 22729 LUTs [Bar15]. From the above we learn that our malicious functionality may have a relatively small area overhead. Each read operation reads 32-bit words. The Zedboard has a 512 MB large external DDR memory, in the worst-case scenario the whole 512 MB of memory needs to be scanned which takes approximately 7.2 seconds. As already discussed the region of memory to be scanned can be reduced based on publicly available data of the kernel memory map.

The goal of this work is to highlight the attack vector in FPGA SoCs, thus the area and performance of the hidden functionality is not the focus. This attack vector can be further extended to scan and modify flags, system function calls or symmetric keys. These attacks could be carried out with lesser hardware resources if no large payloads need to be stored in the hardware.

4.6 Discussion

Through this work, we can see that with the currently available security mechanisms on the Zynq-7000 such attacks cannot be prevented. Hence designers need to consider additional mechanisms to protect the systems against such threats. Here we discuss different countermeasures that are available and their impact against attacks like the one presented in this work.

The AXI interconnect on the Zynq can be programmed at design time to restrict the access of peripherals to the memory [Xil14]. By default, no restrictions are implemented on the AXI bus. In our opinion, the default setting is preferable, as it might be impractical to define the restrictions of the peripherals during the hardware design stage, as the hardware designer might not be completely aware of the software system.

One general way to protect the system against attacks through malicious functionality in hardware cores, is to conduct a thorough code review of the hardware cores and check the coverage of the testbenches supplied with hardware cores before integrating them into a system. Only thoroughly tested cores can be integrated into the system and thus prevent the whole system from being compromised by untrusted hardware cores. However, lack of time and competence can lead to small and stealthy malicious functionality being overlooked. Furthermore, in many cases, the hardware core is supplied as a netlist which contains no RTL description. Thus, further increasing the difficulty for a thorough review.

Shared memory attacks like the one presented here, can be prevented through the integration of IOMMUs which is called as SMMU by ARM [ARM16]. Using IOMMUs, the access rights of the peripherals can be dynamically managed as described in Section 2.2. An IOMMU is available in the newest high-end FPGA SoCs like the Xilinx UltraScale+ [Xil17] and Intel Stratix 10 [Alt15]. These new FPGA SoCs are significantly more powerful and expensive when compared to the FPGA SoCs like the Xilinx's Zynq and Intel's Arria 10. As our FPGA SoC does not have an IOMMU, an IP core can be developed, which emulates the behaviour of an IOMMU. Brunel et al. [BPOD14] develop such an IP core for SoCs. However, this core is resource intensive. Instead, an easy-to-use and lightweight security enhanced AXI-wrapper (like a stripped down IOMMU) can be integrated to every hardware core with direct access to the memory. A detailed description of the security enhanced wrapper will be provided in Chapter 6.

4.7 Summary

We demonstrated how the software in FPGA SoCs can be corrupted using malicious hardware cores. This threat arises from the fact that the hardware cores on an FPGA SoC are connected through a shared bus system. A malicious hardware core connected on this bus may be used to attack other resources connected on the same bus. In this work, we focused on hidden functionality in third party hardware cores that can be used to circumvent memory protection and corrupt the software running on the processing system. As a proof of concept, we show how the secure update process can be manipulated using a malicious core in the FPGA.

A general way to protect against this is to thoroughly test all third party hardware cores before integration. However, this is a complex and time consuming process. In newer FPGA SoCs, these threats can be handled through the use of an IOMMU. However, the lower-end devices which are already in use are still at risk as they do not have an IOMMU. For such systems, a security enhanced AXI-wrapper can be developed that prevents malicious cores from accessing unauthorized sections of the memory.

Chapter 5

Hardware Threats to Secure Boot on FPGA SoCs

This chapter demonstrates how an FPGA hardware design in FPGA SoCs can compromise the important secure boot process of the main CPU to boot from a malicious network source instead of an authentic signed kernel image. We perform a proof of concept on a Xilinx Zynq-7000 FPGA SoC and generalize the threat to other devices.

The secure boot mechanism on the Xilinx Zynq-7000 is described Section 5.1. Section 5.2 outlines the attack on the secure boot process. In Section 5.3 the related work is discussed. Section 5.4 provides a detailed description of the attack. In Section 5.5 the results of this work are discussed and in Section 5.6 the results are generalized to other platforms. Section 5.7 provides a summary of this chapter.

The results of this work were part of the publication: *How to break secure boot on FPGA SoCs through malicious hardware*. In Cryptographic Hardware and Embedded Systems - CHES 2017 [JHZ+17].

5.1 Secure Boot Process on the Xilinx Zynq-7000

Secure boot is one of the most fundamental security mechanisms. It is the foundation for all later security mechanisms. As described in Section 2.1, secure boot assures that no unauthorized code is allowed to run on a system. This is achieved by verifying all the code to be executed for integrity and authenticity using cryptographic means before its execution. A chain-of-trust is established starting from a hardware core root-of-trust which can be keys in eFuses or internal hardwired ROM code.

The goal of this chapter is to highlight that even for FPGA SoCs with a secure hardware root-of-trust and a secure boot process, as well as runtime protection (by isolation of software components), there is still a major risk that the system may be compromised. This risk comes from the reconfigurable hardware.

Figure 5.1 depicts the boot process on the Zynq-7000 FPGA SoC. It consists of five stages after power-up:

1. BootROM (non-accessible internal hardwired code)
2. First stage bootloader (FSBL)
3. FPGA configuration (bitstream)
4. Second stage bootloader (i.e. U-boot)
5. Operating system (OS)

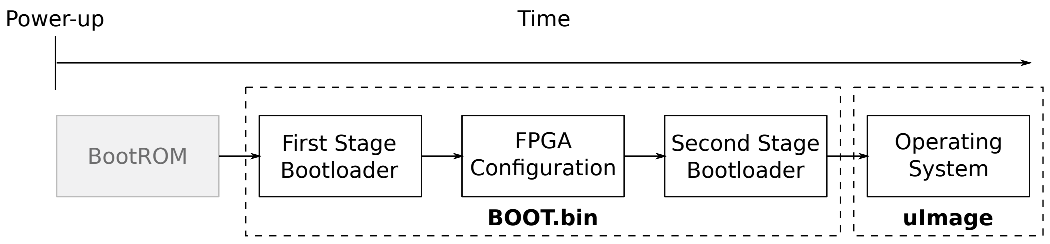


Figure 5.1: Overview of boot process on Xilinx Zynq-7000

The FSBL, bitstream and second stage bootloader are packed into a single boot image, i.e. BOOT.bin as separate partitions. Each partition within the boot image is separately encrypted and authenticated. Figure 5.2 depicts the structure of such a partition. It contains the payload as the main part which is encrypted and authenticated using AES-HMAC and subsequently signed using RSA. For HMAC-based authentication and integrity checks, the HMAC key as well as the HMAC digest are appended to the payload before encryption. Xilinx uses the MAC-then-encrypt order of encryption and authentication, i.e. the message digest of each partition is first computed followed by its encryption. The key for the AES encryption can either be stored in the battery-backed random access memory (BBRAM) or in eFuses of the chip. The selection of the AES key source can be enforced by setting the corresponding eFuse. If the optional RSA algorithm is used for authentication, an RSA signature verification is computed in software and the signature as well as the certificate are appended to the partition. The hash of the RSA

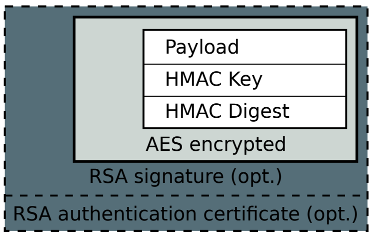


Figure 5.2: Content of an authenticated and encrypted boot partition

public key, which is used to validate the certificate, is stored in an on-chip eFuse array.

After power-up, the CPU executes the hardwired instructions from the internal BootROM which is a small and non-accessible read-only memory of 128 KB. It also initializes the clocks and configures the processor core along with the necessary peripherals to fetch the FSBL from NVM based on the boot mode stating the source where the FSBL can be fetched (i.e. SD card, NAND flash or NOR flash). The boot mode is determined by the voltage levels on the chip's external pins[5]. The BootROM code then copies the FSBL from the NVM to the 192 KB[6] on-chip memory (which is typically large enough). The FSBL code is decrypted and authenticated using the AES/HMAC core on the fly while copying it to the internal memory. Upon successful verification of the HMAC, control is handed off to the FSBL and it is executed from the same internal memory. The FSBL is a Xilinx-specific bootloader which initializes clocks, GPIOs, DDR controller and the FPGA fabric. Following the initializations, the FSBL loads subsequent partitions. Xilinx provides a template of the FSBL code, which can be customized. The FSBL controls the decryption and authentication of the bitstream and second stage bootloader. If a bitstream is part of the boot image, this is loaded next. (The bitstream may alternatively be loaded at a later stage of boot through the U-boot or the OS. This, however, is uncommon since it requires additional code to be inserted into the U-boot or later OS instead of using the Xilinx template. Also, hardware acceleration would not be available for the verification of the OS image.) The bitstream is decrypted and authenticated using AES/HMAC while it is loaded into the FPGA configuration memory. If the verification fails, the FPGA containing an unauthentic configuration is not activated. As a next step, the second stage bootloader, e.g. U-boot is decrypted and authenticated. As the on-chip memory is not large enough for

[5]Those pins are accessible to possible attackers but no unauthorized images can be started.

[6]The rest of the on-chip memory is reserved for the BootROM code until control is handed off to the FSBL

typical loaders, the decrypted U-boot is stored on the external DDR memory. If the verification is successful, control is then handed off to U-boot. After initialization of the platform (processor, clocks, memory) and reservation of memory, U-boot enters the main loop where it decrypts and authenticates the kernel image. U-boot then reads the kernel image header and jumps to the address of the kernel header, handing off control to the OS.

As can be devised, a secure chain-of-trust is established starting from the BootROM code. If any of the partitions cannot be successfully verified, the system goes into a secure lockdown mode. In case of a lockdown, the AES key in the BBRAM is cleared and the configuration memory of the FPGA is cleared (the keys and settings in the eFuses remain untouched).

5.2 Attacking the Secure Boot on FPGA SoCs

In this section, we outline the general idea of an attack on the secure boot process in the context of FPGA SoCs. The main underlying observation for all secure boot processes is that the verification of the authenticity and integrity of a software image is either not done in-place, or, more importantly, the process which performs the verification and later hands off control to the subsequent stage is inherently not atomic in the sense that it could be interrupted by manipulations. This is an issue that we generally want to highlight and which is of particular relevance for embedded systems built upon FPGA SoCs. In most cases of conventional SoCs, there is no reason to believe that a manipulation of the memory is happening while the CPU is executing the secure boot code, since no-one besides the CPU is accessing the memory. However, in the case of FPGA SoCs, hardware cores on the FPGA have access to the shared memory bus. Hence, such cores, once loaded, present as immediate additional actors on those buses and may manipulate memory content while the CPU is not 'aware' of this. Specifically, such hardware cores, once loaded, are able to manipulate the boot process such that a malicious software image is executed instead of an authentic one. This can be achieved by the hardware secretly overwriting parts of a running bootloader.

It is important to note that the malicious functionality in the hardware, for the reasons explained in Chapter 1 and Chapter 3, is part of an FPGA configuration file and contains authentic signatures. It is also important to note that it often makes sense to load the FPGA before starting the software system so that the hardware acceleration is available to verify large software images to reduce software startup times in secure boot scenarios.

5.3 Related Work

Previous contributions have already highlighted some of the issues arising from unwanted additional functionality in hardware blocks that leak or corrupt sensitive information. Kutzner et al. [KPS13] describe how an AES core could be maliciously modified such that the key of the last round is output instead of the cipher text. Other contributions have demonstrated how cryptographic keys could be leaked via intentional side-channels such as the power consumption [LKG+09] or over the wireless channel [JM10]. King et al. [KTC+08] and Yang et al. [YHD+16] show how a privilege escalation of applications can be achieved at run-time using malicious hardware blocks. Li et al. [LDP15] show how malicious functionality in a hardware core can be used to circumvent the password check of a Linux OS running on a Xilinx Zynq-7000. In Chapter 4, we show how public authentication keys can be overwritten by a malicious hardware core in FPGA SoCs so that the devices accepts malicious system updates.

In this chapter, we show that even the secure boot process, one of the most important and basic security features of embedded systems, can be compromised by malicious hardware blocks in the FPGA on FPGA SoCs. We describe a proof of concept on the Xilinx Zynq-7000 device and explain why even later models which include IOMMUs (and are fit to counteract attacks such as described by Li et al. [LDP15] and in Chapter 4) will likely be susceptible to such attacks. Our proof of concept system includes a conventional software stack along with an additional hardware block for the FPGA.

5.4 Breaking the Secure Boot on Xilinx Zynq-7000

In this section, we describe our proof of concept where we practically break the secure boot process of the Xilinx Zynq-7000 FPGA SoC using a hardware trojan.

5.4.1 Hardware Module

As a likely scenario for the proof of concept, we chose an interface for the hardware module similar to a high throughput accelerator which we connect to the AXI bus of the Xilinx Zynq-7000. We use an interface which is typical for high-speed hardware accelerators that consists of a low-speed slave interface for configuration and control (i.e. source address, destination address, length, and enable signal) as well as a high-speed master interface for data transfer [Bar15, Bar16, Ens13]. The master interface is DMA-like and, hence, reduces the load on the processing system by not requiring the CPU for data input/output. The CPU only passes the configuration information after which the module starts to perform its core function whilst accessing data directly from the memory.

The module also contains unwanted functionality. Using the high-speed data interface, the module progressively scans the external DDR memory and maliciously alters its contents. It specifically scans the U-boot binary in the DDR memory for the kernel boot parameters. After finding the target memory location, the boot parameters are modified to load an unauthorized kernel image from a remote server over the network instead of booting from the verified source. For this unwanted functionality, our example hardware module requires an additional 117 LUTs and 46 Slices. This hardware overhead will likely pass unnoticed since, e.g. a high-speed AES-GCM core for the same family of FPGAs and including a similar interface requires 22.7 k LUTs and 6.9 k Slices [Bar15], which is larger by orders of magnitude.

5.4.2 Sequence of Events During the Attack.

Figure 5.3 depicts the Xilinx Zynq-7000 FPGA SoC and highlights the data flow of the encrypted and decrypted images during the secure boot process as it is described in Section 5.1. The numbers in the figure indicate the sequence of events during the attacked boot process. In Step 1 (dotted grey path), the encrypted and authenticated partitions are read from NVM (our system boots from an SD-card) or DDR memory and piped through the AES/HMAC cores for verification[7]. Boot Steps 2-4 (dashed green paths) depict the successive loading of the FSBL, bitstream and U-boot respectively, as described in Section 5.1. Following the successful verification and loading of the bitstream to the FPGA, U-boot is verified and loaded to the external DDR memory and control is handed-over. As soon as the hardware is activated after the successful verification of the HMAC, the malicious core starts to scan the external DDR memory for a particular U-boot setting which is to be modified. This is the actual attack and depicted as Step 5 in Figure 5.3.

Figure 5.4a shows an excerpt of the trusted U-boot image which contains

[7]Encrypted partitions may first be copied from NVM to DDR in order to accelerate data transfer.

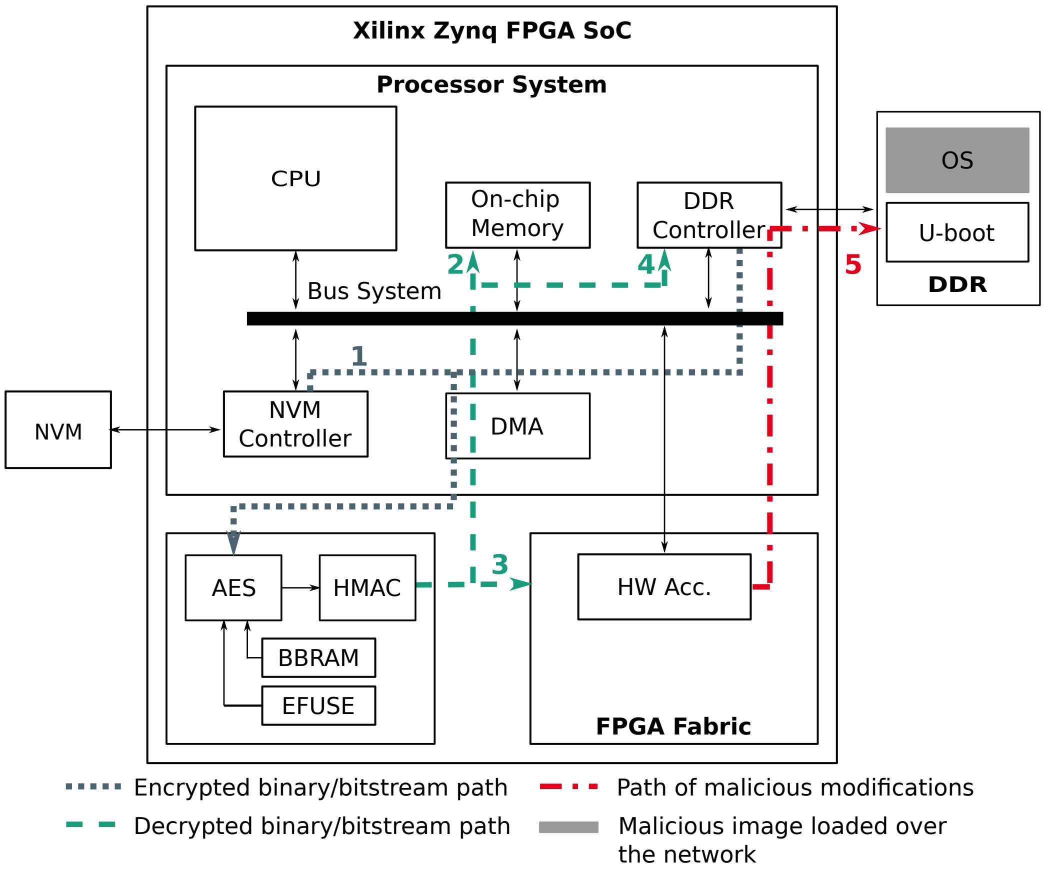


Figure 5.3: Secure boot attack on Xilinx Zynq-7000 FPGA SoC

strings representing the boot parameters in the .rodata section of the image. In Step 5 explained above, the hardware module searches for the SD card boot parameters, i.e. `fi.sdboot=` (see Figure 5.4a). Upon locating the string, the following original boot parameters from the authentic image which are used to load and verify the kernel image on the SD card are overwritten:

```
load mmc 0 <dest_addr> <filename>
zynqaes <src_addr> <src_len> <dest_addr> <dest_len>
```

The `load` command is used to transfer the encrypted and authenticated kernel image from the SD card to the DDR memory. The `zynqaes` command then routes this image from the DDR memory to the AES-HMAC core for decryption and authentication. The malicious functionality overwrites these commands with the following bootp command:

```
bootp <dest_addr> <sever_IP_addr>:<filename>
```

The `bootp` command downloads a file from the specified server IP address

```
fi.sdboot=if mmcinfo; then echo Boot
from SD env variables to RAM... && lo
ad mmc 0 ${devicetree_load_address} $
{devicetree_image} && zynqaes ${devic
etree_load_address} ${devicetree_size
}$ ${devicetree_dest_address} $device
tree_len$ &&load mmc 0 ${ramdisk_load
_address} ${ramdisk_image} && zynqaes
 ${ramdisk_load_address} $ramdisk_siz
e$ ${ramdisk_dest_address} $ramdisk_l
en$ &&load mmc 0 ${kernel_load_addres
s} ${kernel_image} && zynqaes ${kerne
l_load_address} $kernel_size$ ${kerne
l_dest_address} $kernel_len$ &&bootm
${kernel_load_address} ${ramdisk_load
_address} ${devicetree_load_address};
```

(a) Authentic image

```
fi.sdboot=if mmcinfo; then echo Boot
from SD env variables to RAM... && bo
otp 0x2000000 10.148.95.25:dt.dtb &&
bootp 0x4000000 10.148.95.25:ur.gz&&
bootp 0x2080000 10.148.95.25:uI &&boo
tm 0x4000000 0x2080000 0x2000000;vice
tree_len$ &&load mmc 0 ${ramdisk_load
_address} ${ramdisk_image} && zynqaes
 ${ramdisk_load_address} $ramdisk_siz
e$ ${ramdisk_dest_address} $ramdisk_l
en$ &&load mmc 0 ${kernel_load_addres
s} ${kernel_image} && zynqaes ${kerne
l_load_address} $kernel_size$ ${kerne
l_dest_address} $kernel_len$ &&bootm
${kernel_load_address} ${ramdisk_load
_address} ${devicetree_load_address};
```

(b) Corrupted image

Figure 5.4: Excerpt of the .rodata section of the U-boot image

to the DDR memory[8]. Once the image is downloaded to the DDR memory from the remote server, the kernel is booted. For this, the malicious core also writes a regular U-boot boot command to the U-boot, which specifies the locations where the `bootp` command had placed the kernel image, uramdisk and devicetree in the DDR memory, as follows:

```
bootm <kernel_addr> <ramdisk_addr> <devicetree_addr>
```

Figure 5.4b shows the same excerpt as Figure 5.4a, with the described malicious modifications highlighted in grey. (In our set-up, the kernel image (`uI`), devicetree (`dt.dtb`) and uramdisk (`ur.gz`) are three separate images and, hence, successively loaded.) Note that the U-boot code after the overwritten section of code is corrupted. This is highlighted in red in Figure 5.4b but does not make a difference as control is handed off to the OS after the `bootm` command.

Through the last command, control is handed-off and the malicious kernel image is booted without the system or secure boot process having any chance to detect the manipulation.

5.5 Results

Here we provide implementation results of our proof-of-concept. The investigation was carried on a Zedboard Rev. D development board with 512MB of

[8]Alternative U-boot commands that could be used to load a file from a remote server are `tftpboot` and `dhcp`

external DDR memory. The FSBL v2015.4, u-boot-xlnx v2016.1 and linux-xlnx v2016.1 from Xilinx are used [Xilb]. Each partition of the boot image is decrypted and authenticated using the on-chip AES/HMAC.

Regarding timing, the module is clocked at 100 MHz. Each read operation to the external memory takes 70 *ns*. Insertion of the malicious code takes 3.6 μs. Straight after activation, the core begins to scan the memory starting from the address `0x4000000`, which is the default address where the U-boot is stored. This address is static and publicly accessible from the default implementation of the FSBL provided by Xilinx or the U-boot code which is open-source. The overall attack (scanning and overwriting) takes 5.5 *ms*. Note that the scanning would even be faster if the hardware core would use a full memory mapped AXI interface or an AXI-stream interface which support burst data transfers.

To summarize, we were able to carry out the proof of concept attack successfully and have shown that secure boot, which is a critical protection mechanism for embedded systems, can be compromised using malicious hardware in the FPGA fabric of FPGA SoCs.

Note that for this proof of concept, the RSA authentication of images was not enabled and only the AES-HMAC was used for decryption and authentication. However, the same attack can be carried out when RSA is enabled without any modifications to the malicious functionality.

5.6 Discussions and Generalizations

There are several interesting aspects of the described successful attack which require a more detailed discussion. This helps understanding and highlights generalizations to other devices.

IOMMUs. In general, IOMMUs or SMMUs, are designed to protect the system against threats such as the one demonstrated in this contribution where bus peripherals access shared memory resources without proper authorization. IOMMUs are hardware cores which, when properly configured, control bus access rights of such peripherals. They are even available in newer (and partly more expensive) FPGA SoCs such as the Xilinx UltraScale+ [Xil17] and Intel Stratix 10 [Alt15]. However, IOMMUs are typically initialized by the OS, which is the last stage in the boot process. Hence, we conclude that the availability of IOMMUs on FPGA SoCs will not generally prevent the described attack because this would require an earlier configuration. We advise the use of the countermeasure presented in Chapter 6 instead.

ARM TrustZone. ARM TrustZone prevents unauthorized accesses from the normal world to secure world resources (e.g. memory regions) through the use of the TrustZone bit, which is implemented in all system parts (MMU, bus participants, CPU). However, it is likely that cryptographic cores are placed within TrustZone, which allows them unlimited access which may lead to an attack as described. Even if an IP core is not within TrustZone, it is still able to compromise boot, since U-boot is typically not running in TrustZone. So while TrustZone is an important security feature, it is not an effective countermeasure against this attack.

Static Access Restrictions on Xilinx Devices. The Xilinx Zynq-7000 allows to statically restrict the access of peripherals to the memory at design time. This means that under no operational circumstances, the hardware may access certain excluded memory regions. A similar feature is also offered by Microsemi for the SmartFusion2 FPGA SoCs [Mic15]. However, this would pose a drastic limitation for designs since for most cases, the final use of a hardware module such as a cryptographic accelerator will be determined by software and it will be beneficial if all memory regions can be accessed. For example, a cryptographic core that is used for run-time integrity checking of software requires access to all memory regions.

On the Xilinx Zynq UltraScale+ series, a proprietary module known as XMPU module can be used to restrict the access of masters on the bus. This access configuration can optionally be locked at boot time (recommended method by Xilinx [Xil17]). This, however, means that the settings can only be changed after a power-on-reset using a modified and signed image. Alternatively, if this setting is not locked at boot time, the configuration can be changed at run-time.

Generalization. We used the Xilinx Zynq-7000 for a proof of concept. It is currently being deployed and will likely stay in the field for many years. And even though the Zynq Ultrascale+ includes IOMMUs and XMPU, the Zynq-7000 will remain attractive for new designs due to lower costs. Also, as described above, the availability of IOMMUs does not by default prevent the described threats. They also need to be configured properly before the FPGA is loaded.

The order of the boot of FPGA and software system influences the vulnerability of the boot process. In case of FPGA SoCs from Intel, three cases of boot [Alt16] are available: (i) the CPU boots and configures the FPGA during its boot sequence (similar to Zynq-7000), (ii) the FPGA is configured first and the CPU boot sequence is controlled by the FPGA, and (iii) the

FPGA and CPU boot independently. The first case is the same (as in Xilinx) with the same issue, the second is even worse as the FPGA is configured first. In the last, the FPGA and CPU are booted independently so the success will depend on whether the FPGA boots before the OS or during the CPU. In all cases, devices like, e.g. Stratix V and Arria 10 have no IOMMU and are vulnerable at run-time at least.

Microsemi on the contrary offers non-volatile FPGA SoCs, which means that the FPGA is ready to be used directly after power-up and there is no configuration of the FPGA from external memory as is the case with standard SRAM-based FPGA SoCs from Xilinx and Intel. Hence, the threat is imminent from the very start of the system.

In general, whenever the FPGA is configured early during the boot sequence, and this is often the case, a secure boot process can be compromised by a malicious core in the FPGA. In these scenarios the use of the countermeasure described in the Chapter 6 is advised.

Virtual vs. Physical Memory Addressing. There is no virtual addressing before the OS is loaded. Hence, attacks such as the one presented do not have to take address mappings into account and can rely on direct physical addressing instead.

Finding the Location of the Code to be Overwritten. One of the key factors for attack vectors as the one described in this contribution is to estimate the location of the code which is to be overwritten. The less precise this is, the more code needs to be searched which takes more time. Interestingly, the location of the respective U-boot image depends only on a few factors and can be determined using publicly accessible information. Within the U-boot image, the presented attack overwrites U-boot environment variables which are located in the `.rodata` section of the image in the form of strings. By being part of the U-boot image, the variables are authenticated. (Technically, there are cases where such variables are instead retrieved from other, unauthentic external memories, and are loaded onto the heap in RAM at run-time. However, this option is completely unreasonable in the context of secure embedded systems since attackers with physical access may easily modify them [Jeo14]. Hence, we assume that U-boot is compiled such that it does not read environment variables from external memory.)

U-boot is generally unaware where the previous bootloader, in our case the FSBL, was loaded. Hence, after the basic initialization, U-boot checks the current value of the program counter to determine the location. By default, the FSBL loads the U-boot binary to start at address `0x4000000`.

In general there are regions in DDR memory, which need to be preserved for instance to store the kernel image, devicetree and uramdisk. Hence, if U-boot detects that the previous loader has put it into those regions, it relocates itself to a predefined region before continuing execution. The relocation offset is usually at the end of the RAM so that one big continuous part of the memory remains for the OS. By analysing the U-boot code (which is open-source), the offset address can be retrieved[9].

Timing and Durations. The available time to perform a search for the specific string to be overwritten depends on whether it is done before or after the relocation, as described above. In our practical investigation, we found that the initialization before the relocation takes approximately 37 ms. Afterwards, the remaining part of the initialization is done, which takes approximately 400 ms on our example setup. Finally the kernel is loaded as a last step. From this we see that an attack could target the initial location of the U-boot during the 37 ms until it is relocated, or the final location during the 400 ms of further initialization.

Our presented hardware module is able to scan up to 2.1 MB of memory, and successfully modify the specific boot parameters during the shorter time of 37 ms before relocation. For comparison, the size of a standard U-boot is about 3 MB which already hints at the high likeliness of success. Our practical investigations have indeed shown, that for both cases, the attack could be performed successfully.

Caches. The CPU caches are enabled by U-boot and could possibly influence the outcome of such attacks if the respective code to be overwritten is cached while it is overwritten in the RAM. This would lead to the case that the original data is possibly written back from cache in case of a later cache eviction. However, we did not encounter such situations and suspect that the targeted part of the U-boot image, the boot environment variables, are not accessed, thus, not cached until they are used shortly before handing over control to the OS kernel.

5.7 Summary

The feasibility and practical impact of attacks on the secure boot process of FPGA SoCs through hardware on the FPGA was successfully demonstrated. In a time where services such as the Amazon AWS EC2 F1 instances and

[9]The U-boot command `bdinfo` outputs the relocation offset on a running system.

embedded systems based on FPGA SoCs entice the broader use of hardware from IP vendors, the trust level of such outsourced hardware is difficult to determine. In our opinion, hardware cores including unwanted functionality such as the one described here, could become more common in hardware IP marketplaces.

We think that this will have a high impact on designers of secure embedded systems. They will learn that there is 1. need to change the boot order (CPU first if possible), 2. need to correctly configure IOMMU before configuring the FPGA (typically configured after boot by OS Kernel), or 3. use the security wrapper proposed in Chapter 6 to prevent attacks from hardware peripherals in (FPGA-)SoCs.

Chapter 6

Security Enhanced AXI-Wrapper

The growing trend of outsourcing the development of hardware cores in comparison to in-house development is creating new challenges to the system security. As designers lose control over their design process, it is getting harder to ensure that outsourced hardware cores do not contain any additional functionality than those specified. Through the course of this thesis we have shown the effects of malicious hardware on the system security. Chapter 4 and Chapter 5 highlight the effects of malicious hardware on critical security mechanisms.

One general way to protect systems integrating third party hardware cores is to conduct a thorough code review of the hardware cores and check the coverage of the testbenches supplied with hardware cores before integrating them into a system. Only thoroughly tested cores should be integrated into the system and thus prevent the whole system from being compromised by untrusted hardware cores. However, lack of time can lead to small and stealthy malicious functionality being overlooked. In many cases, it will not be realistic for general design teams to acquire the necessary hardware expertise to prevent this. Hence, we developed an efficient countermeasure, which provides full re-usability, and is easy-to-review because of the small code size, that protects against all unauthorized memory accesses through hardware cores while raising an alarm at every attempt. It is a simple hardware wrapper for the AXI bus interface which is easy to integrate with all outsourced hardware cores with memory access and is configured through software. It can be seen as a stripped-down IOMMU which instead works straight from configuration without requiring the software to explicitly enable or configure it and has a smaller set of functionality, thus, trusted code base.

We begin this chapter with an introduction to the AXI4 bus protocol in Section 6.1. This is followed by our proposed design of the security enhanced AXI-wrapper in Section 6.2. We then compare our work to existing solutions in Section 6.3. Section 6.4 provides a description of the practical implementation and results of our solution. Finally, the chapter is summarized in Section 6.6.

This chapter was part of the following publication: *How to break secure boot on FPGA SoCs through malicious hardware. In Cryptographic Hardware and Embedded Systems - CHES 2017* [JHZ+17].

6.1 Introduction to AXI4 Bus Protocol

This section introduces the key concepts of the AXI protocol. AXI4 is part of the ARM AMBA protocol which is a commonly used bus system for micro-controllers, FPGAs and SoCs. The AXI4 protocol is the state-of-art version of the ARM AMBA protocol and was released in 2010 [ARM11]. AXI4 protocol offers 3 types of interfaces:

- AXI4-Lite: Simple interface with no burst transfers.
- AXI4: Standard interface which allows burst transfers of data up to 256 data bytes. Typically used for high performance memory mapped cores.
- AXI4-Stream: Data only interface with unlimited burst sizes, typically used for high speed streaming.

The three types of interfaces provide the flexibility to choose an interface to best suit the application. At the highest level, these three interfaces are similar and essentially consist of five independent channels.

Read Address Channel

The read address channel controls the addresses for any read access and is used to initiate a read transaction. The transaction is initiated by the master which is typically the CPU but in the case of FPGA SoCs, the transaction could also be triggered by a master core in the FPGA fabric. Figure 6.1 shows the channels for the AXI address and data read transaction. A read access begins with the address ready from the master, following which the slave sends an address valid acknowledgement. Upon acknowledgment, the read address is sent to slave. This handshake is the same for all the interface types.

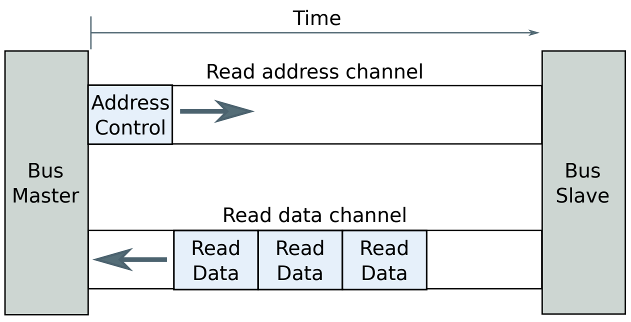


Figure 6.1: AXI read address and data channels

Read Data Channel

The read data channel is used for the actual read data transfer from slave to the master. AXI supports 32-bit and 64-bit data transfers. In case of burst transfers, i.e., full AXI and stream, several bursts of data are read consecutively as shown in Figure 6.1. While in the case of the lite interface, only a single burst of data is read.

Write Address Channel

The write address channel similar to the read address channel initiates a write transaction. It operates similarly for all the interface types. Figure 6.2 show the AXI write address and data channels for a read transaction. Transactions begin with a write request from the master which is then followed by an acknowledgment from the slave. Subsequently the write address is sent to the slave as shown in the upper channel of Figure 6.2.

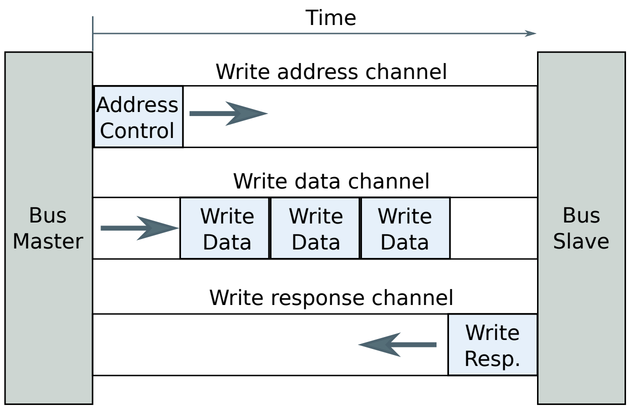


Figure 6.2: AXI write address, data and response channels

Write Data Channel

After the write address channel initializes the write transaction, the master can begin to write the corresponding data to the slave. Here, again in case of burst transfers, several bursts of data are sent successively in bursts of 32 or 64 bits as can be seen in Figure 6.2. For lite transactions only a single burst of data is transferred.

Write Response Channel

Write transactions have an additional channel for the response. The response channel is used to the indicate the end of a write transaction. Subsequently, the slave sends a write valid signal to the master indicating the success of the write operation to the master.

6.2 Concept for Security Enhanced AXI-wrapper

We propose a general countermeasure to protect systems against unauthorized memory access from hardware cores within the FPGA fabric of FPGA SoCs. For this purpose, we developed a flexible and lightweight security-enhanced hardware wrapper for cores with an AXI interface [10]. The goal is to isolate hardware cores with access to memory, such that even if malicious functionality is present, it is not able to propagate and corrupt other parts of the system.

Cores can be easily integrated into the wrapper and subsequently connected to the AXI bus. The wrapper stores access commands and prevents the core from accessing other memory regions than designated by software driving it through the command interface. In principle it can be regarded as a stripped-down IOMMU. However, functionality is restricted to a minimum to support easy review, and a small trusted code base for easy re-use. Also, the wrapper is working with restricted default settings from the very start of the FPGA and does not rely on the OS configuring it (contrary to IOMMUs).

Typically, a software process using a hardware core writes configuration information (source address, destination address, length, enable) to the core via the core's slave interface. Based on this information the core performs its function using the master interface for memory access. Figure 6.3 depicts a stripped-down system architecture of an FPGA SoC including a core which

[10]The source code for the wrapping module can be retrieved from https://github.com/jacobnis/axi-firewall

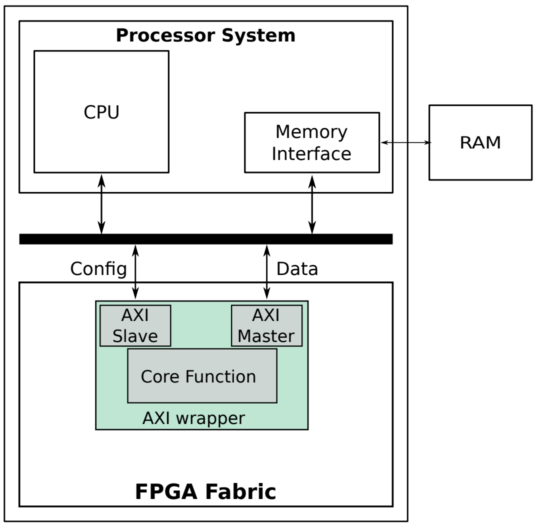


Figure 6.3: Hardware core with security-enhanced wrapper

is wrapped with our proposed design. Our wrapper uses the received configuration information to monitor the AXI-transactions of the master interface and only allows access to the memory range specified by the software process while leveraging the time during the AXI handshake to enforce the access commands.

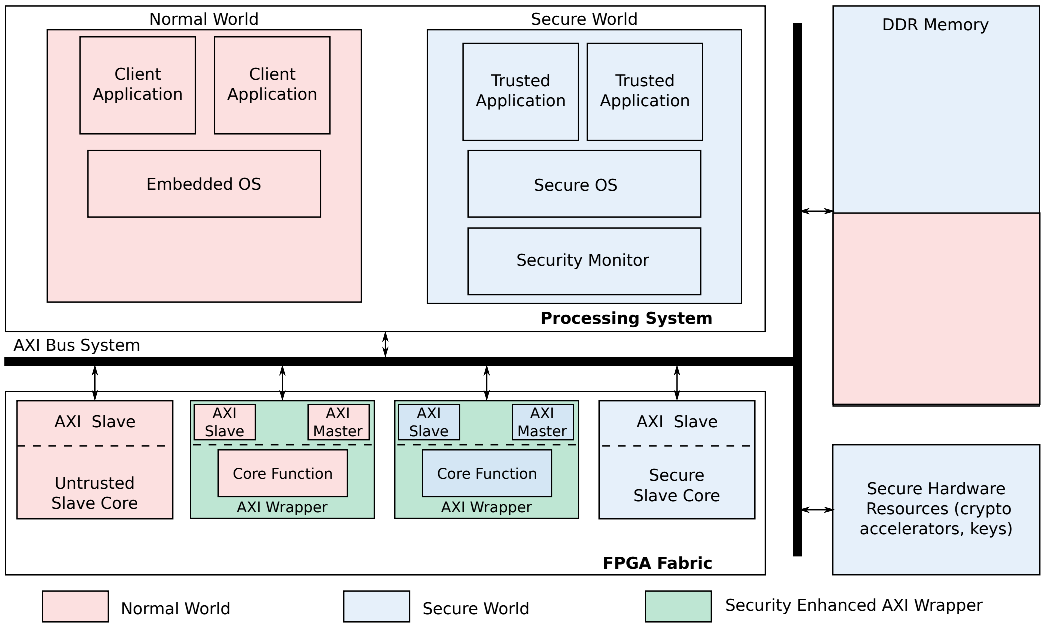


Figure 6.4: Top-level view of system with different isolation mechanisms

Figure 6.4 presents the top-level system overview of an FPGA SoC together with the various third party hardware cores in the FPGA fabric and their corresponding isolation mechanism. The system can be divided into secure (depicted in blue) and normal world (depicted in red) using ARM Trustzone. All resources of the FPGA SoC i.e., the DDR memory, the built-in accelerators as well as the hardware core in the FPGA fabric can be divided into secure or normal as shown in Figure 6.4. At power-up all the resources of FPGA SoC can either by set as secure world entities or normal world entities by the bootloader. In addition to this, the cores with direct access to memory can be isolated using the presented security enhanced AXI-wrapper (depicted in green). For example a simple untrusted third party slave hardware core can be placed outside the trustzone as indicated by the red core, such that it only has access to the normal world and is under the control of the processing system. While the cores with direct access to memory are isolated using our security-enhanced AXI wrapper to ensure that the cores only have access to the resources as configured by the corresponding software.

6.3 Related Work

In a previous contribution, Brunel et al. [BPOD14] have developed a secure AXI bridge which is similar to a full IOMMU core for SoCs. The downside in our view is that it requires significantly more hardware resources and contains a significantly larger amount of source code to review. Coburn et al., [CRRC05] and Cotret et al. [CDG$^+$12] present a post-boot run-time protection core similar to TrustZone. This is achieved by storing system wide or processor specific (for MPSoCs) security policies in large LUTs or BRAMs resulting in a significant overhead in area and latency. In contrast, the proposed wrapper protects devices against malicious hardware IP cores. Further, the wrapper reuses the configuration information passed to it from the firmware and leverages the time between the AXI handshaking for the enforcement. Hence minimizing the overhead in terms of area, performance, latency and maintenance of the security policies.

Xilinx provides a module known as the XMPU which can be used to dynamically restrict memory access from early boot stages onwards [Xil17] but is only available for the Zynq Ultrascale+ devices. Unfortunately, the sources of the module are not publicly available. In contrast, our AXI-wrapper can be used for any hardware cores with memory access and is not restricted to any manufacturer or device. Also, as the source code of the wrapper is small in size and public, it can be easily reviewed and re-used.

6.4 Practical Implementation

In this section, the implementation details of the security enhanced AXI-wrapper is provided. For this we recall the AXI bus protocol as described in Section 6.1. In a typical AXI transaction, the address channel is first set followed by the data channel. A transaction begins with the master sending the address of the memory location to be read/written along with an address valid signal. It then waits until the slave responds with an address ready signal. Next, the slave sends a data valid signal and the master responds with a data ready signal following which valid data can then be read from/written to the memory.

The security enhanced wrapper uses the time between the handshaking to check and enforce the access privileges of the core. The wrapper checks the address issued by the master while it waits for the slave's address ready response signal. If the wrapped underlying core attempts to access memory outside of the allowed range, an alarm signal is set. Thereby aborting a transaction before any valid data could be read from/written to the memory while not affecting the performance of legitimate transactions. In addition to the address bounds check, the wrapper also checks the length of data read/written as per the current configuration. If the core attempts to read or write more data, the remaining transactions are dropped and an alarm is raised.

Another critical task of the wrapper is to ensure that the core is only functional when a software process has explicitly set the enable signal. This is done is order to prevent the core from performing illegal memory accesses while actually in idle state. If the core tries to initiate any unauthorized transactions, an alarm is raised. Thus the core cannot enable itself or modify its configuration settings.

Table 6.1: Wrapper error codes

Error Code	Description
00	No error
01	Exceeded permitted number of transactions
10	Out-of-range write
11	Out-of-range read

The alarm signal can be connected to the interrupt controller or a separate tamper detection unit. Currently, all subsequent transactions are blocked after an alarm is raised. However, based on the criticality of the system, other actions can be taken such as, putting the system into a secure lockdown mode. The wrapper also includes a 2-bit error output code to indicate the cause of the alarm which is listed in Table 6.1.

6.5 Results

Our proposed wrapper has a hardware overhead of 133 LUTs and 55 Slices. There is no cycle count penalty during operation. As an example, a high throughput AES core with an interface to directly access memory requires 22 k LUTs, 6.9 k Slices and 72 BRAMs [Bar15]. If we think of integrating our proposed security enhancements to this core, it would add an area overhead of 0.6% LUTs and 0.7% Slices which is a low overhead compared to the given security gain in our opinion.

6.6 Summary

At a time when more and more stores for hardware cores are popping up, the trust in outsourced hardware cores is a growing concern. One way to protect the system is a thorough source code review of third party IP. This, however, is time consuming and requires a lot of resources. Hence, we presented a simple AXI-wrapper to prevent hardware cores with access to memory from maliciously accessing and modifying unauthorized sections. Alternatively, an early configuration of IOMMUs would be necessary or when available the usage of features like the Xilinx XMPU.

Chapter 7

Secure Configuration of Hardware on FPGA SoCs

System-on-Chips which include FPGAs are important platforms for critical applications since they provide significant software performance through multi-core CPUs as well as high versatility through the integrated FPGA fabric. The integrated FPGA fabric allows to update the programmable hardware functionality, e.g. to include new communication interfaces or to update cryptographic accelerators during the life-time of devices. This makes FPGA SoC an attractive platform for high performance edges devices. Edge computing refers to the transition from conventional centralized cloud-based computing nodes to computing nodes directly attached to the sensors and actuators in the field. Updating software as well as hardware configuration is required for critical applications such as e.g. industrial control devices or vehicles with long life-times. Such updates must be authenticated and possibly encrypted. One way to achieve this is to rely on the FPGA manufacturer-provided cryptographic modules and respective master keys. However, past research has shown several of these manufacturer-provided cryptographic modules vulnerable to side-channel attacks. Thus leaving already deployed devices vulnerable to side-channel attacks with no mechanism to update them. In this contribution, we show how to retrofit Xilinx Zynq FPGA SoCs with an alternative side-channel resistant cryptographic accelerator and a PUF (as no manufacturer-provided secure key storage is available from within the FPGA fabric) to update the user hardware modules in the FPGA fabric using dynamic partial reconfiguration. These key aspects reduce the required trust in manufacturer-provided security features, only the public key for the verification of the initial bootloader and the static bitstream is required.

This chapter begins with a description of the vulnerabilities to design security in FPGAs and FPGA SoCs in Section 7.1 and related work in Section 7.2. Following which partial reconfiguration will be described in Section 7.3. The secure configuration process will be described in Section 7.4 with the practical implementation in Section 7.5 and corresponding results in Section 7.6. In Section 7.7, a security analysis of the whole system is provided and the generalization of the proposed scheme to other platforms is discussed in Section 7.8. Finally a summary is provided in Section 7.9.

Parts of this chapter were part of the following publications: *Securing FPGA SoC configurations independent of their manufacturers. In 30th IEEE International System-on-Chip Conference, SOCC 2017* [JWH+17] and *SCA Secure and Updatable Crypto Engines for FPGA SoC Bitstream Decryption. In 3rd Workshop on Attacks and Solutions in Hardware Security, ASHES 2019* [UJH+19].

7.1 Vulnerabilities to Design Security

Powerful SoCs which include configurable hardware in the form of FPGAs on the same chip are increasingly popular platforms for embedded systems because they offer high performance and high flexibility in software as well as hardware. This makes them suitable for high performance edge computations where vast troves of data need to be processed with low latency, e.g. object recognition for autonomous cars and speech recognition. Typically these applications send their data to a cloud-based backend for processing which increases the latency. With FPGA SoCs such applications could move away from cloud-based processing to data processing directly at the edge node. Embedded systems, which are deployed in the field for a long life-time, require updates from a functional as well as from a security perspective. Also, given the nature of the applications, physical access to the device must be expected. This in addition to the fact that SRAM-based FPGAs need to configured from an external NVM, over a possibly insecure channel, at every power-up raises the need to protect the bitstream against manipulations inclusions as discussed in Chapter 3. Earlier the FPGA industry relied on security through obscurity, in that they use different bitstream formats for each FPGA family. We have seen that having proprietary configuration files does not protect the contained IP against reverse engineering and should not be the only security mechanism [Skr13, TZ13, NR08].

To overcome this, vendors now include bitstream encryption and authentication using state-of-art cryptographic modules. For FPGA devices in particular and also for more general SoCs, a typical method to achieve this is to use chip-manufacturer-provided master-keys or individualized keys in manufacturer-provided key storage as well as built-in cryptographic engines. There have been security issues, however, with manufacturer-provided features. For example, encryption engines included in FPGAs for the decryption and authentication of configurations have been shown to be vulnerable to side-channel analysis, where attackers use measurements of power consumption during cryptographic operations to extract and misuse keys. Successful attacks have been reported for manufacturers of FPGA SoC platforms. First reports hit devices such as the Xilinx Virtex-II Pro [MBKP11] where full triple DES keys for bitstream encryption have been extracted. Current devices such as the Xilinx 7 series have also been targeted successfully [MS16]. The same issues have been reported for Intel, the second important manufacturer of FPGA-SoCs, where Stratix II and Stratix III devices have been broken in a similar way [MOPS13, SMOP15]. Finally, also Microsemi's ProASIC3 devices have been broken [SW12b, SW12a]. These weaknesses can be countered by not solely relying on manufacturer-provided security features for core operations such as the authentication and decryption of configurations.

We present a practical realization of a system design that allows to retrofit FPGA SoCs with alternative side-channel resistant cryptographic accelerators for the authentication and decryption of hardware configurations to support secure boot processes. Since manufacturer-provided secure key storage is typically not accessible from such custom designed hardware, and any form of master key is ill-advised, we also implemented a PUF which is used as device-individual credential store. PUFs generate unique-per-device keys by exploiting smallest manufacturing variations in ICs even though the corresponding design is stable. These keys are generated only on request as soon as the design is configured into the FPGA and are, thus, not hard-coded in the bitstream or binary. However, it is not straightforward to use alternative cryptographic cores instead of built-in cores for bitstream decryption. Thus we use dynamic partial reconfiguration, which is a feature supported by all leading FPGA vendors, together with the above mentioned custom cores to securely load user hardware modules to the FPGA fabric of FPGA SoCs. As our contribution, we describe an implementation of this system design on the popular Xilinx Zynq-7000 FPGA SoC but we also discuss the portability of our design to other FPGA SoCs. The side-channel resistant decryption core used in this work is based on the principle of leakage resiliency, while for the key storage we use an identity-generator type of PUF.

In summary, this system design allows for more flexibility through the integration of custom cryptographic accelerators and storing keys in PUF modules to bind sensitive hardware cores to a specific device. Another advantage with this architecture is the possibility to update the custom cryptographic cores to meet new standards or fix bugs, which is not possible with the built-in cryptographic modules. Further with this system design we are not reliant on any manufacturer-provided closed-source symmetric cryptographic core and its corresponding secret key storage, thus also increasing the trust in the design. We, however, are still reliant on the manufacturer-provided signature verification, to verify the authenticity of the bootloader and static bitstream consisting of the side-channel resistant decryption core, the PUF and the partial reconfiguration (PR) controller. This is necessary to ensure that an attacker did not modify the static bitstream and bootloader, e.g. to insert additional debug lines to read-out the key. For the signature verification, only a hash of the public key is stored on the device which is not sufficient for an attacker to manipulate the signature. The adversary would need to have access to the private key in order to tamper with the signature, which is not stored on the device.

7.2 Related Work

The general idea behind using alternative cryptographic cores instead of the built-in ones for decryption and verification of hardware cores was outlined in a white paper by Xilinx [Pet15]. They describe a scheme to load user hardware cores using partial reconfiguration, an AES-GCM for the decryption and authentication and a PUF for the key storage. However, no working implementation of this scheme was provided and thereby also leaving out a lot of technical challenges, like buffering of the bitstream until verification. Their work also lacks an analysis of the overall system security. It was through our initial work [JWH+17], that a basic implementation of the scheme was provided where we used an AES-GCM core for decryption and authentication together with a twisted bistable ring PUF for the key storage and dynamic PR to securely load user hardware cores. Later Owen et al. [JHC+18] also presented a similar concept, they use a PUF that is sensitive to path delays, thus any modifications to the bitstream would result in the PUF response being different and thereby a different key. This way they achieve a self-authenticating design and are not reliant on any vendor-provided authentication schemes. However, they also perform the encryption on-chip using a separate bitstream, which means that using their scheme the encryption would need to done in a secure environment and thus hindering the possibility of field updates.

We instead use the PUF to bind an external secret key and perform only the decryption on-chip which allows updates to be encrypted offline. Furthermore we also extend our initial work [JWH+17] to include side-channel countermeasures to the decryption core and also include a buffer to store the bitstreams following decryption until the authenticity can be verified.

7.3 Introduction to Partial Reconfiguration

Partial reconfiguration (PR) is a unique feature available in most modern FPGA devices which allows updating parts of the FPGA while not affecting other regions in the FPGA. This requires dividing an FPGA design into two parts: *static and reconfigurable logic* as shown in Figure 7.1. The static logic consists of IP cores whose functionality does not need to be updated regularly. While the reconfigurable module can be updated with different features by switching from one to another reconfigurable module, thereby time sharing the FPGA resources. The advantage with time-sharing of the

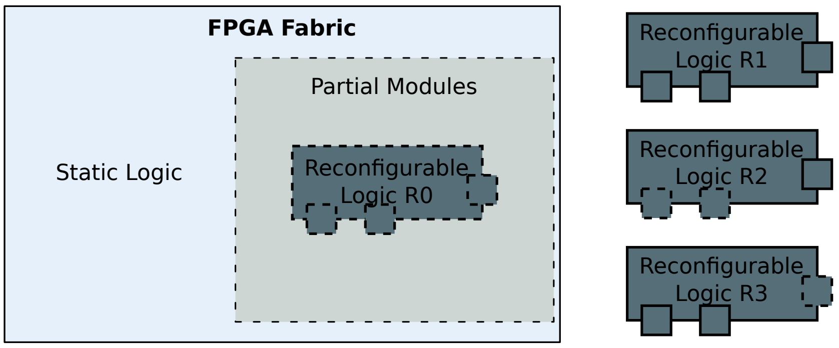


Figure 7.1: Partial reconfiguration overview

resources is that more functionality can be included at no additional area overhead. Further, PR provides the flexibility of updating systems with new accelerators so as to meet new standards. With PR, the reconfiguration time can be significantly reduced by only reconfiguring certain regions of the FPGA, instead of the whole FPGA.

Similar to the standard FPGA development (see Section 2.3.6), a full bitstream is generated which consists of all the static logic plus the blank configuration for the reconfigurable modules. This is often referred to as the *static bitstream*. In addition to the full bitstream, several smaller bitstreams, commonly known as *partial bitstreams*, are developed for each reconfigurable module, e.g. in Figure 7.1 this would be the configuration bitstreams for the modules R1 to R3. The partial bitstreams can be used to update the functionality of individual reconfigurable logic blocks while the full configuration bitstream can be used to reconfigure the whole FPGA including the static logic.

There are two types of PR: *passive/static* and *active/dynamic*. In static PR, mostly seen in older FPGAs, the FPGA is halted during the reconfiguration process. Following, the reconfiguration of the partial modules, the current working state is restored and the processing continues as usual. While, in dynamic PR, the current standard for PR, only the partial module being reconfigured is affected. The rest of the FPGA, static and other reconfigurable modules, can continue functioning without interruption.

7.4 Concept for Secure Update of FPGA SoCs

This section provides a detailed description of how custom cryptographic cores on an FPGA SoC can be used to securely load user hardware cores instead of the manufacturer-provided built-in decryption core and its corresponding key storage. We provide a reference design for the Xilinx Zynq-7000 FPGA SoC.

Figure 7.2 shows the FPGA SoC which includes an FPGA fabric and a multi-core CPU with the modules used for the security-enhanced design depicted in orange and user hardware cores depicted in green. The main modules for the security enhancement are the PUF, hardened AES core and the PR controller, all of which are integrated in the FPGA fabric of the FPGA SoC as part of the static bitstream. All user cores are loaded using dynamic PR following their decryption and authentication using custom cryptographic cores loaded in the static bitstream.

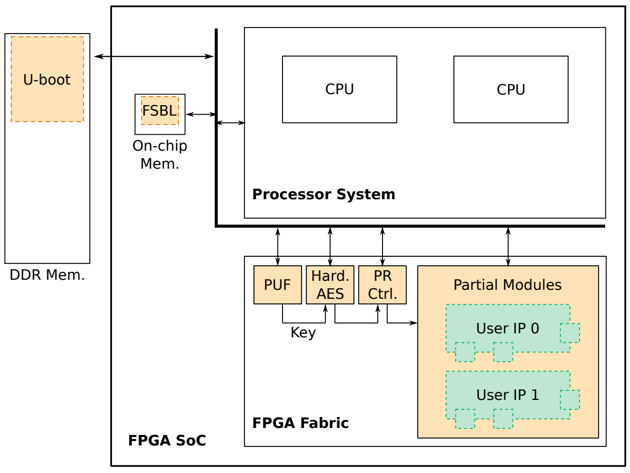


Figure 7.2: System overview of secure configuration of FPGA SoCs

Before deployment, the fuzzy commitment scheme is used to uniquely bind a key to the device-specific bit sequence, i.e. to a certain FPGA and outputs an expanded and masked bitstring, known as helper data. This helper data, corrects errors in the noisy PUF bit sequence and reproduces the key later on demand. Neither the user-generated secret key nor the PUF bit string is hard-coded in the bitstream, the secret key is reproduced at run-time using the helper data, which is publicly stored. In addition to enrolling user-generated secret key and generating the corresponding helper data, prior to deployment the hash of the RSA public key is burnt into the eFuses to enforce the authentication of the static bitstream to be loaded to the FPGA fabric.

Figure 7.3 depicts a regular boot sequence on FPGA SoCs with modifications in order to support the enhancements. The boot process begins after power-up and is divided into the following:

1. BootROM - non-accessible internal hardwired code
2. First stage bootloader (FSBL)- User code used to initialize the system and load subsequent partitions, i.e bitstream and second stage boatloader
3. FPGA configuration - *This is where the FPGA is configured with the previously listed static cryptographic modules*
4. Second stage bootloader - User code, e.g. U-boot, typically used to load the operating system
5. Partial bitstream - User IP that needs to be securely loaded into the FPGA
6. User software - End-user application which can either be a bare-metal application or running on top of an OS

Starting from the FSBL, all subsequent stages could be encrypted and authenticated using solely manufacturer-provided features as described in Section 5.1. However, the goal here is to bind the user IP to the device using the hardened AES core and PUF-based key storage instead of using manufacturer-provided features.

In this work, the FSBL, static bitstream and second stage bootloader, U-boot, are authenticated using the manufacturer-provided RSA while the partial bitstreams consisting of the user IPs are protected using the hardened AES core. Although no sensitive information is hard-coded in the configuration, the integrity of the bootloader and initial FPGA configuration needs

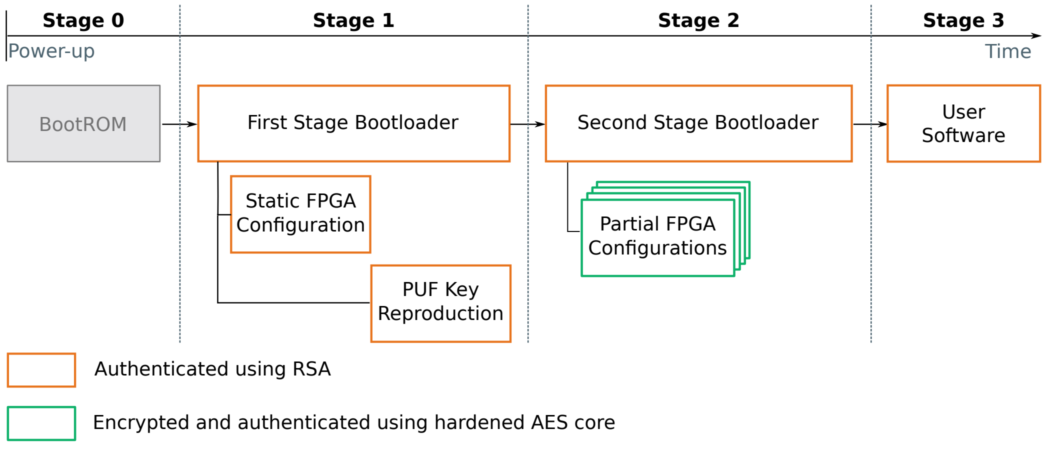


Figure 7.3: Boot flow of the hardened boot process

to be verified before they are loaded onto the device. This ensures that an attacker has not manipulated the initial configuration in a way that could leak the device-intrinsic key after its generation. For this we are reliant on the manufacturer-provided authentication scheme such as RSA on the Xilinx Zynq platform. This is the only stage where we rely on manufacturer-provided security functions. The signature verification only uses the public key of the RSA key pair, a hash of which was burnt to the device eFuses prior to deployment. The private key is neither stored on the device or nor is it processed during the signature verification. Hence there is no need to harden this process and we consider it to be no risk to rely on the manufacturer-provided scheme for this.

As shown in Figure 7.3 after power-up the boot process begins with the BootROM code which first verifies the RSA signature of the FSBL, before loading it to the on-chip memory and handing of the control to it. The FSBL in turn authenticates and loads the static configuration to the FPGA fabric. The static bitstream comprises of the side-channel hardened AES core, PUF, PR controller and the blank configurations of the user IP to be loaded. Following the configuration of the FPGA, the FSBL is modified to send the previously generated helper data to the PUF and trigger the PUF to reproduce the encryption key. Once the key has been successfully reproduced, it is directly transferred to the hardened AES core, thus the key never leaves the FPGA and is not accessible through any shared resources of the FPGA SoC as shown in Figure 7.2. The PUF is then locked until the next power-on-reset. The boot process continues with the FSBL authenticating and loading the second stage bootloader, U-boot. As the on-chip memory is only several kilobytes large, it is not large enough for the U-boot and as result U-boot is loaded to the external DDR memory.

Here, we use U-boot to parse the encrypted partial bitstream consisting of the user hardware cores to the hardened AES for decryption and authentication following the initialization of the system. Once the partial bitstreams have been successfully decrypted and verified, it is transferred to the PR controller which dynamically reconfigures parts of the FPGA while the other regions run uninterrupted. Finally the software stack is loaded to the CPU which in our case is only authenticated as shown in Figure 7.3.

The boot sequence presented in this work is similar to the standard boot process supported by Xilinx as presented in Section 5.1. Looking back at Figure 7.3 instead of loading the full bitstream following its decryption using the built-in decryption core, only the static bitstream consisting of hardened AES core, the PUF and the PR controller is loaded during Stage 1. Deviating from the typical boot process, all user cores are loaded using dynamic partial reconfiguration after verification using the previously loaded custom cryptographic cores during Stage 2 instead of the Stage 1.

If over the lifetime of the device, the user hardware cores require updates, these updates can be encrypted offline using the encryption key and remotely transferred to the device. Should the hardened AES core require updates during its lifetime, e.g. to fix bugs or update to new standards, even this can be done remotely by signing a new revision with the RSA private key and replacing the old one. However, any updates to the PUF would need to be done is secure environment as this would also result in a renewed key enrollment process. Due to the nature of PUFs, any changes to design could result in a different bit sequence, resulting in a different encryption key being reproduced with the existing helper data.

7.5 Practical Implementation of Alternative and Updatable Configuration of FPGA SoCs

In this section, a detailed description of the prototype implementation for the Xilinx Zynq-7000 FPGA SoC is provided. The design and development of the PUF and side-channel hardened AES core are beyond the scope of this work, existing cores are reused for building this prototype. For the PUF, the PicoPUF from Gu et al. [GO15] is directly reused. While for the hardened cryptographic core, the side-channel hardened AES core from Unterstein et al. [UHS$^+$18] is used.

7.5.1 Device Intrinsic Secret Key Storage

PUFs generate unique-per-device keys by exploiting smallest manufacturing variations in integrated circuits even though the corresponding design is stable. These keys are generated only on request as soon as the design is configured into the FPGA and are, thus, not hard-coded in the bitstream or binary. Generally speaking, there are two classes of PUFs: (i) strong or challenge-response type of PUFs that have a unique response for a given input challenge and (ii) weak or identity type PUFs that only have a few or just a single identify string for a device. The PUF used in this work is an identity-generator type PUF, meaning that it does not require an input challenge and is not prone to modeling attacks. A cross coupled feedback loop contained within a single slice of an FPGA is used to generate a single random PUF bit.

Rather than using a PUF circuit to directly generate a device dependent key, a fuzzy commitment scheme is used to mask the user-generated secret key [JW99]. The fuzzy commitment scheme is necessary as the PUF generates a device-unique but noisy bit sequence. During reproduction, the fuzzy commitment corrects errors in the sequence and reproduces the key on demand. This reconstructed key is directly used by cryptographic core, as shown in Figure 7.4, and is not accessible by any other IP core and software in the system. Here, for error correction, a (23,12,7) Golay linear block code is used, where 12 bits of the message is encoded to a 23-bit codeword, with a Hamming distance of at least 7 between any code pair. This module is part of the static bitstream which is directly loaded after power-up by the FSBL. The core is implemented as AXI-lite core with a 256-bit key interface directly to the hardened AES core and additional control signals that indicate the current status of the PUF as can be seen in Figure 7.4.

7.5.2 Leakage Resilient Authenticated Decryption with FIFO

Leakage resilient cryptography is an algorithmic countermeasure against SCA attacks. The aim of leakage resilience is to *bound the data leakage* so that an attacker cannot accumulate data about the processed secret. The advantage of using a leakage resilient cryptographic core, when compared to other side-channel protection schemes for block ciphers like masking and hiding, is that it does not rely on the availability of a true random number generator. For this work, the leakage resilient advanced encryption standard (LR-AES) core from Unterstein et al. [UHS$^+$18] is used. A common prerequisite for SCA is that the attacker is able to guess some intermediate values of the algorithm

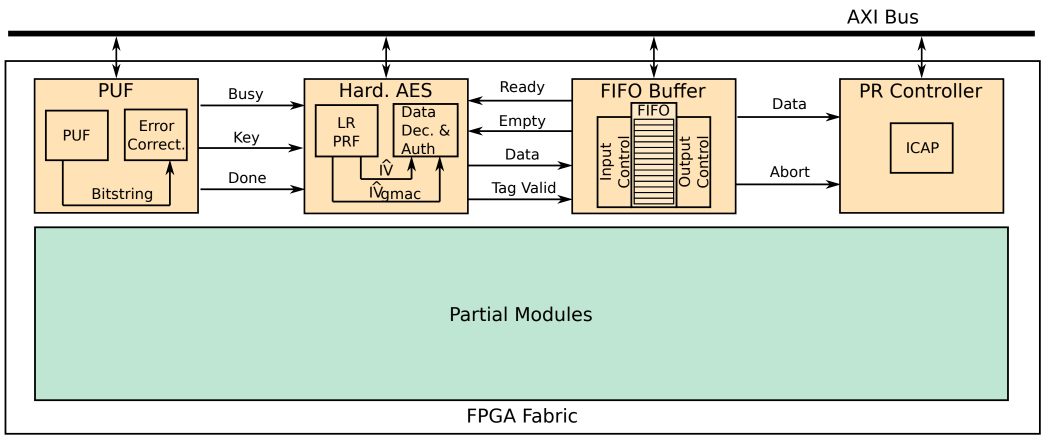


Figure 7.4: Overview of hardware modules in the FPGA

that depends on the secret and an input like the plaintext that is known to the attacker. The general idea behind construct used here is that only the initial leakage resilient pseudo random function (LR-PRF) stage needs to be side-channel secure (through means of leakage resilient cryptography), because behind it, no public inputs are processed by the block cipher.

This scheme can then be combined with any standard block cipher mode of operation to provide encryption only or even authenticated encryption. Here, authenticated encryption is used in order to provide confidentiality as well as integrity to the user hardware modules to be loaded to the FPGA. The side-channel hardened AES core is made up of a NIST standard block cipher AES-128 for the decryption in output feedback mode (OFB) [Nat01], a Galois message authentication code (GMAC) [Nat07] module for the authentication of binaries and a LR-PRF to provide security against sophisticated attacks using localized electromagnetic measurements. The LR-PRF is used to derive a secret pseudorandom state from a public initialization vector (IV) which is then used as a (secret) IV for the AES and GMAC, denoted as $\hat{IV}$ and $\hat{IV}_{gmac}$ in Figure 7.4.

The side-channel hardened AES core processes blocks of 128 bits of data at a time which is sent to it over the AXI interface. Additionally, the module also takes as input 2 x 128-bit nonce as IVs for the LR-PRF and optionally also unencrypted additional authentication data of arbitrary length. The key for the AES is reconstructed using the PUF and directly transferred to it as can be seen in Figure 7.4.

A typical problem when decrypting and authenticating the complete partial bitstream in one pass is that the authentication tag is verified only at the end after all the data is processed. In our case, this means that following decryption the data is sent to the PR controller to reconfigure the device before the authentication tag is verified. In order to avoid unauthenticated data from maliciously configuring or damaging the device, each partial bitstream is divided into segments. Each segment is individually encrypted and the authentication tag is validated at the end of each segment. For this, the side-channel hardened AES core is enhanced to include a BRAM implemented as FIFO, as shown in Figure 7.4. After decryption, the data blocks are transferred to the FIFO. The AES core transfers the decrypted data to the FIFO when the ready signal is high, indicating that the FIFO still has free space to accept data. The FIFO has a width of 32 bits meaning each block of decrypted data is stored over four slots in the FIFO. One full segment of the decrypted data is stored in the FIFO until the authentication tag for that segment is verified. The partial bitstream is divided into segments of 4096 bytes with an 128-bit authentication tag appended to it.

Xilinx Zynq offers two different sizes of BRAM: 18 Kbits and 36 Kbits. Depending on the resource requirement for the use-case we can switch between two sizes for the bitstream segments. By using the smaller BRAM, we can save on hardware resources however this could lead to a longer configuration time for the partial bitstreams. While with the larger BRAM, we have a shorter configuration time at the cost of extra hardware resources.

Once the authentication tag is successfully verified, data of that segment is transferred to the PR controller. Simultaneously, the decryption and authentication of the next segment of the partial bitstream begins.

7.5.3 Partial Reconfiguration (PR) Controller

This controller module receives segments of the decrypted partial bitstream following authentication from the FIFO and then passes it to the Xilinx internal configuration access port (ICAP) interface. The ICAP is a proprietary interface from Xilinx typically used for configuration readback and reconfiguration of the FPGA. The ICAP has direct access to the configuration memory through the configuration registers, hence can be used to dynamically reconfigure the FPGA. It consists of a clock input, two dedicated ports to read and write (each 32-bit wide), an active low enable pin (CSIB) and a RDWRB signal to toggle between device reconfiguration or configuration readback. In the case of device reconfiguration, the reconfiguration data is sent via the 32-bit input port. While in the configuration readback mode, the configuration data is sent out via the output port. No alignment of the bitstream is required, the actual configuration data must be preceded by the sync word `x"AA995566"`, which can be at any offset and succeeded by the desync word `x"0000000D"`.

As ICAP processes 32 bits of data per clock cycle thus, one block of decrypted data is processed over four clock cycles by the ICAP. Additionally in this work, the ICAP only operates in the device reconfiguration mode and hence no configuration data can be read back through the ICAP.

7.6 Results

Here, we describe the prototype implementation for the Xilinx Zynq-7020 FPGA SoC. The resource utilization is noted in Table 7.1. Overall, we only use around 22 percent of the available slices and 1.5 percent of the available BRAM of this device, which is among the smaller, low-cost devices of the Xilinx Zynq-7000 product range. Hence, and despite using a smaller device, 78 percent of the unused slices are still available to implement user hardware cores. In general the PUF can also be partially reconfigured after the successful generation of the key. This also further increases the amount of space left for user hardware cores.

Table 7.1: Resource utilization

Module	**Slices**		**BRAM**
	LUTs	**Registers**	**RAM36**
LR Authenticated Decryption	4568	2810	0
PicoPUF with Fuzzy Commitment	3917	4179	1
PR Controller	50	53	0
ICAP FIFO buffer	99	124	1
Overall (incl. interconnects)	9274	7911	2

All the hardware building blocks are configured and managed over the AXI interface by software drivers that are patched to a standard U-boot and FSBL. The helper data for the PUF is generated once and then hard coded into the FSBL.

7.7 Overall System Security

In this section, we review the overall system security by recalling the boot process and listing the different threat vectors post-integration together with the implemented mitigation techniques. A summary of the findings for each boot stage is listed in Table 7.2.

Table 7.2: Overview of system security

Threat vector	Mitigation
Stage 0. BootROM	
Load malicious FSBL	Integrity check of FSBL using RSA
Stage 1. FSBL and static FPGA configuration	
Learning the PUF	Key reproduction triggered by FSBL, following which PUF is locked
Readout encryption key	Key is directly transferred to AES
SCA on PUF	Helper data is authenticated before use
Stage 2. U-boot	
Loading unauthenticated configuration	FIFO is used to buffer configuration

Load malicious FSBL or U-boot

An adversary may attempt to bypass the implemented security mechanism by replacing the boot images with malicious ones. This is prevented by using the manufacturer-provided RSA signature verification. After power-up, the BootROM verifies the signature of the FSBL, only after successful verification of the signature is the control handed off to the FSBL. This check is enabled by burning an eFuse and cannot be disabled. Subsequently, the FSBL checks the signature of the U-boot. An adversary trying to forge any of those signatures would require the private key which is not stored on the device. Only the hash of the public key is stored on the device's efuses.

Learning the PUF

For this work we use an identity-generator type of PUF which in general is not susceptible to modeling attacks. However, if a challenge-response PUF was to be used it would be prone to modeling attacks, where a mathematical model of the PUF can be derived with access to a set of challenge-response pairs. This model behaves identical to the original PUF, thus allowing an adversary to clone the PUF [RSS+10]. To prevent this, we build in a mechanism to limit the access to the PUF. Following the loading of the static bitstream by the FSBL, the FSBL is allowed to trigger a PUF key reproduction only once. After the successful reproduction of the key, the PUF is locked by the hardware and is not accessible until the next power-up.

Readout of the encryption key

An attacker may attempt to corrupt the bitstream so as to include additional lines to readout the key. This however is prevented by verifying the integrity (at the same time as its authenticity) of the bitstream before it is loaded. Thus, what remains is to attempt to read out the key at run-time, e.g. via shared interfaces. To protect the key from being read by an unauthorised entity, the reproduced key is directly transferred to the LR-AES core. The key never leaves the FPGA fabric and is not transferred over any shared resources of the FPGA SoC, this ensures that no other entity has access to the key.

SCA attacks on the PUF

SCA attacks on PUF designs are two-fold:

1. Attacks on the core PUF instance

2. Attacks on the post-processing

The PUF used in this work is not expected to be susceptible to attacks on the bit generation as it performs evaluations in one clock cycle. The post-processing, however, is at a higher risk of SCA [MSSS11, TPS17]. In our design the post-processing consists of linear operations, which means an attacker would need a large number of traces to successfully carry out differential attacks on the PUF output [Pro05]. As we authenticate the helper data, which is included in the FSBL, we do not have the risk of an attacker loading malicious helper data. This leaves the attacker with limited number of traces. The attacker may still be able to take several measurements of this single trace which can be averaged to reduce the noise. Even in these circumstances it seems unlikely that attacker would be able to perform a simple power analysis.

Load unauthenticated data to FPGA

Authenticated encryption is used to protect confidentiality and integrity of the partial bitstreams where the decryption and authentication are performed in parallel. As the authentication tag can only be checked at the end, an adversary can send unauthentic data and it will be decrypted before the invalid tag is noticed. If the switch boxes and LUTs of the FPGA are falsely configured, it could result in short-circuits in the fabric and thereby destroying parts of the device. To prevent decrypted data from being directly transferred to the PR controller before the authenticity can be verified, the bitstream is divided into segments. Each segment has its own verification tag and is buffered after decryption until it has been successfully authenticated. Only after the successful verification of a segment, the buffered contents are transferred to the PR controller.

7.8 Generalization

While we target a Xilinx Zynq-7020 FPGA SoC for our proof of concept, the method is agnostic to a specific FPGA manufacturer/family as long as it provides certain common functionalities. We now discuss the portability of the presented design to other platforms and the additional steps that need to be taken. The two key requirements to port this design are that the target provides:

1. Authentication and integrity checking of the static bitstream using public key cryptography

2. Partial reconfiguration

In the initial boot step, an authenticity and integrity check of the static bitstream containing the custom cryptographic cores is necessary. We used signature verification with public key cryptography on the Xilinx Zynq-7000 device to achieve this. Public key cryptography for this purpose is a widely adopted feature and is present in the majority of FPGA SoCs that are currently available. The benefit of using public key cryptography is that no additional *secret* key material is necessary and the operation does not need to be protected against key-recovery attacks.

Partial reconfiguration is necessary to reconfigure parts of the FPGA fabric at run-time with user cores that were decrypted by our protected engine. This feature is supported by the leading manufacturers of FPGA SoCs such as Xilinx and Intel. Thus this design can be ported to these devices with low effort.

As the Xilinx Zynq-7000 devices do not provide sufficient user accessible key storage (only 32 bits of user accessible eFuses are available), we opted to use a PUF for key storage. Newer devices, however, often provide user accessible secure key storage. Some devices like the Zynq Ultrascale+ Xilinx and Stratix 10 from Intel even include hard-core PUFs and allow secure boot using these PUFs. In such cases the PUF can be omitted if the key storage is trusted to be secure.

7.9 Summary

In this work, we implemented a scheme to secure FPGA SoC configurations by retrofitting a standard FPGA SoC using dynamic PR, a side-channel secure cryptographic core and a PUF-based secure key storage to bind the configuration to a specific device. It is no longer necessary to rely on manufacturer-provided (possibly insecure) encryption, although it may be still necessary to sign the static FPGA configuration by manufacturer-provided features. With the presented scheme we also have the added advantage of providing field updates to not just the user-hardware cores but also the integrated custom cryptographic cores.

Chapter 8

Conclusion

In this thesis, we have shown the impact of malicious hardware on reconfigurable SoCs. The work provides a comprehensive study about the current threats of malicious hardware modules, i.e. hardware trojans, in the IC development process. Specifically the threat of malicious inclusion at different stages of IC development and the factors influencing it are highlighted. Furthermore, the strengths and challenges of trojan insertion and trojan detection techniques at each development stage are discussed. This analysis concludes that trojans are easier to insert during the functional design phase as compared with later fabrication stages. In addition, trojan detection and prevention techniques are cheaper during the functional design phase than during the fabrication phase. Current research focuses a lot on the detection techniques post-fabrication. This might not be very effective as malicious functionality inserted during the design phase will be difficult to detect by only using post-fabrication detection techniques. Research should focus more on the early design stage, especially in regard with the secure integration of third party IP. As another result, we show that the test development phase is also vulnerable to malicious modification and steps must be taken to prevent this.

With the understanding of the vulnerabilities to the IC development chain, the threat of malicious third party IP cores on the system security of new contemporary platforms, FPGA SoCs is analyzed. We demonstrate the feasibility and practical impact of attacks on key security mechanisms of FPGA SoCs through potentially malicious hardware cores on the FPGA. Two proof-of-concept implementations demonstrate the impact of such attack vectors. We show how the secure software update process can be compromised such that an adversary is able to download malicious updates. From this we learn that such attacks could in newer and expensive devices be prevented by correctly configuring the IOMMU. However, leaving older and resource constrained platforms which have no IOMMU still vulnerable. Next, the impact of malicious hardware in FPGA SoCs on the secure boot process, which is one of the most crucial security mechanisms for embedded devices is studied. Here, a malicious hardware core is used to bypass the secure boot process to load a malicious kernel over the network. Thereby bypassing other implemented security mechanisms such as an IOMMU, as they are usually set-up later in the boot process by the kernel.

With the accessibility to third party IP stores on the rise, and as embedded systems are enticing new fields, trust in outsourced IP cores is a growing concern. Hence, in addition to studying the threats of malicious hardware to system security, techniques to protect the system against such threats are also investigated. In that, two concrete countermeasures are developed to enhance the security of such systems. To prevent unauthorized accesses to the memory we show how third party IP with access to memory can be isolated using our security enhanced AXI-wrapper. This helps protect against attacks like the one described in this work by restricting the IP cores with access to the memory. We also present a sound scheme to retrofit existing systems, whose cryptographic cores have been subject to side-channel attacks. Dynamic partial reconfiguration together with secure and updatable cryptographic cores are used to securely configure user hardware modules in an FPGA SoC. The goal here is to step away from the reliance on manufacturer-provided security mechanisms and move towards user-trusted cryptographic cores to secure the system.

Future Work

This work highlights the general threat of malicious hardware on the overall security of FPGA SoCs. As these platforms become pervasive, so are the threats to such platforms. For instance, most leading cloud service providers are now also integrating FPGAs in their computing centers for applications that require more processing power. In our opinion, the threat vectors will also extend to remote attacks on such platforms.

Attack vectors which were assumed to require physical access, such as side-channel attacks, to the device are now feasible through remote channels. Recent publications have shown, how a malicious core on the FPGA can be used to perform power analysis attacks of other cores on the same fabric [SGMT18a, ZS18] as well as between two chips on the same PCB [SGMT18b]. Hence one area of future work is to analyze and secure FPGA-based cloud services. It is essential to ensure that a malicious core in one FPGA is not able to corrupt other parts of the system.

Another area for future research is in the development of trusted tools. At the moment researchers and other system integrators are mostly dependent on manufacturers provided tools which are typically closed source. Making it hard to review them. In the future, more effort could be spend in developing such open-source tools. Work has already begun in this direction, e.g. Project IceStorm for Lattice FPGAs, but a lot of effort is still required to make them a viable alternative to the manufacturer-provided tools.

List of Publications

[JMHS14] Nisha Jacob, Dominik Merli, Johann Heyszl, and Georg Sigl. *Hardware trojans: Current challenges and approaches.* IET Computers & Digital Techniques, 8(6): pages 264–273, 2014.

[JRZ$^+$17] Nisha Jacob, Carsten Rolfes, Andreas Zankl, Johann Heyszl, and Georg Sigl. *Compromising FPGA SoCs using malicious hardware blocks.* In Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017, pages 1122–1127, 2017.

[JHZ$^+$17] Nisha Jacob, Johann Heyszl, Andreas Zankl, Carsten Rolfes, and Georg Sigl. *How to break secure boot on FPGA SoCs through malicious hardware.* In Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings, pages 425–442, 2017.

[JWH$^+$17] Nisha Jacob, Jakob Wittmann, Johann Heyszl, Robert Hesselbarth, Florian Wilde, Michael Pehl, Georg Sigl, and Kai Fischer. *Securing FPGA SoC configurations independent of their manufacturers.* In 30th IEEE International System-on-Chip Conference, SOCC 2017, Munich, Germany, September 5-8, 2017, pages 114–119, 2017.

[UJH$^+$19] Florian Unterstein, Nisha Jacob, Neil Hanley, Chongyan Gu, and Johann Heyszl. *SCA secure and updatable crypto engines for FPGA SoC bitstream decryption.* In Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop, ASHES@CCS 2019, London, UK, November 15, 2019, pages 43–53. ACM, 2019.

Bibliography

[ABK$^+$07] Dakshi Agrawal, Selçuk Baktir, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. Trojan detection using IC fingerprinting. In *IEEE Symposium on Security and Privacy*, pages 296–310. IEEE Computer Society, 2007.

[Ade08] S. Adee. The hunt for the kill switch. *IEEE Spectr.*, 45(5):34–39, May 2008.

[AK09] Yousra Alkabani and Farinaz Koushanfar. Consistency-based characterization for IC trojan detection. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 123–127, New York, NY, USA, 2009. ACM.

[All06] All your private keys are belong to us . Tobias Klein, 2006. http://www.trapkit.de/research/sslkeyfinder/keyfinder_v1.0_20060205.pdf (Accessed April 16, 2018).

[Alt03] Altera Corporation. FLEX 10K embedded programmable logic device family, 2003. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ds/archives/dsf10k.pdf (Accessed May 18, 2018).

[Alt11] Altera Corporation. *SignalTap II*, August 2011. ftp://ftp.altera.com/up/pub/Intel_Material/11.1/Tutorials/Verilog/SignalTap.pdf (Accessed July 17, 2018).

[Alt13] Altera Corporation. *FPGA-Based Control for Electric Vehicle and Hybrid Electric Vehicle Power Electronics*, 2013. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01210-electric-vehicles.pdf (Accessed August 24, 2018).

[Alt15] Altera Corporation. Stratix 10 Secure Device Manager Provides Best-in-Class FPGA and SoC Security, 2015.

https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01252-secure-device-manager-for-fpga-soc-security.pdf (Accessed October 3, 2019).

[Alt16] Altera Corporation. Arria 10 SoC boot user guide, 2016. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-a10-soc-boot.pdf (Accessed October 3, 2019).

[AMD11] AMD. *I/O Memory Management Unit, Rev 2.0*, 2011. http://developer.amd.com/wordpress/media/2012/10/48882.pdf (Accessed June 1, 2018).

[ARM] ARM mbed. mbed TLS. https://tls.mbed.org/. (Accessed Auguest 15, 2019).

[ARM08] ARM Limited. *Building a Secure System using TrustZone Technology*, 2008. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf (Accessed October 3, 2019).

[ARM11] ARM. *AMBA® AXI™ and ACE™ Protocol Specification, Issue D*, 2011. http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf (Accessed August 15, 2019).

[ARM16] ARM. *System Memory Management Unit Architecture Specification, Version 2.0*, 2016. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0062d.c/IHI0062D_c_system_mmu_architecture_specification.pdf (Accessed October 3, 2019).

[ARS13] ARS Technica. Stop using NSA-influenced code in our products, RSA tells customers, 2013. http://arstechnica.com/security/2013/09/stop-using-nsa-influence-code-in-our-product-rsa-tells-customers/ (Accessed April 16, 2018).

[Bar15] BarcoSilex. BA415-AES-GCM 10 to 100 Gbps IP Core, 2015. http://www.xilinx.com/products/intellectual-property/1-4sw1c9.html (Accessed April 4, 2016).

[Bar16] BarcoSilex. BA413-SHA1, SHA2 and HMAC IP Core, 2016. http://www.barco-silex.com/ip-cores/encryption-engine/BA413 (Accessed April 4, 2016).

[BBG13] James Ball, Juliani Borger, and Glenn Greenwald. Revealed: How US and UK spy agencies defeat internet privacy and security challenges. *The Guardian*, 2013. http://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security (Accessed October 3, 2019).

[BCFH08] Mainak Banga, Maheshwar Chandrasekar, Lei Fang, and Michael S. Hsiao. Guided test generation for isolation and detection of embedded trojans in ICs. In Vijaykrishnan Narayanan, Zhiyuan Yan, Enrico Macii, and Sanjukta Bhanja, editors, *Proceedings of the 18th ACM Great Lakes Symposium on VLSI 2008, Orlando, Florida, USA, May 4-6, 2008*, pages 363–366. ACM, 2008.

[BCN+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[BDG+13] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, Xuan Thuy Ngo, and Laurent Sauvage. Hardware trojan horses in cryptographic IP cores. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*, pages 15–29. IEEE Computer Society, 2013.

[BFS15] Chongxi Bao, Domenic Forte, and Ankur Srivastava. Temperature tracking: Toward robust run-time detection of hardware trojans. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(10):1577–1585, 2015.

[BGV15] J. Balasch, B. Gierlichs, and I. Verbauwhede. Electromagnetic circuit fingerprints for hardware trojan detection. In *2015 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, pages 246–251, August 2015.

[BH09] Mainak Banga and Michael S. Hsiao. VITAMIN: Voltage inversion technique to ascertain malicious insertions in ICs. In *IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2009, San Francisco, CA, USA, July 27,* *2009. Proceedings*, pages 104–107. IEEE Computer Society, 2009.

[BH10] Mainak Banga and Michael S. Hsiao. Trusted RTL: trojan detection methodology in pre-silicon designs. In *HOST 2010, Proceedings of the 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 13-14 June 2010, Anaheim Convention Center, California, USA*, pages 56–59. IEEE Computer Society, 2010.

[BPOD14] Jeremie Brunel, Renaud Pacalet, Salaheddine Ouaarab, and Guillaume Duc. Secbus, a software/hardware architecture for securing external memories. In *2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, MobileCloud 2014, Oxford, United Kingdom, April 8-11, 2014*, pages 277–282, 2014.

[BR99] Vaughn Betz and Jonathan Rose. FPGA routing architecture: Segmentation and buffering to optimize speed and density. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays, FPGA 1999, Monterey, CA, USA, February 21-23, 1999*, pages 59–68, 1999.

[BRPB13] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy dopant-level hardware trojans. In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES*, volume 8086 of *Lecture Notes in Computer Science*, pages 197–214. Springer, 2013.

[BS13] James Ball and Bruce Schneier. Explaining the latest NSA revelations – Q&A with internet privacy experts. The Guardian, 2013. http://www.theguardian.com/commentisfree/2013/sep/06/nsa-surveillance-revelations-encryption-expert-chat (Accessed April 16, 2018).

[BSH12] F. Benz, A. Seffrin, and S.A. Huss. Bitfile interpretation library (bil). GitHub, 2012. https://github.com/florianbenz/bil (Accessed March 1, 2019).

[CB09] Rajat Subhra Chakraborty and Swarup Bhunia. Security against hardware trojan through a novel application of design obfuscation. In *2009 International Conference on Computer-Aided Design, ICCAD 2009, San Jose, CA, USA, November 2-5, 2009*, pages 113–116. ACM, 2009.

[CDG$^+$12] Pascal Cotret, Florian Devic, Guy Gogniat, Benoît Badrignans, and Lionel Torres. Security enhancements for fpga-based mpsocs: A boot-to-runtime protection flow for an embedded linux-based system. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), York, United Kingdom, July 9-11, 2012*, pages 1–8, 2012.

[CG13] Byeongju Cha and Sandeep K. Gupta. Trojan detection via delay measurements: A new approach to select paths and vectors to maximize effectiveness and minimize cost. In Enrico Macii, editor, *DATE*, pages 1265–1270. EDA Consortium San Jose, CA, USA / ACM DL, 2013.

[Cod14] Codenomicon. *Heartbleed, CVE-2014-0160*, 2014. http://heartbleed.com/ (Accessed April 16, 2018).

[CPB08] Rajat Subhra Chakraborty, Somnath Paul, and Swarup Bhunia. On-demand transparency for improving hardware trojan detectability. In *IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2008, Anaheim, CA, USA, June 9, 2008. Proceedings*, pages 48–50. IEEE Computer Society, 2008.

[CRRC05] Joel Coburn, Srivaths Ravi, Anand Raghunathan, and Srimat T. Chakradhar. SECA: Security-enhanced communication architecture. In *Proceedings of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2005, San Francisco, California, USA, September 24-27, 2005*, pages 78–89. ACM, 2005.

[CSPN13] Rajat Subhra Chakraborty, Indrasish Saha, Ayan Palchaudhuri, and Gowtham Kumar Naik. Hardware trojan insertion by direct modification of FPGA configuration bitstream. *IEEE Design & Test*, 30(2):45–54, 2013.

[CWP$^+$09] Rajat Subhra Chakraborty, Francis G. Wolff, Somnath Paul, Christos A. Papachristou, and Swarup Bhunia. MERO: A statistical approach for hardware trojan detection. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2009.

[DAR06] DARPA BAA06-40: TRUST for Integrated Circuits. In *Defense Advanced Research Projects Agency*, 2006. https://www.darpa.mil/program/trusted-integrated-circuits (Accessed October 3, 2019).

[DeH99] André DeHon. Balancing interconnect and computation in a reconfiguable computing array (or, why you don't really want 100% LUT utilization). In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays, FPGA 1999, Monterey, CA, USA, February 21-23, 1999*, pages 69–78. ACM, 1999.

[Dic14] Dick Wilkins. UEFI firmware security best practices. *UEFI Plugfest*, 2014. http://www.uefi.org/sites/default/files/resources/2014_UEFI_Plugfest_06_Phoenix.pdf (Accessed August 24, 2018).

[DWZZ13] Zheng Ding, Qiang Wu, Yizhong Zhang, and Linjie Zhu. Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation. *Microprocessors and Microsystems*, 37(3):299 – 312, 2013.

[EGMP17] Maik Ender, Samaneh Ghandali, Amir Moradi, and Christof Paar. The first thorough side-channel hardware trojan. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 755–780. Springer, 2017.

[Ens13] Ensilica. Ensilica eSi - SHA-256, 2013. http://www.ensilica.com/wp-content/uploads/eSi-SHA-256.pdf (Accessed April 4, 2016).

[For05] Defense Science Board Task Force. Report of defense science board task force on high performance microchip supply, 2005. http://www.cra.org/govaffairs/images/2005-02-HPMS_Report_Final.pdf (Accessed October 10, 2014).

[Fre08] Freescale Semiconductor. Understanding cryptographic performance. Technical report, 2008. http://www.embeddeddeveloper.com/news_letter/files/CRYPTOWP_Rev2.pdf (Accessed April 4, 2016).

[FWS+17] Marc Fyrbiak, Sebastian Wallat, Pawel Swierczynski, Max Hoffmann, Sebastian Hoppach, Matthias Wilhelm, Tobias Weidlich, Russell Tessier, and Christof Paar. Hal — the missing piece of the puzzle for hardware reverse engineering, trojan detection and insertion. Cryptology ePrint Archive, Report 2017/783, 2017.

[Gam11] Gamma International. *Tactical IT Intrusion Portfolio: FINFIREWIRE*, 2011. https://wikileaks.org/spyfiles/files/0/293_GAMMA-201110-FinFireWire.pdf (Accessed August 15, 2019).

[GBMB15] Benoît Gonzalvo, Eric Bourbao, Fabien Majéric, and Lilian Bossue. JTAG Combined Attacks. In *TRUDEVICE 2015- 4th Workshop on Secure Hardware and Security Evaluation*, pages 56–69, Saint-Malo,France, 2015.

[GBMB16] Benoît Gonzalvo, Eric Bourbao, Fabien Majéric, and Lilian Bossue. JTAG combined attacks. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2016.

[GN13] Barton Gellman and Ellen Nakashima. U.S. spy agencies mounted 231 offensive cyber-operations in 2011, documents show, 2013. http://www.washingtonpost.com/world/national-security/us-spy-agencies-mounted-231-offensive-cyber-operations-in-2011-documents-show/2013/08/30/d090a6ae-119e-11e3-b4cb-fd7ce041d814_story.html (Accessed October 3, 2019).

[GO15] Chongyan Gu and Máire O'Neill. Ultra-compact and robust fpga-based PUF identification generator. In *2015 IEEE International Symposium on Circuits and Systems, ISCAS 2015, Lisbon, Portugal, May 24-27, 2015*, pages 934–937. IEEE, 2015.

[Hel16] Helion. HTSHA-FAST64: Fast SHA-384/512 Hashing, 2016. http://www.xilinx.com/products/intellectual-property/1-8dyf-612.html (Accessed April 4, 2016).

[Hex] Hex-Rays. IDA Debugger. https://www.hex-rays.com/products/ida/index.shtml. (Accessed August 08, 2019).

[HFK+10] Matthew Hicks, Murph Finnicum, Samuel T. King, Milo M. K. Martin, and Jonathan M. Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *2010 IEEE Symposium on Security and Privacy*, pages 159–172, 2010.

[HZGJ17] J. He, Y. Zhao, X. Guo, and Y. Jin. Hardware trojan detection through chip-free electromagnetic side-channel statistical analysis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2939–2948, October 2017.

[IBM99] IBM. The CoreConnect™ bus architecture, 1999. http://www2.informatik.hu-berlin.de/~fwinkler/psvfpga/amirix/CoreConnect.pdf (Accessed August 15, 2019).

[IFI15] IFIXIT. *Pebble Time Teardown*, 2015. https://www.ifixit.com/Teardown/Pebble+Time+Teardown/42382 (Accessed August 24, 2018).

[Int17] Intel Corporation. *Intel user-customizable SoC FPGAs*, 2017. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/br/br-soc-fpga.pdf (Accessed August 15, 2019).

[Int19] Intel Corporation. *Intel Agilex FPGAs and SoCs*, 2019. https://www.intel.com/content/www/us/en/products/programmable/fpga/agilex.html (Accessed November 4, 2019).

[Jef16] Jeff Barr. Developer preview – EC2 instances (F1) with programmable hardware. *Amazon Web Services*, 2016. https://aws.amazon.com/about-aws/whats-new/2016/11/available-for-preview-amazon-ec2-f1-instances-custom-fpgas-for-hardware-acceleration/ (Accessed October 2, 2019).

[Jeo14] Jeong Wook Oh. Reverse engineering flash memory for fun and benefit. In *Blackhat US*, 2014.

[JGS+14] M. Jagasivamani, P. Gadfort, M. Sika, M. Bajura, and M. Fritze. Split-fabrication obfuscation: Metrics and techniques. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 7–12, May 2014.

[JHC$^+$18] Don Owen Jr., Derek Heeger, Calvin Chan, Wenjie Che, Fareena Saqib, Matthew Areno, and Jim Plusquellic. An autonomous, self-authenticating, and self-contained secure boot process for field-programmable gate arrays. *Cryptography*, 2(3):15, 2018.

[JHZ$^+$17] Nisha Jacob, Johann Heyszl, Andreas Zankl, Carsten Rolfes, and Georg Sigl. How to break secure boot on FPGA SoCs through malicious hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 425–442, 2017.

[JJ08] Susmit Jha and Sumit Kumar Jha. Randomization based probabilistic approach to detect trojan circuits. In *11th IEEE High Assurance Systems Engineering Symposium, HASE 2008, Nanjing, China, December 3 - 5, 2008*, pages 117–124. IEEE Computer Society, 2008.

[JJC13] Appelbaum Jacob, Horchert Judith, and Stöcker Christian. Shopping for spy gear: Catalog advertises NSA toolbox. In *Der Spiegel*, 2013. http://www.spiegel.de/international/world/catalog-reveals-nsa-has-back-doors-for-numerous-devices-a-940994.html (Accessed April 16, 2018).

[JM08] Yier Jin and Yiorgos Makris. Hardware trojan detection using path delay fingerprint. In *IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2008, Anaheim, CA, USA, June 9, 2008. Proceedings*, pages 51–57, 2008.

[JM10] Yier Jin and Yiorgos Makris. Hardware trojans in wireless cryptographic ICs. *IEEE Design & Test of Computers*, 27(1):26–35, 2010.

[JMHS14] Nisha Jacob, Dominik Merli, Johann Heyszl, and Georg Sigl. Hardware trojans: Current challenges and approaches. *IET Computers & Digital Techniques*, 8(6):264–273, 2014.

[JRZ$^+$17] Nisha Jacob, Carsten Rolfes, Andreas Zankl, Johann Heyszl, and Georg Sigl. Compromising FPGA SoCs using malicious hardware blocks. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 1122–1127, 2017.

[JW99] Ari Juels and Martin Wattenberg. A fuzzy commitment scheme. In Juzar Motiwalla and Gene Tsudik, editors, *CCS '99, Proceedings of the 6th ACM Conference on Computer and Communications Security, Singapore, November 1-4, 1999.*, pages 28–36. ACM, 1999.

[JWH$^+$17] Nisha Jacob, Jakob Wittmann, Johann Heyszl, Robert Hesselbarth, Florian Wilde, Michael Pehl, Georg Sigl, and Kai Fischer. Securing FPGA SoC configurations independent of their manufacturers. In *30th IEEE International System-on-Chip Conference, SOCC 2017, Munich, Germany, September 5-8, 2017*, pages 114–119, 2017.

[KAABS15] B. Khaleghi, A. Ahari, H. Asadi, and S. Bayat-Sarmadi. FPGA-based protection scheme against hardware trojan horse insertion using dummy logic. *IEEE Embedded Systems Letters*, 7(2):46–50, June 2015.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 388–397, 1999.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[KPS13] Sebastian Kutzner, Axel York Poschmann, and Marc Stöttinger. Hardware trojan design and detection: A practical evaluation. In *Proceedings of the Workshop on Embedded Systems Security, WESS 2013, Montreal, Quebec, Canada, September 29 - October 4, 2013*, pages 1:1–1:9. ACM, 2013.

[KRRT10] Ramesh Karri, Jeyavijayan Rajendran, Kurt Rosenfeld, and Mohammad Tehranipoor. Trustworthy hardware: Identifying and classifying hardware trojans. *IEEE Computer*, 43(10):39–46, 2010.

[KTC$^+$08] Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. Designing and implementing malicious hardware. In *First USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET '08, San Francisco, CA, USA, April 15, 2008, Proceedings*. USENIX Association, 2008.

[LDP15] Letitia W. Li, Guillaume Duc, and Renaud Pacalet. Hardware-assisted memory tracing on new SoCs embedding FPGA fabrics. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 461–470. ACM, 2015.

[LDT12] Min Li, Azadeh Davoodi, and Mohammad Tehranipoor. A sensor-assisted self-authentication framework for hardware trojan detection. In Wolfgang Rosenstiel and Lothar Thiele, editors, *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, pages 1331–1336. IEEE, 2012.

[Lei15] Steve Leibson. *The Roads Must Roll: Zynq SoC will be used to build Intelligent Transport System in Singapore*. Xilinx Inc, November 2015. https://forums.xilinx.com/t5/Xcell-Daily-Blog/The-Roads-Must-Roll-Zynq-SoC-will-be-used-to-build-Intelligent/ba-p/600630 (Accessed August 15, 2019).

[LKG$^+$09] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne Burleson. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 382–395. Springer, 2009.

[LL08] Jie Li and John Lach. At-speed delay characterization for IC authentication and trojan horse detection. In Mohammad Tehranipoor and Jim Plusquellic, editors, *IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2008, Anaheim, CA, USA, June 9, 2008. Proceedings*, pages 8–14. IEEE Computer Society, 2008.

[LSM$^+$16] Meng Li, K. Shamsi, T. Meade, Zheng Zhao, Bei Yu, Yier Jin, and D. Z. Pan. Provably secure camouflaging strategy for IC protection. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2016.

[Mar11] Mark Beaumont and Bradley Hopkins and Tristan Newby. Hardware Trojans-Prevention, Detection, Countermeasures (A Literature Review). *Defence Science and Technology Organisation, Command, Control, Communication and Intelligence Division*, 2011.

[MBKP11] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks - extracting keys from Xilinx Virtex-II FPGAs. *IACR Cryptology ePrint Archive*, 2011:390, 2011.

[McK15] Frank McKeen. Intel software guard extensions(Intel SGX). Intel Corporation, 2015. https://web.stanford.edu/class/ee380/Abstracts/150415-slides.pdf (Accessed June 1, 2018).

[MGK$^+$13] Michael Muehlberghuber, Frank K. Gurkaynak, Thomas Korak, Philipp Dunst, and Michael Hutter. Red team vs. blue team hardware trojan analysis: Detection of a hardware trojan on an actual ASIC. In ACM, editor, *Hardware and Architectural Support for Security and Privacy - HASP 2013, Second Workshop, Tel-Aviv, Israel, June 23, 2013, Proceedings.*, pages 1 – 8. ACM, 2013.

[Mic13] Microsemi Corporation. Specify and program security settings and keys with SmartFusion2 and IGLOO2 FPGAs, 2013. https://www.microsemi.com/document-portal/doc_view/132882-specify-and-program-security-settings-and-keys-with-smartfusion2-and-igloo2-fpgas (Accessed October 3, 2019).

[Mic15] Microsemi Corporation. UG0443: User guide SmartFusion2 and IGLOO2 FPGA security and best practices, revision 5.0, 2015. https://www.microsemi.com/document-portal/doc_download/132037-ug0443-smartfusion2-and-igloo2-fpga-security-best-practices-user-guide (Accessed October 3, 2019).

[Mic18a] Microsemi Corporation. *DS0128: IGLOO2 FPGA and SmartFusion2 SoC FPGA, Version 12.0*, 2018. https://www.microsemi.com/document-portal/doc_download/132042-igloo2-fpga-datasheet (Accessed October 3, 2019).

[Mic18b] Microsemi Corporation. *PolarFire SoC: Industry's First RISC-V SoC FPGA Architecture*, 2018. https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga (Accessed May 14, 2019).

[MOPS13] Amir Moradi, David Oswald, Christof Paar, and Pawel Swierczynski. Side-channel attacks on the bitstream encryption mechanism of Altera Stratix II: Facilitating black-box analysis using software reverse-engineering. In *The 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, February 11-13, 2013*, pages 91–100, 2013.

[MS16] Amir Moradi and Tobias Schneider. Improved side-channel analysis attacks on Xilinx bitstream encryption of 5, 6, and 7 series. *IACR Cryptology ePrint Archive*, 2016:249, 2016.

[MSSS11] Dominik Merli, Dieter Schuster, Frederic Stumpf, and Georg Sigl. Side-channel analysis of PUFs and fuzzy extractors. In *Trust and Trustworthy Computing - 4th International Conference, TRUST 2011, Pittsburgh, PA, USA, June 22-24, 2011. Proceedings*, pages 33–47, 2011.

[Nat01] National Institute of Standards and Technology (NIST). NIST special publication 800-38A recommendation for block cipher modes of operation. Technical report, 2001.

[Nat07] National Institute of Standards and Technology (NIST). NIST special publication 800-38D recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. Technical report, 2007.

[Noh12] Karsten Nohl. Low-cost chip microprobing. In *29th Chaos Communication Congress, 29C3*, 2012.

[NR08] Jean-Baptiste Note and Éric Rannaud. From the bitstream to the netlist. In Mike Hutton and Paul Chow, editors, *Proceedings of the ACM/SIGDA 16th International Symposium on*

Field Programmable Gate Arrays, FPGA 2008, Monterey, California, USA, February 24-26, 2008, page 264. ACM, 2008.

[NYW$^+$12] Seetharam Narasimhan, Wen Yueh, Xinmu Wang, Saibal Mukhopadhyay, and Swarup Bhunia. Improving IC security against trojan attacks through integration of security monitors. *IEEE Design & Test of Computers*, 29(5):37–46, 2012.

[Ope10] OpenCores. *Wishbone B4: Wishbone System-on-Chip (SoC) Interconnection Architecturefor Portable IP Cores*, 2010. http://opencores.org/opencores,wishbone (Accessed August 15, 2019).

[OSS17] Johannes Obermaier, Robert Specht, and Georg Sigl. Fuzzy-glitch: A practical ring oscillator based clock glitch attack. In *2017 International Conference on Applied Electronics (AE)*, pages 1–6. IEEE, September 2017.

[Pet15] Ed Peterson. *Leveraging asymmetric authentication to enhance security-critical applications using Zynq-7000 all programmable SoCs*. Xilinx, October 2015.

[PLS13] Nicole Perlroth, Jeff Larson, and Scott Shane. NSA able to foil basic safeguards of privacy on web, September 2013. http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html?pagewanted=all&_r=0 (Accessed October 3, 2019).

[PNNM09] Miodrag Potkonjak, Ani Nahapetian, Michael Nelson, and Tammara Massey. Hardware trojan horse detection using gate-level characterization. In *Proceedings of the 46th ACM/IEEE Design Automation Conference, DAC 2009*, pages 688–693, 2009.

[Pot10] Miodrag Potkonjak. Synthesis of trustable ICs using untrusted CAD tools. In Sachin S. Sapatnekar, editor, *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, pages 633–634. ACM, 2010.

[Pro05] Emmanuel Prouff. DPA attacks and S-boxes. In *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, pages 424–441, 2005.

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.

[RDVK16] J. Rajendran, A. M. Dhandayuthapany, V. Vedula, and R. Karri. Formal security verification of third party intellectual property cores for information leakage. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 547–552, January 2016.

[ROSW16] Eyal Ronen, Colin O'Flynn, Adi Shamir, and Achi-Or Weingarten. IoT goes nuclear: Creating a ZigBee chain reaction. *IACR Cryptology ePrint Archive*, 2016:1047, 2016.

[RPSK12] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of logic obfuscation. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC 2012, San Francisco, CA, USA, June 3-7, 2012*, pages 83–89. ACM, 2012.

[RSK13] Jeyavijayan Rajendran, Ozgur Sinanoglu, and Ramesh Karri. Is split manufacturing secure? In *Design, Automation and Test in Europe, DATE 2013, Grenoble, France, March 18-22, 2013*, pages 1259–1264. EDA Consortium San Jose, CA, USA / ACM DL, 2013.

[RSS$^+$10] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 237–249. ACM, 2010.

[RSSK13] Jeyavijayan Rajendran, Michael Sam, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of integrated circuit camouflaging. In *Proceedings of the 2013 ACM SIGSAC Conference* *on Computer and Communications Security, CCS 2013*, pages 709–720. ACM, 2013.

[Rus13] Russ Sevinsky. Funderbolt adventures in thunderbolt DMA attacks. In *BlackHat US*, 2013.

[RWTP08] Reza M. Rad, Xiaoxiao Wang, Mohammad Tehranipoor, and Jim Plusquellic. Power supply signal calibration techniques for improving detection resolution to hardware trojans. In *2008 International Conference on Computer-Aided Design, ICCAD 2008, San Jose, CA, USA, November 10-13, 2008*, pages 632–639. IEEE Computer Society, 2008.

[RZZ+15] Jeyavijayan Rajendran, Huan Zhang, Chi Zhang, Garrett S. Rose, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Fault analysis-based logic encryption. *IEEE Trans. Computers*, 64(2):410–424, 2015.

[SAB15] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, pages 57–64. IEEE, 2015.

[SFK+17] Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, Amir Moradi, and Christof Paar. Interdiction in practice - hardware trojan against a high-security USB flash drive. *J. Cryptographic Engineering*, 7(3):199–211, 2017.

[SGMT18a] Falk Schellenberg, Dennis R. E. Gnad, Amir Moradi, and Mehdi Baradaran Tahoori. An inside job: Remote power analysis attacks on FPGAs. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pages 1111–1116. IEEE, 2018.

[SGMT18b] Falk Schellenberg, Dennis R. E. Gnad, Amir Moradi, and Mehdi Baradaran Tahoori. Remote inter-chip power analysis side-channel attacks at board-level. In Iris Bahar, editor, *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018*, page 114. ACM, 2018.

[Skr13] Benjamin Skrehot. Reverse engineering the configuration of a Xilinx Virtex-6 FPGA. Master's thesis, Ruhr-Universität Bochum, June 2013.

[SMOP15] Pawel Swierczynski, Amir Moradi, David Oswald, and Christof Paar. Physical security evaluation of the bitstream encryption mechanism of Altera Stratix II and Stratix III FPGAs. *TRETS*, 7(4):34:1–34:23, 2015.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP 2013*, pages 48–62. IEEE Computer Society, 2013.

[SS99] Adi Shamir and Nicko Someren. Playing 'hide and seek' with stored keys. In Matthew Franklin, editor, *Financial Cryptography: Third International Conference, FC'99 Anguilla, British West Indies, February 22–25, 1999 Proceedings*, pages 118–124. Springer Berlin Heidelberg, 1999.

[ST12] Hassan Salmani and Mohammad Tehranipoor. Layout-aware switching activity localization to enhance hardware trojan detection. *IEEE Transactions on Information Forensics and Security*, 7(1):76–87, 2012.

[Ste14] Steven J. Murdoch. Introduction to trusted execution environments (TEE). *University of Cambridge*, 2014. http://sec.cs.ucl.ac.uk/users/smurdoch/talks/rhul14tee.pdf (Accessed October 3, 2019).

[STP09] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. New design strategy for improving hardware trojan detection and reducing trojan activation time. In *Proceedings of the 2009 IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2009*, pages 66–73. IEEE Computer Society, 2009.

[SW12a] Sergei Skorobogatov and Christopher Woods. Breakthrough silicon scanning discovers backdoor in military chip. In *Cryptographic Hardware and Embedded Systems – CHES 2012*, pages 23–40, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[SW12b] Sergei Skorobogatov and Christopher Woods. In the blink of an eye: There goes your AES key. *IACR Cryptology ePrint Archive*, 2012:296, 2012.

[SWR$^+$10] Yuriy Shiyanovskii, Francis G. Wolff, Aravind Rajendran, Christos A. Papachristou, Daniel J. Weyer, and W. Clay. Process reliability based trojans through NBTI and HCI effects. In Tughrul Arslan, Didier Keymeulen, David Merodio, Khaled Benkrid, Ahmet T. Erdogan, and Umeshkumar D. Patel, editors, *AHS*, pages 215–222. IEEE, 2010.

[Sym19] SymbiFlow. *SymbiFlow - open source FPGA tooling for rapid innovation*, 2019. https://symbiflow.github.io/ (Accessed October 10, 2019).

[TPS17] Lars Tebelmann, Michael Pehl, and Georg Sigl. EM side-channel analysis of BCH-based error correction for PUF-based key generation. In *Proceedings of the 2017 Workshop on Attacks and Solutions in Hardware Security*, ASHES '17, pages 43–52. ACM, 2017.

[TSZ$^+$11] Mohammad Tehranipoor, Hassan Salmani, Xuehui Zhang, Michel Wang, Ramesh Karri, Jeyavijayan Rajendran, and Kurt Rosenfeld. Trustworthy hardware: Trojan detection and design-for-trust challenges. *IEEE Computer*, 44(7):66–74, 2011.

[TZ13] Jiang Tun and Yi-Zhong Zhang. Method of bitstream resolving and circuit recovering of FPGA. *Computer Engineering*, 39(5):305, 2013.

[UHS$^+$18] Florian Unterstein, Johann Heyszl, Fabrizio De Santis, Robert Specht, and Georg Sigl. High-resolution EM attacks against leakage-resilient PRFs explained - and an improved construction. In *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*, pages 413–434, 2018.

[UJH$^+$19] Florian Unterstein, Nisha Jacob, Neil Hanley, Chongyan Gu, and Johann Heyszl. SCA secure and updatable crypto engines for FPGA soc bitstream decryption. In *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop, ASHES@CCS 2019, London, UK, November 15, 2019*, pages 43–53. ACM, 2019.

[WPBC08] Francis Wolff, Chris Papachristou, Swarup Bhunia, and Rajat S. Chakraborty. Towards trojan-free trusted ICs: Problem analysis and detection scheme. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 1362–1365. ACM, 2008.

[WRC08] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized I/O devices. In Rebecca Isaacs and Yuanyuan Zhou, editors, *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*, pages 15–28. USENIX Association, 2008.

[WSS13] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. FANCI: Identification of stealthy malicious logic using boolean functional analysis. In *ACM Conference on Computer and Communications Security*, pages 697–708. ACM, 2013.

[WSTP08] Xiaoxiao Wang, Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. Hardware trojan detection and isolation using current integration and localized current analysis. In *Proceedings of the 2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, DFT '08, pages 87–95. IEEE Computer Society, 2008.

[WTP08] Xiaoxiao Wang, Mohammad Tehranipoor, and Jim Plusquellic. Detecting malicious inclusions in secure hardware: Challenges and solutions. In *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2008*, pages 15–19. IEEE Computer Society, 2008.

[XFT14] Kan Xiao, Domenic Forte, and Mohammad Tehranipoor. A novel built-in self-authentication technique to prevent inserting hardware trojans. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 33(12):1778–1791, 2014.

[Xila] *Xilinx: The Programmable Logic Data Book.*

[Xilb] Xilinx Inc. Xilinx Github. https://github.com/Xilinx (Accessed August 15, 2019).

[Xil12] Xilinx Inc. *UG695: ISE In-Depth Tutorial, v14.1*, April 2012. https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/ise_tutorial_ug695.pdf (Accessed October 3, 2019).

[Xil14] Xilinx Inc. *UG585: Zynq-7000 All Programmable SoC - Technical Reference Manual, v1.8.1*, September 2014. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf (Accessed October 3, 2019).

[Xil15] Xilinx Inc. Security monitor IP, 2015. http://www.xilinx.com/support/documentation/product-briefs/security-monitor-ip-core-product-brief.pdf (Accessed October 4, 2019).

[Xil16a] Xilinx Inc. *PG 172: Integrated Logic Analyzer v6.1 - LogiCORE IP Product Guide Debugger*, April 2016. https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_1/pg172-ila.pdf (Accessed October 3, 2019).

[Xil16b] Xilinx Inc. *UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices*, June 2016. https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf (Accessed October 3, 2019).

[Xil16c] Xilinx Inc. xilinx-v2016.1: linux-xlnx, April 2016. https://github.com/Xilinx/linux-xlnx/releases/tag/xilinx-v2016.1 (Accessed August 15, 2019).

[Xil17] Xilinx Inc. *UG 1085: Zynq UltraScale+ MPSoC-Technical Reference Manual*, February 2017. http://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf (Accessed October 3, 2019).

[Xil18a] Xilinx Inc. *UG910: Vivado Design Suite User Guide, v2018.1*, April 2018. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug910-vivado-getting-started.pdf (Accessed October 3, 2019).

[Xil18b] Xilinx Inc. *XAPP1323: Developing Tamper-Resistant Designs with Zynq UltraScale+ Devices, v1.1*, August 2018. https://www.xilinx.com/support/documentation/application_notes/xapp1323-zynq-usp-tamper-resistant-designs.pdf (Accessed October 3, 2019).

[Xil19] Xilinx Inc. *WP505 Versal: The First Adaptive Compute Acceleration Platform (ACAP), v1.0.1*, September 2019. https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf (Accessed November 4, 2019).

[YHD+16] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd M. Austin, and Dennis Sylvester. A2: Analog Malicious Hardware. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 18–37, 2016.

[ZAZ+15] Boyou Zhou, Ronen Adato, Mahmoud Zangeneh, Tianyu Yang, Aydan Uyar, Bennett B. Goldberg, M. Selim Ünlü, and Ajay Joshi. Detecting hardware trojans using backside optical imaging of embedded watermarks. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 111:1–111:6. ACM, 2015.

[ZS18] Mark Zhao and G. Edward Suh. FPGA-based remote power side-channel attacks. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 229–244. IEEE Computer Society, 2018.

[ZT11a] Xuehui Zhang and Mohammad Tehranipoor. Case study: Detecting hardware trojans in third-party digital IP cores. In *HOST 2011, Proceedings of the 2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 5-6 June 2011, San Diego, California, USA*, pages 67–70. IEEE Computer Society, 2011.

[ZT11b] Xuehui Zhang and Mohammad Tehranipoor. RON: An on-chip ring oscillator network for hardware trojan detection. In *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*, pages 1638–1643. IEEE, 2011.

[ZYW+15] Jie Zhang, Feng Yuan, Lingxiao Wei, Yannan Liu, and Qiang Xu. VeriTrust: Verification for hardware trust. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(7):1148–1161, 2015.