TUM School of Computation, Information and Technology
Technische Universität München

Remote Security Threats and Protection of Modern FPGA-SoC Architectures

Mathieu E. A. Gross

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Die Dissertation wurde am 19.04.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 13.07.2023 angenommen.

Abstract

In the last decade we have seen an increase in the popularity of hardware based acceleration through the performance achieved in domains such as machine learning, communication, and security. Due to their flexibility and power-efficiency, Field Programmable Gate Arrays (FPGAs) are an interesting hardware acceleration platform which is nowadays adopted in the cloud computing domain as well as in embedded systems.

In this thesis, we consider FPGA System on Chips (FPGA-SoCs), which are embedded systems containing an FPGA and multiple processing units on a same die. These platforms can be accessed and reconfigured remotely, therefore it is important to protect them from a remote adversary.

This work aims at presenting remote security threats faced by modern FPGA-SoC architectures and techniques for protecting them. To achieve this goal, we review existing and introduce new remote attack vectors for FPGA-SoCs and evaluate their impact on modern FPGA-SoC platforms.

We demonstrate memory and peripherals manipulation attacks through malicious FPGA logic on a modern FPGA-SoC architecture containing state-of-the-art isolation mechanisms. Our attacks can bypass isolation mechanisms such as a Memory Management Unit (MMU), an Input/Output Memory Management Unit (I/O MMU), and a Peripheral Protection Unit (PPU). The implemented attacks are shown to be capable of compromising a Trusted Execution Environment (TEE) based on ARM TrustZone as well as the secure boot mechanism implemented on a Zynq UltraScale+ (ZU+) FPGA-SoC from Xilinx.

The second attack type considered in this thesis are remote fault attacks. By using the shared Power Distribution Network (PDN) contained on some FPGA-SoC platforms, we were capable of generating a voltage drop via malicious logic that can trigger faults on a software executed in the embedded processor core contained in an FPGA-SoC.

In order to perform the previous mentioned attacks, the insertion of malicious logic and its activation are necessary. For the FPGA-SoCs used in this thesis, the manufacturer considers the reconfiguration interfaces as trusted under secure boot assumption. This leads to un-authenticated bitstreams load being possible even after secure boot of a device, which eases the process of malicious logic insertion. Concerning the malicious logic activation, we implemented a Central Processing Unit (CPU) to FPGA power covert channel which achieves a transmission rate of 16.7 kbit/s together with a 2.3% bit error rate.

Finally we demonstrate how the usage of a TEE based on ARM TrustZone can improve the security of FPGA-SoCs. By using a TEE, we restrict the capabilities and access to the bitstream reconfiguration interfaces which complicates the malicious logic insertion. We also implemented a hybrid hardware/software Trusted Platform Module (TPM) which can be used for complementing security mechanisms such as secure boot.

Kurzfassung

In den vergangenen zehn Jahren hat die Popularität der hardwarebasierten Beschleunigung aufgrund der erzielten Leistung in Bereichen wie maschinelles Lernen, Kommunikation und Sicherheit zugenommen. Aufgrund ihrer Flexibilität und Leistungseffizienz sind Field Programmable Gate Arrays (FPGAs) eine interessante Hardware-Beschleunigungsplattform, die heutzutage sowohl im Bereich des Cloud Computing als auch in eingebetteten Systemen eingesetzt werden.

In dieser Doktorarbeit betrachten wir FPGA System on Chips (FPGA-SoCs), d.h. eingebettete Systeme, die ein FPGA und mehrere Verarbeitungseinheiten auf einem Chip enthalten. Auf diese Plattformen sind Fern-Zugriff und Fern-Rekonfiguration möglich. Aus diesen Gründen ist es wichtig, FPGA-SoCs vor einem Fern Angreifer zu schützen. Diese Arbeit identifiziert, Sicherheitsbedrohungen, denen moderne FPGA-SoC-Architekturen ausgesetzt sind, und Techniken um diese Systeme zu schützen. Um dieses Ziel zu erreichen, überprüfen wir bestehende Remote-Angriffsvektoren für FPGA-SoCs und stellen neue vor. Wir bewerten außerdem ihre Auswirkungen auf moderne FPGA-SoC-Plattformen, die modernste Isolationsmechanismen auf Systemebene enthalten.

Wir demonstrieren Angriffe zur Manipulation von Speicher und Peripherien durch bösartige FPGA-Logik auf einer FPGA-SoC-Architektur mit modernsten Isolationsmechanismen. Unsere Angriffe können Isolationsmechanismen wie eine Memory Management Unit (MMU), eine Input/Output Memory Management Unit (I/O MMU) und eine Peripheral Protection Unit (PPU) umgehen. Die implementierten Angriffe sind nachweislich in der Lage, eine Trusted Execution Environment (TEE), sowie den sicheren Bootvorgangs, der auf einem Zynq UltraScale+ (ZU+) FPGA-SoC von Xilinx implementiert ist, zu kompromittieren.

Die zweite in dieser Arbeit betrachtete Angriffsart sind Remote-Fehler-Angriffe. Da viele FPGA-SoCs eine gemeinsame Spannungsversorgung für FPGA und Verarbeitungseinheit (Central Processing Unit (CPU)) nutzen, waren wir in der Lage, über eine bösartige Logik einen Spannungsabfall zu erzeugen, der Fehler in Software einem eingebetteten Prozessorkern generieren konnte.

Um die oben genannten Angriffe auszuführen, ist das Einfügen von bösartiger Logik und deren Aktivierung erforderlich. Bei dem in dieser Arbeit verwendeten FPGA-SoCs betrachtet der Hersteller die Rekonfigurationsschnittstellen als vertrauenswürdig unter der Annahme eines sicheren Bootvorgangs. Dies führt dazu, dass nicht authentifizierte Bitströme auch nach dem sicheren Booten eines Geräts geladen werden können, was das Einfügen bösartiger Logik erleichtert. Was die Aktivierung der böswilligen Logik betrifft, so haben wir einen verdeckten Kanal von CPU zu FPGA implementiert, der eine Übertragungsrate von 16.7 kbit/s zusammen mit einer Bitfehlerrate von 2.3% erreicht.

Schließlich demonstrieren wir, wie die Verwendung einer auf ARM TrustZone basierenden TEE die Sicherheit von FPGA-SoCs verbessern kann. Durch die Verwendung eines TEE schränken wir die Fähigkeiten und den Zugriff auf die Rekonfigurationsschnittstellen ein, was das Einfügen bösartiger Logik erschwert. Wir haben auch ein hybrides Hardware/Software Trusted Platform Module (TPM) implementiert, das zur Ergänzung von Sicherheitsmechanismen wie secure boot verwendet werden kann.

Acknowledgments

First of all I would to thank my advisor Prof. Georg Sigl for giving me the chance to pursue a doctoral degree. His support, faith in my research, and constructive feedback helped me during my whole doctoral project.

I would also like to thank my current and former colleagues from the chair for the great time and productive discussions we had together at the chair.

I'm very thankful to all my co-authors and people who jointly contribute or give constructive feedback on my research: Dr.-Ing. Nisha Jacob, Andreas Zankl, Dr.-Ing. Jonas Krautter, Dr.-Ing. Dennis Gnad, Prof. Mehdi Tahoori, Michael Gruber, and Dr.-Ing. Fabrizio De Santis.

A special thanks also goes to Stefan Wiehler and Andreas Schuler from Missing Link Electronics for the great cooperation we had concerning the usage of OP-TEE for the Zynq UltraScale+ platform.

I'm also thankful to all the students I supervised and especially to Konrad Hohentanner and Robert Kunzelmann for their support to my research.

Finally I would like to thank my family, friends, and Laura for their love, encouragements, and support during these years.

Contents

Acronyms

ACE	AXI Coherency Extension
ACP	Accelerator Coherency Port
AES	Advanced Encryption Standard
AES-GCM	AES Galois Counter Mode
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
APU	Application Processing Unit
ATF	ARM Trusted Firmware
AXI	Advanced eXtensible Interface Bus
BBRAM	Battery Backed RAM
BRAM	Block RAM
CA	Client Application
CCI	Cache Coherent Interconnect
CPU	Central Processing Unit
CSU	Configuration Security Unit
DDR	Double Data Rate
DFA	Differential Fault Analysis
DMA	Direct Memory Access
DMAA	Direct Memory Access Attack
DoS	Denial-of-Service
DRAM	Dynamic Random Access Memory
DRBG	Deterministic Random Bit Generator
dTPM	discrete TPM
DVFS	Dynamic Voltage and Frequency Scaling
EAL	Evaluation Assurance Level
ECDSA	Elliptic Curve Digital Signature Algorithm
eFuse	electronic fuse
EL	Exception Level
eMMC	embedded MultiMedia Card
FEK	File Encryption Key
FIPS	Federal Information Processing Standard
FIQ	Fast Interrupt Request
FPGA	Field Programmable Gate Array
FPGA-SoC	FPGA System on Chip
FSBL	First Stage Boot Loader
FSM	Finite-State Machine

fTPM	firmware TPM
GPIO	General Purpose I/O
GPU	Graphic Processing Unit
HMAC	Hash Message Authentication Code
HT	Hardware Trojan
HUK	Hardware Unique Key
I/O	Input/Output
I/O MMU	Input/Output Memory Management Unit
I2C	Inter-Integrated Circuit
ICAP	Internal Configuration Access Port
IP	Intellectual Property
IRQ	Interrupt Request
ISA	Instruction Set Architecture
IV	Initialization Vector
LPD	Low Power Domain
LSB	Least Significant Bit
LUT	Look-Up Table
MC	MixColumns
MMU	Memory Management Unit
MPSoC	Multiprocessor System on a Chip
MPU	Memory Protection Unit
NIST	National Institute of Standards and Technology
NoW	Normal World
OCM	On-chip Memory
OP-TEE	Open Portable Trusted Execution Environment
OS	Operating System
PC	Program Counter
PCAP	Processor Configuration Access Port
PCR	Platform Configuration Register
PDN	Power Distribution Network
PMU	Platform Management Unit
PMU-FW	Platform Management Unit Firmware
PoC	Proof of Concept
PPK	Primary Public Key
PPU	Peripheral Protection Unit
PRNG	Pseudo-Random Number Generator
PS	Processing System
PSP	Platform Security Processor
PSU	Power Supply Unit

PTE	Page Table Entry
PTT	Platform Trust Technology
PUF	Physical Unclonable Function
REE	Rich Execution Environment
RO	Ring Oscillator
ROM	Read Only Memory
RootFS	Root File System
RPMB	Replay Protected Memory Block
RPU	Real-Time Processing Unit
RSA	Rivest Shamir and Adleman
RTOS	Real Time Operating System
SB	SubBytes
SCA	Side-Channel Analysis
SCR	Secure Configuration Register
SCU	Snoop Control Unit
SDM	Security Device Manager
SeW	Secure World
SFTP	SSH File Transfer Protocol
SGX	Software Guard Extensions
SHA-3	Secure Hash Algorithm 3
SIM	Subscriber Identity Module
SMC	Secure Monitor Call
SMMU	System Memory Management Unit
SoC	System on Chip
SPK	Secondary Public Key
SR	ShiftRows
SRAM	Static Random Access Memory
SSK	Secure Storage Key
TA	Trusted Application
TCG	Trusted Computing Group
TCM	Tightly Coupled Memory
TDC	Time-to-Digital Converter
TEE	Trusted Execution Environment
TLB	Transaction Look-aside Buffer
TPM	Trusted Platform Module
TRNG	True Random Number Generator
TSK	Trusted Application Storage Key
USB	Universal Serial Bus
VIC	Vivado Isolation Configuration
VM	Virtual Machine
XMPU	Xilinx Memory Protection Unit
XPPU	Xilinx Peripheral Protection Unit

ZU+	Zynq UltraScale+

List of Figures

List of Tables

1 Introduction

This chapter presents the context, scope, objectives, and contributions of this thesis. After a brief introduction to FPGAs as a flexible hardware acceleration platform for the cloud as well as embedded systems, it highlights security challenges resulting from their usage in heterogeneous computing systems such as FPGA-SoCs. This chapter ends with the description of the outline adopted for this thesis.

1.1 Motivation and Thesis Context

In the area of information, artificial intelligence is reshaping technology and bringing new concepts such as self driving cars, intelligent industrial infrastructures, and smart healthcare technologies to the market. According to OpenAI [3], the computing demand for artificial intelligence is doubling every 3.4 months since 2012. In parallel, we have seen a slower increase in the performance of processors with technology limitations preventing scaling of frequency due to reliability, power consumption, and heating problems. To cope with the end of Moore's law [75], new computation paradigms relying on application specific accelerators as an extension to a general purpose processor have emerged. One of the new computing schemes consists of using an FPGA as a hardware acceleration extension card that is added to a classical computer as an alternative to Graphic Processing Unit (GPU) based acceleration. Through their reconfigurability, FPGAs have the advantage of being a flexible computation platform that can be used for accelerating a wide range of algorithms in hardware. Their good performance and power efficiency have lead to FPGA based acceleration becoming trendy in the cloud computing. Companies such as Amazon, Alibaba Cloud, and Baidu have now integrated the renting of FPGAs as well as the selling of Intellectual Property (IP) cores for FPGA based acceleration in their catalog. Besides the cloud computing domain, FPGAs are also beneficial to the edge computing domain by integrating them in so-called System on Chips (SoCs). FPGAs are today particularly found in FPGA-SoCs, which correspond to SoCs that integrate one or more embedded processor cores together with an FPGA on the same chip. Computation on these platforms usually takes advantage of a hardware/software co-design approach, with complex software running on an embedded processor core and the accelerators deployed in the reconfigurable logic. These systems also take advantage of a flexibility in software as well as in hardware and can be reconfigured remotely.

Due to their wide usage in today's systems, understanding the security of heterogeneous systems based on FPGAs is crucial. While FPGA security has been intensively studied by researchers, most of the demonstrated attacks were performed on platforms where FPGAs were considered as isolated systems with an attacker that has physical proximity to a device [76, 37, 113, 13, 66]. With the current integration of FPGAs in complex heterogeneous computing systems that are connected to the Internet, FPGA security has become a more complex topic. In this thesis, we particularly consider the edge scenario and the security threats specific to FPGA-SoC platforms.

Previous work have shown that integrating an FPGA and multiple processing units together on the same chip lead to new attack vectors. Among these attack vectors, FPGA enabled attacks through malicious logic present a crucial threat to the security of FPGA-SoCs. These attacks can notably bypass isolation mechanisms that are typically implemented in the Operating System (OS) with the help of an MMU [46]. Moreover, FPGA logic can be used for performing remote side-channel and fault attacks, where a laboratory setup is usually required. This is achieved by using dedicated circuits that can sense the variations of a Power Distribution Network (PDN) or stress it, so that a singnificant voltage drop can be generated [28, 56].

These attacks usually assume the presence of a malicious functionality hidden in a third party IP core, which is referred to as a Hardware Trojan (HT). This threat is of high relevance in a hardware/software co-design approach with closed source FPGA IP cores that can be obtained from an online market place. Alternatively, an attacker capable of executing software on an FPGA-SoC and of partially reprogramming an FPGA-SoC with her own logic is commonly considered as a possible threat model for FPGA enabled attacks [28, 56].

1.2 Objectives

The main goal of this thesis is the the impact evaluation of FPGA enabled attacks on the security of FPGA-SoCs as well as the presentation of techniques to mitigate such threats. To achieve this goal, following objectives should be addressed:

- Understanding and classification of remote threats faced by modern FPGA-SoCs.
- Impact assessment of these threats through the development of attacks that can compromise FPGA-SoCs' security.
- Development and evaluation of protection techniques against remote attacks in FPGA-SoCs.

1.3 Contributions

In order to address the objectives presented in section 1.2, this thesis contains following contributions related to the field of FPGA-SoC security.

1.3.1 Comprehensive Analysis of FPGA-SoCs Security from a System Level Point-of-View

An analysis of existing attacks, security mechanisms, and trust assumptions made by FPGA-SoCs manufacturer's is necessary for understanding the security shortcomings of modern FPGA-SoCs and identify new attack vectors. This analysis leads to the identification of the relevant attack vectors affecting the FPGA-SoCs considered in this thesis that we present in chapter 2.

Through this analysis, we identify the possibility of reading back an FPGA configuration after an encrypted bitstream load that occured during secure boot. This capabilitiy can potentially lead to IP theft issues. Surprinsingly, it is also possible to reprogram some FPGA-SoCs with an un-authenticated bitstream even after a first authenticated bitstream load that occured during secure boot [101]. Both security vulnerabilities are exploitable by software running at user privilege level [102], which makes such attacks realistic. The second attack vector in particular, contributes to make the insertion of malicious logic easier for an attacker. This vulnerability can be used for the FPGA enabled attacks that we introduce in the chapters 3, 4, and 5 of this thesis.

1.3.2 Memory and Peripherals Manipulation Attacks on Modern FPGA-SoCs

Using FPGA logic for memory manipulation is beneficial for bypassing memory isolation mechanisms implemented with the help of an MMU and the OS. This type of attack was notably used for breaking secure boot on a Zynq-7000 FPGA-SoC [45, 46]. Such attacks are however harder to perform on modern FPGA-SoC platforms (Xilinx ZU+ [107] or the Intel Stratix 10 [43]) that integrate an I/O MMU, as this component can extend the memory isolation principle of a standard MMU for Input/Output (I/O) devices and accelerators.

In chapter 3 we demonstrate the possibility of performing such attacks on the ZU+ and extend the attack principles to the manipulation of peripherals. The attacks demonstrated in chapter 3 were capable of altering the secure execution of software in a TEE and the secure boot of the ZU+.

1.3.3 Faulting the Execution of Software on an FPGA-SoC's CPU through FPGA Logic

In the recent years, we have seen that physical attacks such as Side-Channel Analysis (SCA) [26, 111, 85] and fault attacks [25, 56, 4] can be performed remotely on FPGA platforms by implementing voltage sensors and power-hammering circuits within the FPGA logic. While this class of attack were principally considered in a cloud computing setup with multiple tenants using the same FPGA, such attacks are also relevant in the FPGA-SoC scenario. In the single tenant FPGA-SoC context, malicious logic could be used as an attack vector to compromise the execution of software in one processing unit of an FPGA-SoC [28].

In chapter 4, we demonstrate the possibility of faulting the execution of software on one of the FPGA-SoC's CPU cores by using a power-hammering circuit. We showcase the possibility of skipping addition and multiplication instructions executed on an ARM Cortex-A9 CPU, as well as a Differential Fault Analysis (DFA) on a software implementation of Advanced Encryption Standard (AES).

1.3.4 Implementation of a Power Covert Channel between a CPU and an FPGA

In addition to the insertion of malicious logic inside an FPGA-SoC, the attack vectors introduced in sections 1.3.3 and 1.3.5 require the activation of the attacker logic through a covert communication channel so that the attack primitive can remain stealthy. For this purpose, we implemented a CPU to FPGA power covert channel which achieves a transmission rate upto 16.7 kbit/s together with a 2.3% bit error rate. The implementation and characterization of the covert channel are presented in chapter 5.

1.3.5 Protection of FPGA-SoCs via the Usage of a Trusted Execution Environment and a Hybrid-TPM

Due to the security breaches which can be achieved from malicious logic insertion, protecting FPGA-SoCs from this threat is crucial. In this thesis, we show how the usage of a TEE based on ARM TrustZone can help in improving FPGA-SoC's security. The proposed approach consists in restricting the access and capabilities to the bitstream reconfiguration interfaces by incorporating the FPGA reconfiguration framework in a TEE. Moreover, we enhance a software implementation of a TPM with hardware cryptographic accelerators and an entropy source that is derived from on-chip Static Random Access Memory (SRAM) start-up patterns.

1.4 Outline

The rest of this thesis is organized as follows:

- Chapter 2 first introduces the FPGA-SoC architectures considered in thesis as well as some of the security mechanisms they contain. We also explain the concepts of TEE and TPMs and clarify how these technologies can be beneficial to FPGA-SoCs. Chapter 2 finally provides a general overview of the security threats faced by FPGA-SoCs and a more detailed explanation of the attack vectors that are exploited in this thesis.
- Chapter 3 describes attacks capable of bypassing memory and peripherals isolation mechanisms on FPGA-SoCs via malicious FPGA logic.
- In chapter 4, we consider a more generic attacker model and describe fault attacks where FPGA logic can alter the execution of software on an FPGA-SoC's CPU.
- The next chapter describes the implementation of a CPU to FPGA power covert channel in FPGA-SoCs and its application as activation function of hardware trojans.
- Chapter 6 demonstrates the security profits that can be obtained through the usage of a TEE and a hybrid-TPM on FPGA-SoCs.
- Finally, chapter 7 concludes this thesis.

2 Background

In the following we introduce the general architecture of FPGA-SoCs and present the two architectures used in this thesis in section 2.1. Section 2.2 presents state-of-the-art security mechanisms of FPGA-SoCs before, during, and after the boot process. This is followed by a description of the software architecture for trusted code execution on FPGA-SoCs in section 2.3. Finally, section 2.4 provides an analysis of the security threats faced by FPGA-SoCs that we put into perspective with the security mechanisms introduced in section 2.2. Furthermore it explains the attack vectors exploited in this thesis.

2.1 FPGA-SoC Architecture

FPGAs are popular platforms for accelerating computations in hardware. Due to their good computational power combined with a low power consumption, these platforms have a great popularity in the embedded world and are now widely adopted in so-called FPGA-SoCs. The general architecture of an FPGA-SoC is depicted in figure 2.1.

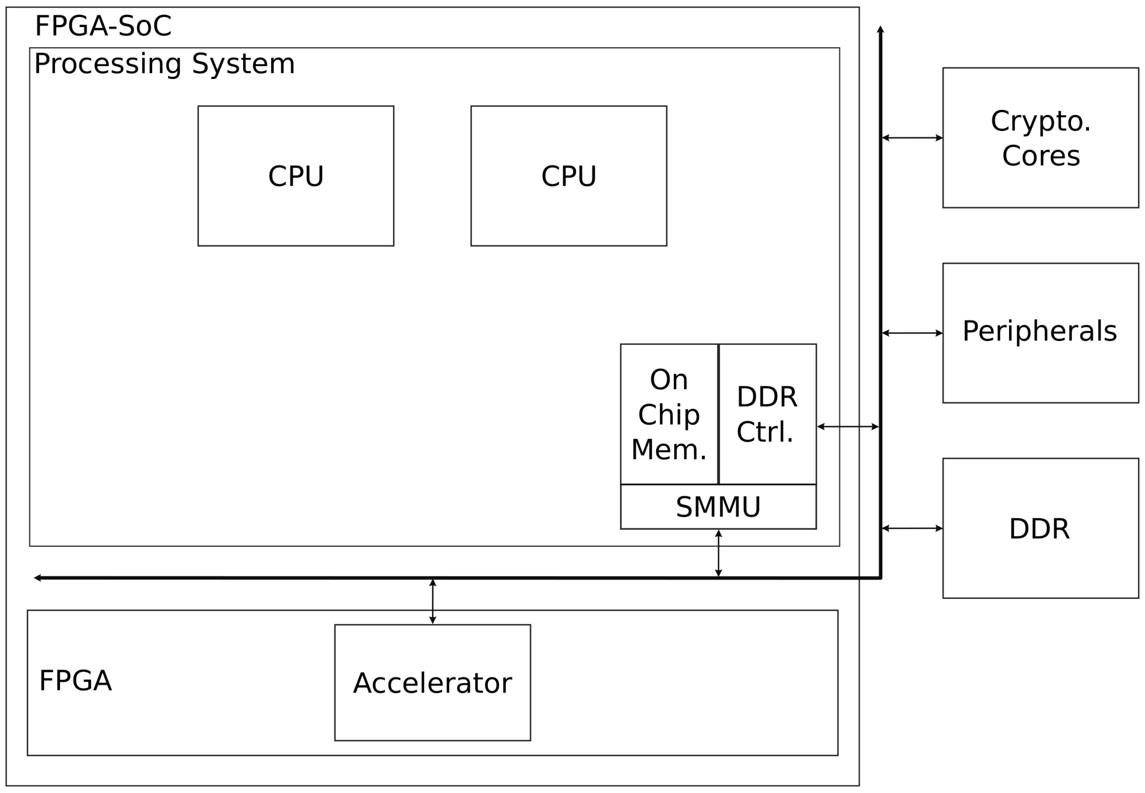


Figure 2.1 FPGA-SoC architecture

It consists mainly of an FPGA where dedicated hardware accelerators can be used and multiple processing units. These two main blocks are associated with On-chip Memory

(OCM) for storing sensitive data on chip as well as a larger external Double Data Rate (DDR) memory. A set of peripherals for high-speed Direct Memory Access (DMA), cryptographic acceleration, and I/O communication are also contained inside the FPGA-SoC. The different processing units and the memory subsystem are interconnected together through the ARM Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface Bus (AXI) bus system.
The work presented in this thesis relies on two different FPGA-SoC architectures from Xilinx: the Zynq-7000 [108] and the ZU+ [107] which we introduce in more details below.

Zynq-7000: The Zynq-7000 FPGA-SoC contains a dual-core ARM Cortex-A9 CPU and a Xilinx 7-series FPGA. Each processor core has a 32 kB L1 instruction and data cache and a unified 512 kB L2 cache that is shared between the processor cores. The memory subsystem, which is accessible via the AXI bus system consists of a 256 kB OCM and a larger external DDR memory. I/O communication is supported through Inter-Integrated Circuit (I2C), Ethernet, and Universal Serial Bus (USB) interfaces. The Zynq-7000 also includes a secure boot mechanism which is backed by an AES and Hash Message Authentication Code (HMAC) accelerators and a secure key storage inside the device's Battery Backed RAM (BBRAM) or electronic fuses (eFuses). In this thesis, we use the PYNQ-Z1 board as a Zynq-7000 compliant architecture. For this board, as well as for other boards that use the Zynq-7000 SoC [111, 28], the PDN of the FPGA-SoC is shared between the ARM Cortex-A9 and the FPGA. The sharing of the PDN is the foundation for the experiments performed in the chapters 4 and 5 of this thesis.

Zynq UltraScale+: The ZU+ architecture is the modern version of Xilinx FPGA-SoCs. The ZU+ architecture is depicted in figure 2.2. In this thesis, the EG variant of this architecture is used. It consists of a quad-core ARM Cortex-A53 denoted as Application Processing Unit (APU), an ARM Mali-400 GPU, a dual-core ARM Cortex-R5 Real-Time Processing Unit (RPU), and an FPGA. The Multiprocessor System on a Chip (MPSoC) includes 256kB OCM and a larger external DDR memory, both of which can be accessed with the ARM AMBA AXI bus and the ARM CoreLink Cache Coherent Interconnect (CCI). Another key component is the Platform Management Unit (PMU), which is responsible for power management and monitoring of system components. In comparison to the Zynq-7000, the ZU+ contains more mechanisms for isolation(an System Memory Management Unit (SMMU), eight Xilinx Memory Protection Units (XMPUs), and an Xilinx Peripheral Protection Unit (XPPU)) as well as security features which are integrated inside the Configuration Security Unit (CSU). This component is mainly responsible for the secure boot of the device through the usage of hardware cryptographic accelerators and a secure key storage inside the FPGA-SoC (see section 2.2.2.1 for further details concerning secure boot on the ZU+). The CSU is further used for tamper monitoring and response and programming of the FPGA fabric via the Processor Configuration Access Port (PCAP). Besides the CSU, the ZU+ also contains isolation mechanisms such as a SMMU, XMPUs, and ARM TrustZone technology. In the next section, detailed information concerning the before mentioned security mechanisms are introduced.

2.2 Security Mechanisms of FPGA-SoCs

Security on FPGA-SoCs relies on protection mechanisms before, during, and after the boot process [1, 42]. These mechanisms essentially consist of a secure key storage inside

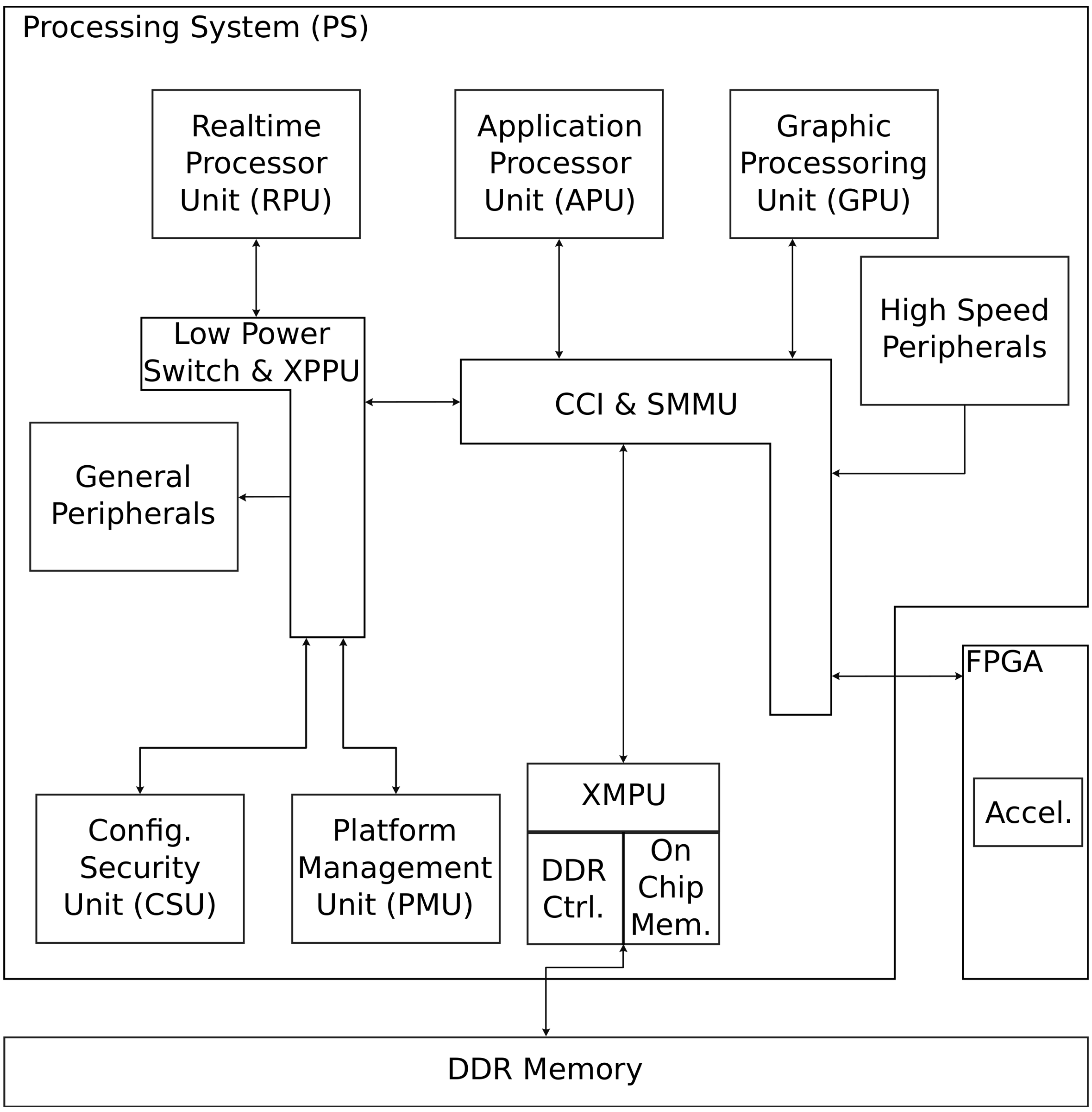

Processing System (PS)
Realtime Processor Unit (RPU)
Application Processor Unit (APU)
Graphic Processoring Unit (GPU)
High Speed Peripherals
Low Power Switch & XPPU
CCI & SMMU
General Peripherals
FPGA
Accel.
XMPU
Config. Security Unit (CSU)
Platform Management Unit (PMU)
DDR Ctrl.
On Chip Mem.
DDR Memory

Figure 2.2 Architectural representation of the Zynq UltraScale+ FPGA-SoC

the device, a secure boot process that is performed with device specific keys and isolation as well as tamper-protection mechanisms that protect the device during runtime. In this section, we introduce security mechanisms found on modern FPGA-SoCs such as the Xilinx ZU+ [107] or the Intel Stratix 10 [43].

2.2.1 Pre-boot Security Mechanisms

Pre-boot security mechanisms on FPGA-SoCs consist of techniques for protecting the cryptographic keys inside the device, while it is powered-off. To achieve this, FPGA-SoC manufacturers include secure non-volatile storage mediums such as eFuses and BBRAM. For security purposes, these memories are only writable and cannot be read back via software. In contrast to eFuses, a BBRAM can be programmed multiple times but requires an external battery to keep its content while the FPGA-SoC is powerred-off.

An AES key for decryption and authentication of a software is usually stored in one of these Non-Volatile Memorys (NVMs). Alternatively, for a secure boot process based on signature verifications, the hash value of a public key can be stored in the eFuses for an integrity check of the public key before its use for authentication.

On the ZU+, a 256-bit AES key can be stored in eFuses or BBRAM. This key can also be stored in an obfuscated way, by encrypting it with a key derived from the device's Physical Unclonable Function (PUF). Besides the storage of an AES key, the eFuses can also be used for storing the hash of two Rivest Shamir and Adleman (RSA) Primary Public Keys (PPKs) and a Secondary Public Key (SPK) number, which are used in the hardware root of trust secure boot scheme (see section 2.2.2.1). The PPK hashes as well as the SPK number can also be revoked, for preventing the authentication of a boot image containing an unpatched security vulnerability in the context of a rollback attack.

2.2.2 Secure Boot Implementation on Modern FPGA-SoCs and Possible Extensions Achieved through the Usage of TPMs

Secure boot is a fundamental security feature to ensure the integrity and authenticity of the hardware and software loaded during the boot process. This typically involves a signature verification of all the components started during boot with a prior hash verification of the public keys. Alternatively, an authentication of the boot image with symmetric authenticated encryption scheme such as AES Galois Counter Mode (AES-GCM) is possible. In this section we first introduce the secure boot mechanism of the ZU+. In a second time we introduce the notion of TPM and the security extensions it can provide to a secure boot process.

2.2.2.1 Secure Boot Implementation on Modern FPGA-SoCs

Secure boot on modern FPGA-SoCs is usually implemented with the help of a special security co-processor such as the CSU [1] on the ZU+ or the Security Device Manager (SDM) [42] on the Intel Stratix 10. In this section, the secure boot process of the ZU+ is introduced more in details, as this security concept is relevant for the chapters 3 and 6 of this thesis.

Two secure boot options are available on the ZU+: hardware root of trust and encrypt only. The encrypt only secure boot relies on an authentication of the boot partitions with AES-GCM but has been shown to be vulnerable to boot header manipulation attack, due to the absence of partition headers authentication [20]. Therefore, Xilinx recommends the usage of the hardware root of trust secure boot, which relies on an RSA authentication of the boot image together with a hash verification of the RSA public keys involved for the authentication. This secure boot modus is depicted in figure 2.3.

Upon startup of the ZU+, a hardware state machine performs some verification tests and compares a Secure Hash Algorithm 3 (SHA-3)-384 digest of the PMU Read Only Memory (ROM) with a value that is stored inside the device. If the two values match, the PMU ROM is executed, which leads to some early device initialization and a SHA-3-384 digest check of the CSU ROM before the CSU gets released. The CSU is then responsible for authenticating an RSA PPK. This step is performed by loading the PPK value inside OCM and comparing its SHA-3-384 digest with a value that is stored in the device's eFuses. Two PPK values can be used for secure boot with the boot image header specifying which value should be used for authentication. If the computed PPK digest

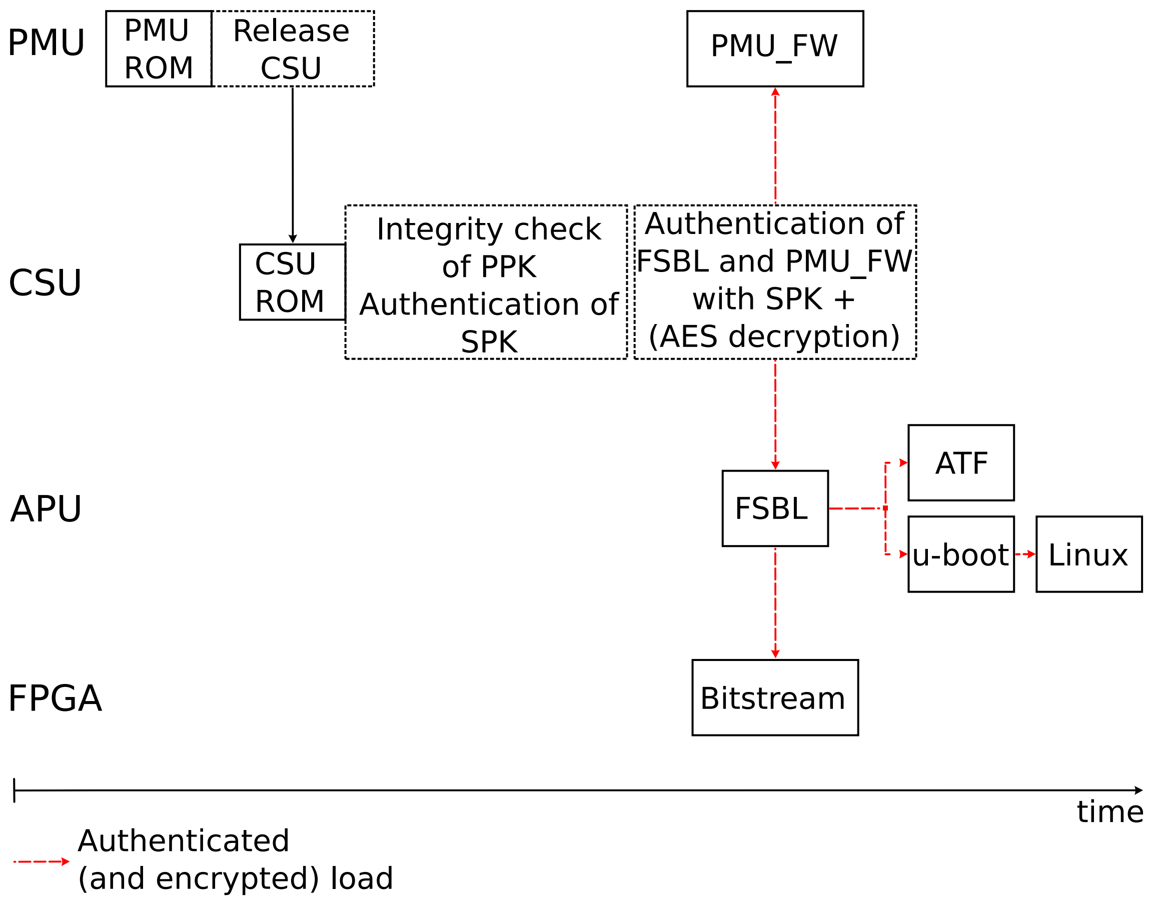

PMU
PMU ROM
Release CSU
PMU_FW
CSU
CSU ROM
Integrity check of PPK Authentication of SPK
Authentication of FSBL and PMU_FW with SPK + (AES decryption)
APU
FSBL
ATF
u-boot
Linux
FPGA
Bitstream
time
Authenticated (and encrypted) load

Figure 2.3 Hardware root of trust secure boot on the Zynq UltraScale+

value match with what is stored in the eFuses, this key can be further used for authenticating an SPK. The tuple {SPK, SPK_{ID}, $\text{SPK}_{\text{signature}}$} are contained inside the boot image. The SPK authentication is performed by firstly comparing the SPK_{ID} with a value stored in the eFuses, secondly the SPK signature is computed by using one of the two PPKs and the computed signature is checked against the $\text{SPK}_{\text{signature}}$ value that is stored in the boot image. If the SPK authentication succeeds, this key can be further used for authenticating the First Stage Boot Loader (FSBL) and the Platform Management Unit Firmware (PMU-FW), which may be optionally encrypted with AES-GCM 256. The AES key used for decryption is referred as the device key; and it can be retrieved via the device's PUF, eFuses or BBRAM depending on what is specified in the boot header.

The APU gets released after the FSBL authentication. The FSBL performs the early processor initialization and authenticates the subsequent partitions with the SPK, namely the bitstream, ARM Trusted Firmware (ATF), and u-boot. If confidentiality of the boot image is also wished, those components can be once again optionally encrypted with the device key. Once the FSBL has finished its work, u-boot gets executed. This partition completes the boot chain by authenticating and optionally decrypting the OS.

To protect the secure boot process from SCA, Xilinx has additionally built a protocol based *key rolling* security feature within the AES-GCM core [101]. This mechanism consists of limiting the number of side-channel data that an attacker can collect under a specific key. Although this mechanism makes SCA more difficult, a reduction of the key space has been shown possible via a sophisticated attack on the first five AES rounds [37]. Based on their analysis, the authors recommends a change of the key after the encryption of 20 to 30 data blocks. Using a *key rolling parameter* in this range should offer sufficient protection against state-of-the-art and upcoming SCA.

2.2.2.2 Extension of Secure Boot Achievable through the Usage of a TPM

Secure boot enables the build of a chain-of-trust through an authentication and integrity check of all the components loaded during the boot process of an FPGA-SoC. As explained in section 2.2.2.1, this process relies on several RSA signature verifications that are performed together with a hash verification of an RSA public key stored inside the eFuses. For advanced security options such as remote attestation, in which a remote verifier ensures that a device has booted in an intended state, TPMs are typically used.

TPMs are dedicated tamper protected security chips aiming at providing hardware security features to a computer system. We refer to this type of TPMs as discrete TPMs (dTPMs) for the rest of this thesis. From a functional point of view, dTPMs can be viewed as security chips with support for cryptographic primitive in an isolated execution environment and secure storage of cryptographic keys. Their main advantage resides in a physical isolation, a protection against physical attacks, and an availability in the early boot phase of an embedded system. The security guarantees of TPMs are further reinforced through their certification with security standards such as Common Criteria Evaluation Assurance Level (EAL) 4+ or Federal Information Processing Standard (FIPS) 140-2. The main applications of TPMs are protection of disk encryption keys, remote attestation, and secure cryptographic key storage. The Trusted Computing Group (TCG) has defined a TPM 2.0 specification to replace the insecure TPM 1.2 specification. In contrast to TPM 1.2, the TPM 2.0 standard offers support for more secure cryptographic primitives and has a reference implementation of its software stack that we described in section 2.3.2. According to this specification, a TPM should provide support for symmetric/asymmetric cryptography and hashing. It should contain a true random number generator which is used to generate cryptographic keys and have a secure storage capability for those keys.

One important component of a TPM are the so-called Platform Configuration Registers (PCRs), which are used to monitor the state of a system via hash values. These PCRs can be used to implement measured boot, where each of the components contained in the boot of a system are measured via hash values. These hash values are then going to be signed by the TPM and sent to a remote verifier. The next step is a comparison of the platform measurements with reference values to ensure that the system is executing an intended configuration.

An additional security feature which can complement secure boot is cryptographic key sealing. In this scenario, a TPM measures all the components involved in the boot chain and stores the measurements inside PCRs. Once the secure boot process is completed, a cryptographic key sealed with PCR values corresponding to a given secure boot configuration is accessed. This process only succeeds with the PCR values reflecting the secure boot configuration. Chapter 6 presents a cryptographic key sealing implementation achieved with the hybrid-TPM that is introduced in the same chapter.

2.2.3 Runtime Security of Moden FPGA-SoCs

Once an FPGA-SoC has securely booted, runtime security mechanisms further protect the system from potential attacks. Generally these mechanisms rely on isolation between different processing units of an FPGA-SoC or within a same processing unit. In this section, we first introduce security mechanisms for isolating memory and peripherals. These mechanisms play a vital role for ARM TrustZone [6] technology that we introduce in a second time. Finally we present some tamper detection and response mechanisms that are available on FPGA-SoCs.

2.2.3.1 Memory and Peripherals Isolation/Access Restriction

Isolation and access restriction of memory and peripherals is a key practice for achieving security and safety. By isolating the memory between processes and among processing units, the impact of one component on the rest of the system is limited. Similarly, restricting the access to a peripheral prevents the usage of this peripheral for malicious purposes. Achieving separation and access restriction on FPGA-SoCs rely on a combination of hardware components and support of the OS. In the following, we introduce the main components available in FPGA-SoCs to reach this goal.

MMU: The MMU is a key component that enables the usage of virtual addresses by an OS. Its purpose consists in translating virtual addresses as seen by the OS into physical addresses. By using virtual addresses, the OS is able to work with a bigger memory address space than what is physically available on the device. For the OS, the memory space is divided in so-called memory pages. The virtual to physical address translation process consists in finding the physical page number that corresponds to a virtual page. This mapping is stored in the physical memory inside Page Table Entrys (PTEs). To speedup the virtual to physical address translation, the recently used PTEs are stored inside the Transaction Look-aside Buffer (TLB) which acts as a cache memory.
Besides address translation, an MMU is also responsible for managing the ownership and access-rights of memory pages. These two features contribute to the memory isolation between processes, users, and Virtual Machines (VMs). An MMU can however only protect the memory which is accessed by a processor. To protect the memory from I/O devices such as a DMA peripheral or an FPGA connected as acceleration card, an I/O MMU should be used.

I/O MMU and SMMU: The I/O MMU is an extension of an MMU for I/O devices; address translation for CPU transactions are still handled by a classical MMU while address translation for I/O devices is taking care of by the I/O MMU with the OS support concerning the translation tables maintenance. Thereby, an I/O MMU handles memory translation for I/O devices and extends memory protection to peripherals.
In the context of modern FPGA-SoCs that contains several ARM processing units, the I/O MMU is also referred as an SMMU [7]. An SMMU operates with a two-stage address translation. The first stage, consists in translating virtual addresses into intermediate physical addresses and is used for providing isolation within the OS. The second stage maps the intermediate physical addresses into physical addresses and is typically used for virtualizing DMA devices between VMs.

Memory Protection Unit (MPU) and XMPU: In contrast to MMUs and I/O MMUs, an MPU only handles memory protection and does not offer virtualization. An MPU operates by defining memory regions and access permissions of masters to these memory regions. On the ZU+ architecture, eight XMPUs work in collaboration with the SMMU to offer memory protection (DDR, OCM). These units operate in a similar way as a traditional MPU and additionally support ARM TrustZone technology (see section 2.2.3.2) for defining a memory region as secure or non-secure.

PPU and XPPU: A PPU is another important isolation feature that is usually available in embedded systems for restricting the access to some peripherals and configuration registers. On the Xilinx FPGA-SoCs considered in this thesis, this feature is referred as XPPU. The XPPU operates similarly as the XMPUs except that the concept of memory regions are replaced by appertures. In the Xilinx terminology, an apperture refers to a set of register addresses and the apperture permission list defines the masters access rights to a given apperture. Like the XMPUs, ARM TrustZone is also supported, which can be used for enabling a further access restriction to secure masters only.

2.2.3.2 System Level Isolation with ARM TrustZone Technology

In addition to the memory isolation mechanisms introduced in section 2.2.3.1, ARM TrustZone provides supplementary separation mechanisms by building hardware-based isolation directly in the CPU. In this thesis, we consider the usage of this technology specifically for Cortex-A process variants using the ARMv7-A or ARMv8-A Instruction Set Architecture (ISA) and exclude the variant of this technology for ARMv8-M microcontrollers.

The separation is achieved through partitioning of the hardware resources (registers, memory, and caches) between two distinct execution environments: the Normal World (NoW) and the Secure World (SeW) (see figure 2.4). The resources tagged as secure, can only be accessed when the ARM processor is executing in the SeW. The Secure Configuration Register (SCR) reflects the world in which the processor is currently running. A world switch is possible through a Secure Monitor Call (SMC) which is going to be handled by a Secure Monitor contained inside ATF, the reference implementation of the Secure Monitor software. ATF is executing at Exception Level (EL) 3 and is protected from other system components executing at a lower EL. ARM TrustZone enables a further system-wide isolation by defining two distinct interrupt sources (Fast Interrupt Request (FIQ) for the SeW and Interrupt Request (IRQ) for the NoW). Interrupts triggered on the FIQ source can only be handled in the SeW and similarly IRQ interrupts are handled in the NoW. Finally it is possible to statically or dynamically tie I/O peripherals to a specific world. ARM TrustZone is extensible to the FPGA fabric through the usage of a security bit and the AXI Interconnect. A master can generate a secure transaction by setting the security bit to 0, otherwise the transaction is non-secure. The AXI Interconnect enables the protection of slaves by configuring them as secure or non-secure. A secure slave can only be accessed by a secure master transaction, while a non-secure slave is accessible by both secure or non-secure master transactions.

Combining secure boot, ARM TrustZone, and the isolation mechanisms presented in section 2.2.3.1 enable the conception of a TEE, a security standard for isolated and trusted software execution defined by Global Platform [23]. We introduce the requirements for a TEE as well as the software architecture it relies on in section 2.3.1.

2.2.3.3 Tamper Detection and Response Mechanisms

Tamper detection mechanisms are vital for ensuring that an FPGA-SoC is used in normal operating conditions after a secure startup. If that is not the case, an attacker might be able to bypass some security features, in the context of a fault attack. For that purpose, Xilinx has equiped the ZU+ with temperature and voltage sensors. If the FPGA-SoC is operated outside of normal environmental conditions, the CSU can respond by putting the FPGA-SoC in a secure-lockdown state, triggering an interrupt, resetting the system or erasing the BBRAM key [101]. However, these on-chip temperature and voltage sensors are not suitable for the detection of fast voltage drops which can occur during a fault attack. For these attacks, external voltage and temperature sensors should be used [101] or alternatively implemented within the FPGA logic [82].

2.3 Software Architecture for Trusted Code Execution on FPGA-SoCs

In section 2.2, we introduced security mechanisms which protect an FPGA-SoC before, during, and after the boot process. In this section, we make a special focus on the software architecture that is used during the boot and runtime of an FPGA-SoC towards the achievement of trusted execution. We first introduce the notion of TEE and the underlying architecture it relies on. In a second time we present the notion of firmware TPM (fTPM), which is a TPM implemented inside a TEE. We complete the presentation of fTPM with an introduction to the TPM 2.0 software architecture.

2.3.1 Trusted Execution Environment

ARM TrustZone enables the use of a TEE [23], a standard supported by Global Platform for trusted and isolated software execution. Global Platform requires following security functionalities for the TEE standard [24]:

- Authentication of all software running in a TEE before its execution.
- Integrity and confidentiality of all TEE assets which is assured through cryptography and isolation.
- Random number generation and derivation of keys and key pairs for asymmetric cryptography.
- Protection of TEE code and secret assets such as cryptographic keys from unauthorized tracing and control.
- Protection against downgrade of TEE firmware.

The architecture of a TEE is depicted in figure 2.4. The TEE is running in an isolated execution environment, in parallel to a standard OS which is executed in a so-called Rich Execution Environment (REE). In contrast to an REE, only authenticated and unaltered security critical software is meant to be executed inside a TEE. This ensures that the trusted computing base is kept as small as possible. To interact with the TEE, the REE Kernel is enhanced with a TEE Driver. NoW Client Applications (CAs) use the Global Platform TEE Client Application Programming Interface (API) to communicate with the trustlets or Trusted Applications (TAs). This API enables the transfer of input and output parameters between a CA and a TA.

The TAs are often obtained from third parties software sources. To guarantee the integrity and authenticity of the trustlets, a signature verification is performed in the TEE before their actual execution. If the signature verification is successful, the trustlets are executed at EL0 (user mode). Trustlets use the Global Platform TEE Internal Core API to access to the EL1 (kernel mode) trusted OS functions such as cryptography and secure storage.

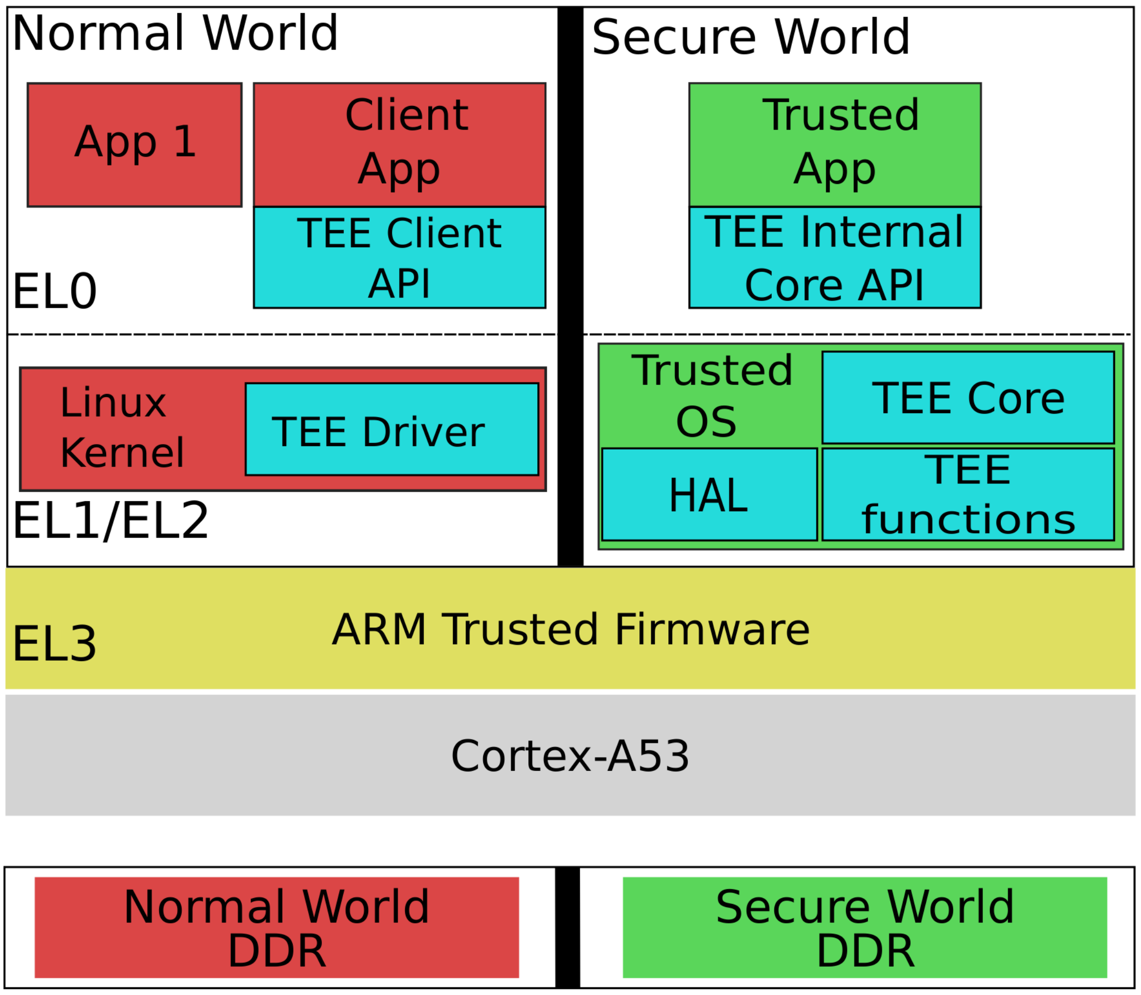


Figure 2.4 TEE Platform architecture

2.3.2 Firmware Implementation of TPMs and TPM-Software Architecture

With the release of Windows 11, TPMs 2.0 have become mandatory as essential security building block of the operating system. Before this trend, Intel and AMD already implemented directly TPMs within their processor firmware with Intel Platform Trust Technology (PTT) [44] and the AMD Platform Security Processor (PSP), which is a closed source dedidcated ARM security co-processor that is compliant to the TPM 2.0 standard [96]. Both of these TPMs fulfill the functionalities of a dTPM without requiring extra hardware and are accepted by the Windows 11 OS.

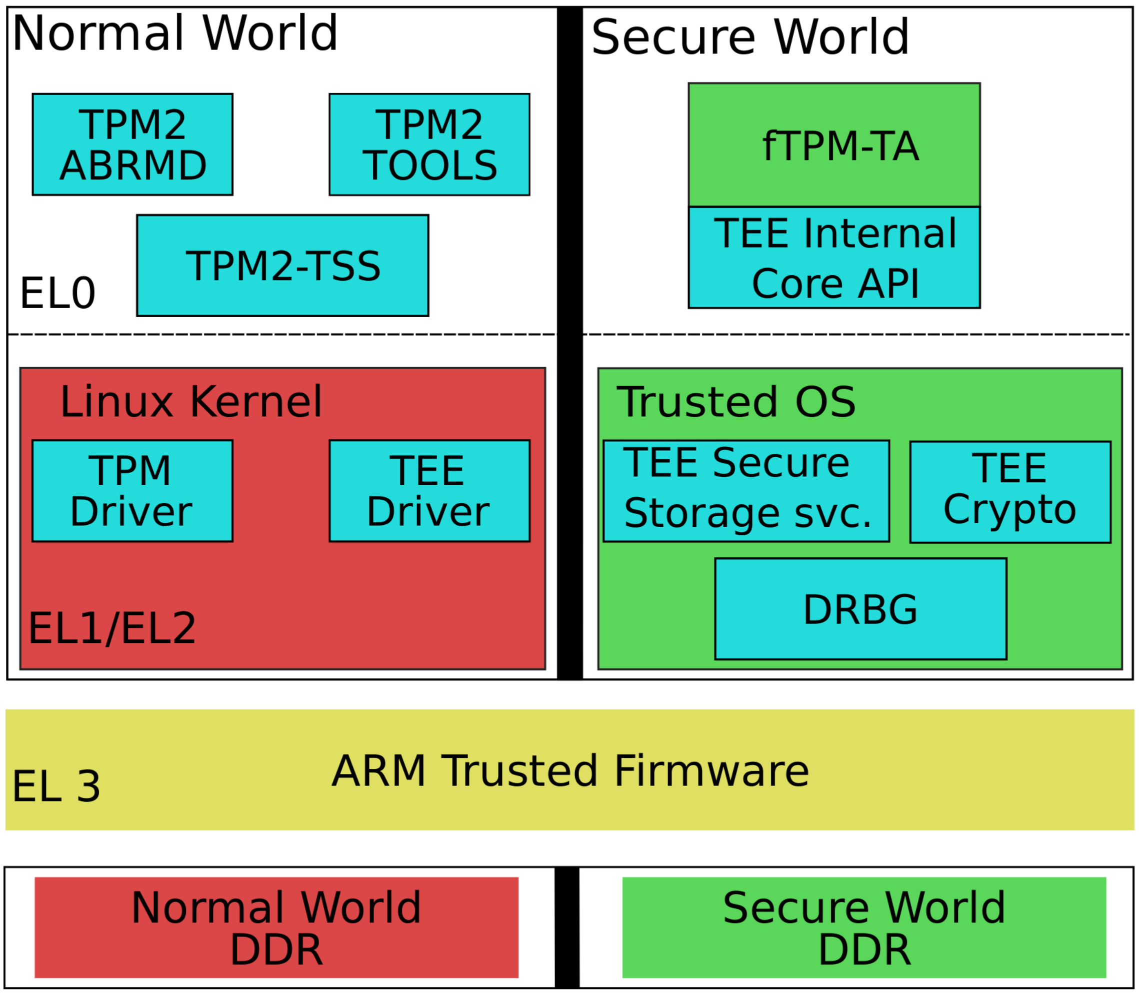


Figure 2.5 Architectural representation of Microsoft's fTPM running inside an ARM TrustZone-based Trusted Execution Environment.

In this thesis, we consider fTPMs capable of running inside an ARM processor implementing the ARMv8-A ISA. This corresponds to the processor class found on Xilinx ZU+ and Intel Stratix-10 devices. For these particular processors, Microsoft released in 2015 an open source implementation of a firmware-TPM which will refer to as fTPM [84]. fTPM is compatible with the TPM 2.0 specification and is used in millions of mobile devices. It implements all the functionalities of a dTPM in software with the help of an ARM TrustZone based TEE (see figure 2.5). fTPM is running as a user level trusted application in the SeW. From the user's point of view, the TPM functionalities are accessible via the *tpm2-tools*, TPM 2.0 software stack (TPM2-TSS), and the TPM2 access broker and resource manager daemon (TPM2-ABRMD) software components. TPM 2.0 commands are handled by a TPM driver which interacts with the fTPM-TA. Many functionalities of the TPM are self contained inside the TA, but in case it needs support from the Trusted OS to handle a specific command, this is possible via the TEE Internal Core API. In Microsoft's reference implementation, secure storage is implemented via the Replay Protected Memory Block (RPMB) partition contained in an embedded MultiMedia Card (eMMC). Some TEE implementations such as Open Portable Trusted Execution Environment (OP-TEE) [62], an open source TEE that we are using in this work offer alternative secure storage features which we rely on instead (REE file system assisted secure storage for OP-TEE, see

Appendix A.0.2 for further details).

Overall, fTPM is an interesting alternative to dTPMs: it offers better performance and does not require additional hardware. However, to be deployed securely in a system, some crucial points such as the source of entropy used by the Deterministic Random Bit Generator (DRBG) should be integrated inside fTPM. In chapter 6, we explain how the security features contained in the ZU+ architecture can be beneficial to fTPM and describe a methodology to generate entropy from embedded SRAM. Furthermore, we explain how our hybrid-TPM can be used for improving the security of the boot process.

2.4 Security Threats Faced by FPGA-SoCs

In this section, a taxonomy of security threats faced by FPGA-SoCs is firstly introduced. This classification sorts the different attacks in three categories: physical attacks, attacks that can be performed by software, and attacks deployable through the FPGA logic. We put the identified security threats in perspective with the security mechanisms introduced in section 2.2 to asses the protection of FPGA-SoCs against these threats. Table 2.1 summarizes the protection of FPGA-SoCs against the identified attack vectors. In chapter 6, we discuss how the insertion of Xilinx security mechanisms in a TEE can help in protecting against some of the attacks analyzed in this taxonomy. Secondly, more details are given on the attack vectors that are used in the chapters 3, 4, and 5 of this thesis.

2.4.1 Physical Attacks

Physical attacks refer to a class of attacks where an attacker requires physical access to a device. This thesis focuses mainly on remote attacks, therefore the overview of physical attacks that threatens FPGA-SoCs is on purpose kept small in this section. Among those attacks, SCA on the bitstream decryption engine were demonstrated possible [76, 52, 37]. This type of attack can affect the confidentiality of IPs included in a bitstream. The new generation of FPGA-SoCs contain built-in countermeasures against this threat such as *key rolling*. This technique consists of limiting the number of side-channel data that can be collected with a specific key and thereby contribute to make SCA on the bitstream decryption engine harder (see section 2.2.2.1 and [37]). Fault attacks performed via voltage [113], clock glitching setups or lasers [13] were shown to be a serious threat for the execution of cryptographic algorithms on FPGA-SoCs [81]. On modern FPGA-SoCs from Xilinx and Intel, the security co-processor is implemented with redundancy, which makes such attacks more difficult. Besides cryptographic algorithms, the non-volatile key storage in BBRAM or key generation via a PUF were also the target of optical attacks [66] and SCA [72]. A further target of physical attacks in FPGA-SoCs is the external DDR memory. This memory is vulnerable to the cold-boot attack, where an attacker cools the DDR memory via a cooling spray and extract secrets from it by restarting the system with a malicious bootloader that can dump the memory content [36]. Fortunately, this attack can be easily mitigated with secure boot, which is available on all the FPGA-SoCs considered in this thesis. Finally the last class of physical attacks that particularly affects FPGA-SoCs that do not have an I/O MMU are Direct Memory Access Attack (DMAA) [8].

2.4.2 Software Induced Hardware Attacks

In contrast to physical attacks, attacks performed by software can be carried out remotely on an FPGA-SoC and are therefore often considered as more practicable. In this section, an overview of attacks that can be carried out through software is given. We particularly consider software induced hardware attacks due to their similarity with FPGA enabled attacks (section 2.4.3). Such attacks rely on software to exploit a vulnerability that is inherently located in hardware. The first class of attacks considered correspond to attacks affecting the memory management within the OS and the underlying memory hardware architecture. As mentioned in section 2.2.3.1, memory isolation and access permission within the OS is implemented with a combination of hardware and software mechanisms. This is crucial for mitigating memory corruption attacks such as a buffer overflow. Further attacks exploiting the memory hierarchy are SCA that are implemented through the microarchitecture of an FPGA-SoC processor. This refers to cache attacks [110] and more complex SCAs such as Spectre [53] and Meltdown [64]. Fault attacks are another class of attacks that are deployable remotely with software. For that purpose, the Rowhammer [51] bug can be exploited to induce faults in Dynamic Random Access Memory (DRAM) modules by making repeated and fast access to specific DRAM rows, which results in the apparition of faults in the adjacent rows. Although modern DDR4 modules include more protection against this type of attacks, they have been shown as still being vulnerable to it [65, 49]. Fault attacks on an ARM processor can further be implemented by taking advantage of Dynamic Voltage and Frequency Scaling (DVFS), a processor performance optimization that has been misused for fault attacks. Fault attacks on an ARM TrustZone TEE were shown possible by raising the CPU clock frequency [91] or by lowering its operating voltage [83]. Besides software induced hardware attacks, rollback attacks [16], which correspond to the execution of an older version of an authenticated software are also possible on FPGA-SoCs and TEEs [16]. To mitigate this threat, public keys used for the authentication of an outdated software should be revocated. Finally, the last class of attacks performed by software correspond to the readback and manipulation of the FPGA configuration through the PCAP. On Xilinx FPGA-SoCs, such attacks are possible even with the assumption of secure boot, as explained in section 2.4.4.1.

2.4.3 Attacks Performed by the FPGA

Attacks performed by the FPGA are another class of remote attacks in FPGA-SoCs whose target can be either another IP core located in the FPGA logic, one of the processing units or the memory. These attacks are notably used to bypass system level security mechanisms such as an MMU. FPGA-SoCs contain memory interfaces which enable a coherent access to the processor's cache hierarchy. This type of interface enables cache attacks performed by the FPGA logic. On the Zynq-7000, the memory interfaces available in the FPGA fabric are not protected by an SMMU. This lack of protection is also found with the Accelerator Coherency Port (ACP) available on the ZU+ and makes DMAA performed by malicious logic possible (see section 2.4.4.2). A further class of attacks that can be performed by FPGA logic correspond to fault [4, 25, 56, 54] and side-channel attacks [92, 28, 111] which are possible due to the sharing of the PDN. Further details about this class of attacks is given in section 2.4.4.4. Finally, the last class of attacks that can be performed by the FPGA logic corresponds to logic readback or reconfiguration through the Internal Configuration Access Port (ICAP). Similarly to the software variant of this attack mentioned in section 2.4.2, such attacks are possible even after the load of an authenticated and encrypted bitstream (see section 2.4.4.1).

Physical	Software attacks	FPGA
Cold-boot ✓/✓	Memory corruption attacks ★/★	DMAA/cache SCA through malicious logic ✓/★
Power SCA ✗/★	SCA (cache, timing) ✗/✗	FPGA bitstream readback/modification ✗/✗
Fault attacks ✗/★	Fault attacks via DVFS ✗/★	On-chip power SCA through malicious logic ✗/★
DMAA ✗/✓	Rowhammer ✗/✗	On-chip fault/DoS attacks through malicious logic ✗/★
	FPGA bitstream readback/modification ✗/✗	
	Downgrade attacks ✗/★	
	DoS ✗/✗	

✓ = covered; ★ = partially covered; ✗ = not covered

Table 2.1 Relevant attack vectors affecting the Zynq-7000/ZU+ FPGA-SoCs and coverage obtained with manufacturer security mechanisms

2.4.4 Attack Vectors Exploited in this Thesis

After the identification of general threats that affect FPGA-SoCs, we provide further details on the four attack vectors that are exploited in the contributions of this thesis.

2.4.4.1 Runtime Hardware Trojans Insertion and IP Theft through Relaxed Trust Assumptions

Partial or total remote runtime reconfiguration of FPGA-SoCs is a desirable feature to take advantage of their flexibility or reprogram the FPGA in the event of a security incident. However, this feature might also be misused to introduce HTs in an FPGA-SoC during runtime. Another security issue can result from debugging interfaces. Those interfaces have an important role in the early design stages of FPGA logic. FPGA-SoCs integrate the possibility of reading back the FPGA configuration to facilitate this process. However, such a feature should not be possible after an encrypted bitstream load that occurred during secure boot, on an FPGA-SoC already deployed on the field.

On Xilinx Zynq-7000 and ZU+ FPGA-SoCs, two interfaces can be used for remote logic reconfiguration and logic configuration readback: the ICAP which is used in standard Xilinx FPGAs and the PCAP which is specific to FPGA-SoCs, where FPGA reconfiguration can be performed by software running on one of the processor cores. In the context of secure boot, the software and FPGA configurations are authenticated and optionally encrypted before being executed. Xilinx considers the PCAP and ICAP as trusted interfaces if they are used by a software which was authenticated in secure boot [101, 48]. This assumption can only be hold true post-secure boot if authentication mechanisms are used before an FPGA-SoC update. However, due to relaxed trust assumptions made by Xilinx, the two FPGA reconfiguration interfaces available on FPGA-SoCs could be used for loading a non-authenticated bitstream inside the FPGA, even if the initial bitstream load was authenticated in a secure boot context. Additionally, the Zynq-7000 and ZU+ offer the possibility of reading back the FPGA configuration even if the bitstream was initially loaded in an encrypted format. This security issue might lead to potential IP theft scenarios.

From a software point of view, runtime modification and readback of the FPGA configuration do not necesseraily require elevated privileges. The programming of the FPGA logic from Linux relies on the *FPGA Manager* kernel driver [105, 106]. The interaction with this driver from the userspace is possible through the *libdfx* library developed for Xilinx FPGA-SoCs [102].

The relaxed trust assumptions in the FPGA reconfiguration interfaces combined with the possibility of reconfiguring an FPGA-SoC from the userspace contribute to make the programming of malicious hardware inside the FPGA logic easier for an attacker and is the basis for the insertion of malicious IP cores, so called HTs as primitives that can be used for attacking other processing cores (see chapters 3, 4, and 5). In chapter 6, a protection mechanism of the PCAP is introduced. The framework consists of disabling the ICAP access and restricting the access of the PCAP to a secure driver that is running within a TEE built with ARM TrustZone support. In comparison to the Linux PCAP driver, the presented driver forces the usage of authenticated bitstream loading.

2.4.4.2 Remote Memory and Peripherals Manipulation Attacks on FPGA-SoCs

Memory isolation is crucial for ensuring correct execution of software and separation between processes. As explained in section 2.2.3.1, the OS and system level isolation mechanisms such as an MMU and ARM TrustZone technology are designed towards the achievement of this goal. However, FPGA-SoCs are complex platforms where the memory subsystem is shared between multiple processing units, the FPGA, and some I/O devices. This sharing of memory opens the possibility for memory manipulation attacks that can bypass memory isolation mechanisms.

On the first FPGA-SoC generation, which typically corresponds to the Zynq-7000 [108] or the Intel Cyclone-V architecture [41], the memory subsystem is accessible via physical addresses from the FPGA logic. Moreover, FPGA-SoCs built on this architecture do not contain any of the memory protection mechanisms mentionned in section 2.2.3.1. Due to these two reasons, several work have demonstrated powerfull DMAAs carried out from the FPGA logic that compromise the software executed on a processor core [45, 46, 15].

Fortunately, as mentionned in section 2.2.3.1, the new generation of FPGA-SoCs contain many memory isolation mechanisms that work in combination with ARM TrustZone. They notably contain an I/O MMU that enables memory isolation for the FPGA and I/O devices which can mitigate the FPGA enabled DMAAs presented in [45, 46, 15]. However, some of the FPGA to memory interfaces can still access the memory subsystem with physical adresses. In that case, the memory protection is ensured by an MPU. A further threat is the lack of protection offered by an SMMU in the early boot phase. Since an SMMU is configured only once an OS has booted, the boot phase of FPGA-SoCs is still vulnerable to DMAAs [45]. In the chapter 3 of this thesis, we analyze the possibility of mounting memory and peripheral manipulation attacks from the FPGA for the ZU+ architecture further and present attacks that can bypass the security mechanisms introduced in section 2.2.3.1.

2.4.4.3 Bypassing TrustZone Security Boundaries from the FPGA Logic

As explained in section 2.2.3.2, ARM TrustZone technology enables the deployment of a TEE (see section 2.3.1), which notably ensures isolation and authentication of security critical software. The deployment of ARM TrustZone to the FPGA logic is possible through the usage of the of a security bit for AXI read (ARPROT[1]) and write (AWPROT[1]) transactions. Letting the security bit unset corresponds to a secure transaction while setting it indicates a normal transaction. By configuring ARM TrustZone for the FPGA logic, is it possible to restrict the access to certain memory regions or peripherals to secure IP cores contained in the FPGA logic. A non-secure IP core contained in the FPGA logic is therefore not capable of accessing assets located in the SeW. Accessing those assets is however possible for a non-secure IP in the context of a privilege escalation [11]. This consists of a hidden functionality in a non-secure IP core which lead to the generation of read and/or write transactions with the security bit unset. Preventing such a privilege escalation would require the definition of a security policy table that contains the security configuration of each master and a transaction checker located on the AXI bus. This simple protection technique is however not implemented by default for the FPGA-SoCs considered in this thesis.

In chapter 3 of this thesis, we combine this attack vector with memory manipulation attacks and demonstrate the possibility of breaking ARM TrustZone memory isolation on an FPGA-SoC. Further experiments in chapter 3 exploit this attack vector for writing to the BBRAM and eFuses in a scenario where the hardware root of trust secure boot of the ZU+ is compromised.

2.4.4.4 Remote Electrical Threats due to a Shared Power Distribution Network

The power dissipation of a chip can be divided into a static part which is proportional to the current and its variation and a dynamic part which is influenced by the toggling of the transistors. Like every power supply, the PDN and especially the voltage regulators it contains cannot deliver a constant voltage to an FPGA-SoC. Instead, the delivered voltage is dependent on the current demand, with a voltage drop observed through the current drain inside the PDN resistive components and a further voltage drop that is generated by the current variation inside the inductive components of the PDN [35].

Several works had exploited this property by designing FPGA circuits which purposely generate a high voltage drop. Such circuits are referred as power-hammering circuits for the rest of this thesis. Power-hammering circuits are usually implemented with Ring Oscillators (ROs) [56, 69] or with circuits that rely on the glitch-amplification effect [71, 58]. Previous works have demonstrated that power-hammmering circuits are capable of generating a voltage drop that is sufficient to crash an entire FPGA [25, 89, 58]. Power-hammering circuits can additionally cause faults targetting a victim circuit located in another part of the FPGA [56, 4, 68, 54]. Furthermore, by enabling the power-hammering with certain toggling frequencies, resonances of the PDN and voltage regulator can be exploited precisely. The resulting voltage drop was demonstrated precise enough to cause timing faults, able to do DFA on AES [56], affect a neural network accelerator [4], and a True Random Number Generator (TRNG) [68], while more recent attempts also aim to keep the power-hammering circuits stealthy and thus harder to detect and prevent [54].

Glitch-amplification is the technique that is used in the chapter 4 of this thesis for implementing remote fault attacks performed by the FPGA logic on an embedded CPU. The glitch amplification principle is depicted in figure 2.6. It relies on a *glitch generator* that consists of a fast clocked flipflop, a delay chain, and a wide-input XOR. Due to the delay chain, the two input signals of the XOR gate are slightly delayed, which lead to additional signal transitions within one clock period, referred as glitches. With an increase in switching activity, glitches contribute to an increase in the dynamic power consumption of an FPGA-SoC. To further increase the dynamic power consumption, the glitch amplification phenomena relies on a *power-burning logic*. This consists of logic elements and the capacitance contained in the wires along the routing path. The toggling of the signals inside these elements is the main dynamic power consumption source of the glitch amplification effect.

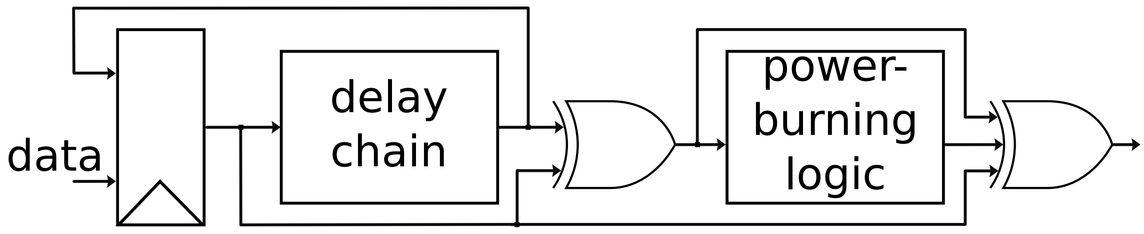


Figure 2.6 Generation of voltage drop with a circuit based on the glitch-amplification effect [71]

A further electrical threat faced by FPGA-SoCs is the possibility of observing the PDN flucutations via on-chip voltage sensors implemented in the FPGA logic. In order to observe the voltage variations resulting from the PDN, the inversely proportional relation between the supply voltage and the propagation delay of a signal is used. The propagation delay variation of a clock signal can be measured in a so-called delay-line circuit, which acts as a voltage sensor. One example of such a circuit is the Time-to-Digital Converter (TDC) circuit [26] represented in figure 2.7.
A TDC measures the propagation delay of a clock signal inside a circuit consisting of an initial delay and a chain of delay elements, which constitute the delay-line. Latches and registers are used to depict the propagation of the clock signal inside the delay-line during one clock period. The delay-line state is then reflected as a thermometer code inside the registers. In case of a voltage decrease, the clock signal propagates less inside the delay-line, which is seen by a decrease in the delay-line state's Hamming Weight. Inversely, if the voltage raises, the delay-line state's Hamming Weight increases. By using this principle, TDCs can be effectively used as voltage sensors and have been used for side-channel attacks against a cryptographic core located in the FPGA [85] or a software implementation running on a CPU [28].

The PDN can also be used for implementing a covert channel. In that case, both capabilities of stressing and observing the PDN through logic circuits implemented within the FPGA are used. In the work of [27, 22], ROs have been used as PDN stressors and TDCs respectively ROs as PDN observers. Power covert channels were also demonstrated to be feasible between a PC's CPU and an FPGA mounted on an acceleration card in [22]. By modulating the shared Power Supply Unit (PSU) usage via the Linux `stress` and `sleep` functions, the authors implemented a covert channel with a transmission rate of up to 6.1 bit/s and a 97% transmission accuracy. In the chapter 5 of this thesis, a similar scenario is evaluated in the SoC context, with the FPGA and CPU co-located on the same chip.

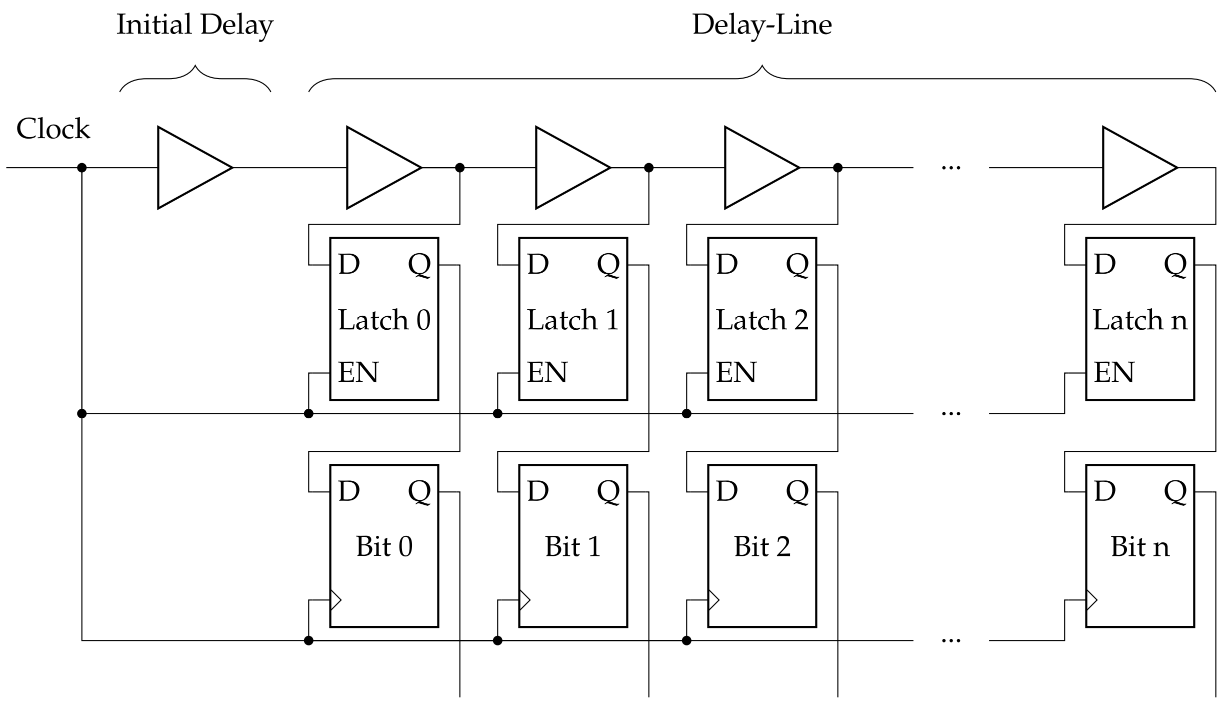


Figure 2.7 Schematic of an n-bit delay-line Time-to-Digital Converter with an initial delay unit

3 Compromising Memory and Peripherals Isolation within an FPGA-SoC via Malicious Hardware

FPGA-SoCs contain memory and peripherals which are shared between the different processing units they integrate. The sharing of logical resources enables performance and flexibility for designs but at the same time it can open the doors to some security attacks performed by the FPGA logic on a CPU. In this chapter, we investigate such attacks.

Similar to the works of [46, 15, 61] we exploit a security vulnerability of FPGA-SoC architectures, which allows a HT to perform DMAA on the CPU subsystem. This chapter, however, considers the ZU+ architecture, which contains more protection mechanisms than the previous Zynq-7000 architecture. We show the feasibility of performing powerful DMAAs on ARM TrustZone, despite the protection provided by this technology against DMAAs. Through memory dump and memory manipulation, we achieve the execution of un-authenticated software in a TEE built with ARM TrustZone and can reveal cryptographic keys securely stored within the TEE. Besides the memory isolation issue, we demonstrate that a peripheral isolation issue also exists. We showcase a proof of concept attack allowing a HT connected to the ACP to bypass the secure boot configuration set by a device owner via the access to the eFuses and BBRAM peripherals. An attack on secure boot was already demonstrated on a Zynq-7000 platform in [46]. This chapter considers a similar attack on the ZU+ platform and uses a different approach as the one proposed in [46].

The rest of this chapter is organized as follows: section 3.1 describes the ACP and explains a security vulnerability in the mechanism used to isolate CPU private memory/peripherals from a tightly coupled ACP master. Section 3.2 demonstrates two concrete attack examples on a TrustZone based TEE. Section 3.3 demonstrates an attack which compromises the hardware root of trust secure boot mode of the ZU+. Section 3.4 discusses possible mitigations against the attacks presented in this work and their portability to other FPGA-SoCs. Section 3.5 provides a summary of this chapter.

The results presented in this chapter were part of two publications: *Breaking TrustZone Memory Isolation through Malicious Hardware on a Modern FPGA-SoC in the Attack and Solutions in Hardware Security workshop - ASHES 2019 [31].* This publication was further enhanced with *Breaking TrustZone memory isolation and secure boot through malicious hardware on a modern FPGA-SoC in Journal of Cryptographic Engineering -JCEN September 2021 (volume 11, issue 3) [30].*

3.1 Security Vulnerability of the Accelerator Coherency Port

This section firstly describes the usage of the ACP inside the ZU+ MPSoC. Subsequently, the mechanisms used to prevent an ACP master to access processor private memory and peripherals are discussed.

3.1.1 ACP Slave Interface on the ARM Cortex-A53

As described in figure 2.2 and section 2.1, the ZU+ contain many processing units which are intended for performance, real-time application, security, and power management. In this section, a special focus is made on the APU ,the FPGA and the memory access through the FPGA logic. A more detailed architectural representation of the ZU+ that integrates previous aspects is represented in figure 3.1. The ZU+ integrate several memory interfaces which enable the FPGA fabric to access the system memory via the Processing System (PS). Among all the available FPGA fabric memory interfaces, the ACP is recommended for applications where a hardware accelerator is tightly coupled with the APU. In comparison to the other FPGA memory interfaces, the ACP has the fastest memory access. This is achieved via a direct connection to the Snoop Control Unit (SCU) of the APU (see figure 3.1). The ACP is interfacing memory via 40 bit physical addresses and a 128 bit data bus. Connecting a hardware accelerator to the SCU instead of the CCI enables a master in the FPGA fabric to have a fast coherent access to the APU L1 and L2 caches. If the data requested by the hardware accelerator is not present in the ARM Cortex-A53 caches, the ACP optionally enables the allocation of a new cache line inside the L2 cache. This coherent interface is however restricted to 16 Bytes and 64 Bytes burst transactions. The ACP only provides I/O coherency and is therefore not suitable for a hardware accelerator which has private caches. For this particular use case, the AXI Coherency Extension (ACE) interface, an interface which provides bi-directional coherency should be used instead. The ACE port has nevertheless slower access times to data than the ACP because of additional latency induced by the CCI. ACP and ACE are the only interfaces in the logic fabric which can access memory via physical addresses. The other memory interfaces contained inside the FPGA fabric access memory via virtual addresses through the SMMU.

3.1.2 Processor and ACP Master Memory Isolation

The ACP is typically used to connect a hardware accelerator to the ARM Cortex-A53 memory subsystem. In this scenario, it is necessary to restrict the visible address space of the hardware accelerator such that it cannot compromise the software running on the processor. The ideal candidate for this is the SMMU. However, as depicted in figure 3.1, the ACP is not connected to the SMMU. Alternatively, the XMPUs should be effective to restrict the memory access rights of a hardware accelerator. To verify that the XMPUs can indeed prevent a hardware accelerator to access the whole APU memory via the ACP, a closer look at the XMPUs' isolation mechanisms is necessary.

As explained in section 2.2.3.1, the XMPUs enables memory isolation via the definition of several memory regions. A memory region is characterized by:

- The start address of the region (R_START).
- The end address of the region (R_END).

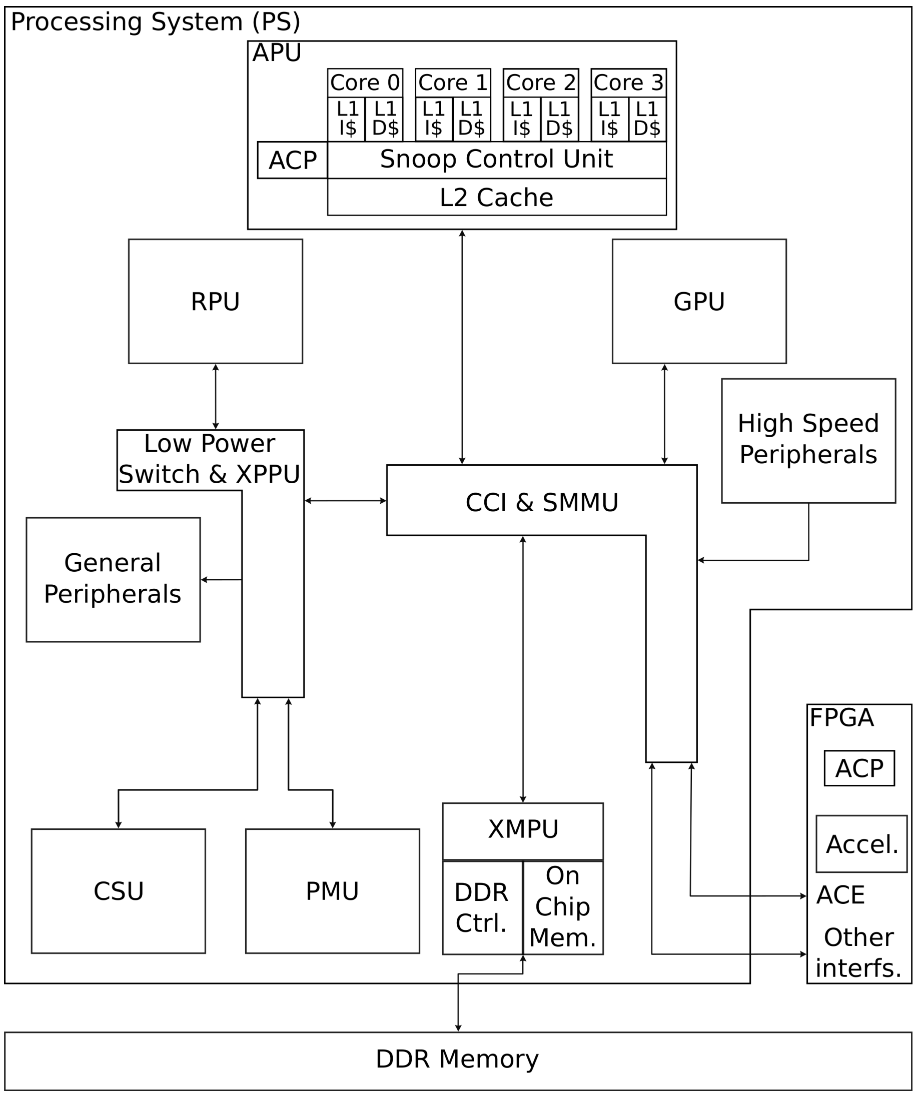


Figure 3.1 Refined Architectural Representation of the Zynq UltraScale+ FPGA-SoC

- The security property (secure/non-secure) of the region (R_SECURE) which is checked against the security bits of an AXI transaction (AXI_ARPROT[1]/AXI_AWPROT[1]).
- The region master ID value (R_MID_V) and the region master ID mask (R_MID_MASK).

An incoming read or write request at an address (AXI_ADDR) is checked against the conditions listed in equation 3.1 for each memory region (R_i) defined in the XMPUs' configuration registers.

$$\begin{cases} R_i\_START \leq AXI\_ADDR \leq R_i\_END \\ AXI\_MID\_V \& AXI\_MID\_MASK == R_i\_MID\_V \& R_i\_MID\_MASK \\ AXI\_ARPROT[1]/AWPROT[1] == R_i\_SECURE \end{cases} \tag{3.1}$$

Only AXI transactions satisfying equation 3.1 are granted. AXI transactions which are not matching the security configuration of a region are rejected by the XMPUs and can be optionally notified to the master via an interrupt. The ZU+ documentation [107] provides the necessary information regarding APU master ID. APU transactions have their master ID defined according to equation 3.2.

$$APU\_MID[9:0] = 0010||AXI\_MID[5:0] \tag{3.2}$$

Xilinx does not provide further information regarding the ACP master ID. An inspection of the ARM Cortex-A53 Technical Reference Manual [5] reveals that the 6 lowest bits of the AXI read/write transaction ID can differentiate APU and ACP transactions. The encoding for the 6 lowest bits of the read/write ID is given in table 3.1. The different transaction types for the ARM Cortex-A53 include normal read/write transactions performed by the CPU cores as well as exclusive transactions which are semaphore-type operations. Synchronization barriers generated by one of the CPU cores or the SCU are also possible. Finally, reads and writes performed by the ACP close the list of possible transaction types.

AXI_MID	**Number of supported outstanding transactions per ID**	**Transaction type**
0b0000nn[1]	4	Core nn exclusive read/write or non-reorderable device read/write
0b0001nn[1]	1	Core nn barrier
0b001001	1	SCU generated barrier or distributed virtual memory complete
0b01xx00	1	ACP read/write
0b1xxxnn[1]	1	Core nn read/write

[1] Where nn is the core number 0b00, 0b01, 0b10 or 0b11

Table 3.1 Read/write transactions ID encoding for the ARM Cortex-A53

3.1.3 Processor and ACP Master Peripheral Isolation

As mentioned in section 3.1.1, the ACP is typically used to connect a hardware accelerator to the memory subsystem. In addition, this interface also enables access to system peripherals and some configuration registers for a hardware accelerator. For security and safety reasons, it is good practice to make peripherals accessible only to specific masters. Some peripherals are by design only accessible to a restricted list of masters, for others the access restriction can be achieved via the use of the XPPU. In section 3.1.2, we explained that the XMPUs cannot isolate memory regions of the APU from a hardware accelerator using the ACP. In this section, we investigate whether this issue extends to the XPPU as well.

As explained in section 2.2.3.1, the XPPU is operating in a similar way as the XMPUs, except that the notion of memory regions is replaced with apertures. An aperture is a range of register addresses. The aperture permission list defines the masters which are allowed to read/write to a given aperture. In total, 400 apertures are defined on the ZU+.

The access control realized by the XPPU is explained in equation 3.3. The first step consists in identifying the aperture corresponding to an incoming AXI transaction (APPER$_{\text{inc}}$). Once it is found, the XPPU performs a master ID filtering operation similar to the one of the XMPUs (see section 3.1.2). The access can only be granted if the result of the filtering operation is contained in the list of the authorized master profiles for the aperture. The final check consists in verifying that the security of the transaction matches the one of the aperture.
If any of these 3 checks fail, the peripheral access is denied, which results in a rejection of the transaction.

$$\begin{cases} (AXI\_MID\_V \& AXI\_MID\_MASK) \in APPER_{\text{inc}}\_AUTHORIZED\_MASTERS \\ AXI\_ARPROT[1]/AWPROT[1] == APPER_{\text{inc}}\_SECURE \end{cases} \tag{3.3}$$

Similar to the observations made in section 3.1.2, we expect the peripheral isolation between the APU and a hardware accelerator using the ACP to work from a theoretical point of view. To verify whether this is the case, we follow the procedure from section 3.1.2 and configured the XPPU isolation inside Vivado. As a result, the XPPU should prevent the ACP from accessing the address space of a peripheral while allowing APU to access that peripheral. However, this did not work in practice, because as with the XMPU, the XPPU cannot distinguish APU and ACP transactions. A closer look at the XPPU registers reveals that the APU core 0 master profile is configured with the ID 128 (b1000 0000)and mask 960 (b0011 1100 0000). Since the six lowest bits of the mask are unset, it is not possible for the XPPU to distinguish APU core 0 and the ACP transactions (see equation 3.2, table 3.1, and the discussion of section 3.1.2).

This mask value leads to peripherals isolation issues. A HT contained inside an accelerator interfacing memory via the ACP can access peripherals which it is not supposed to. Peripherals such as cryptographic accelerators, secure key storage medium(PUF, BBRAM, and eFuses), and voltage and temperature sensors contained on an FPGA-SoC are interesting peripherals which might be used for an attack. To illustrate the security implications of the peripherals isolation issue, we have implemented an attack in which an attacker can break secure boot (see secure boot background information in section 2.2.2.1) and take control of a ZU+ device by interfering with the eFuses and the BBRAM. This attack is described in section 3.3.

3.2 DMA Attacks on OP-TEE

This section shows that a HT contained inside an ACP master can compromise the software running on the APU via memory manipulation. Our first Proof of Concept (PoC) demonstrates how the HT can affect the signature verification of trustlets before their execution inside OP-TEE [62]. OP-TEE is a TEE initially developed by ST-Ericsson and STMicroelectronics as a closed source project before being released as an open source project by Linaro in 2014. The second PoC demonstrates the retrieval of an AES key securely stored via software support. This key is used for an AES-GCM decryption performed inside a trustlet and can be found in a SeW memory dump.

3.2.1 System Description

Architectural description: This work uses a Xilinx ZU+ MPSoC ZCU102 Evaluation Kit. The system considered in this work is presented in figure 3.2. It consists of the PS and a third party IP contained in the reconfigurable logic (IP 1). An Embedded Linux solution (Rich OS) is running on the APU. Furthermore, an ARM TrustZone based TEE is executing in parallel to the Rich OS. The TEE consists of ATF executing inside the OCM and OP-TEE executing inside the DDR memory. OCM and DDR are partitioned in the NoW and the SeW. Since IP 1 is obtained from a third party, it cannot be fully trusted. Unfortunately, a hidden malicious functionality is contained inside IP 1. To fulfill its functionality, IP 1 shares a portion of the NoW DDR with the APU (APU/IP 1 shared section represented in figure 3.2). The XMPUs are used to prevent IP 1 from accessing memory outside of this section. The configuration of the XMPUs is done according to Xilinx recommendation, with a tool integrated inside Vivado [1]. The partitioning of the DDR memory and the OCM is shown in table 3.2. As depicted in figure 3.2, only the ATF is running inside the OCM, therefore the whole OCM has been placed inside the SeW. Since a TEE is lightweight, a small portion of the DDR memory (8 MB) has been configured as secure. The rest of the DDR memory (1500 MB) is occupied by the Rich OS running on the APU. Among these 1500 MB, 100 MB are shared between the APU and the ACP master. This configuration is typical for the use of a tightly coupled accelerator inside the FPGA fabric. Such scenarios are relevant for a wide range of applications such as video processing, machine learning, and cryptography.

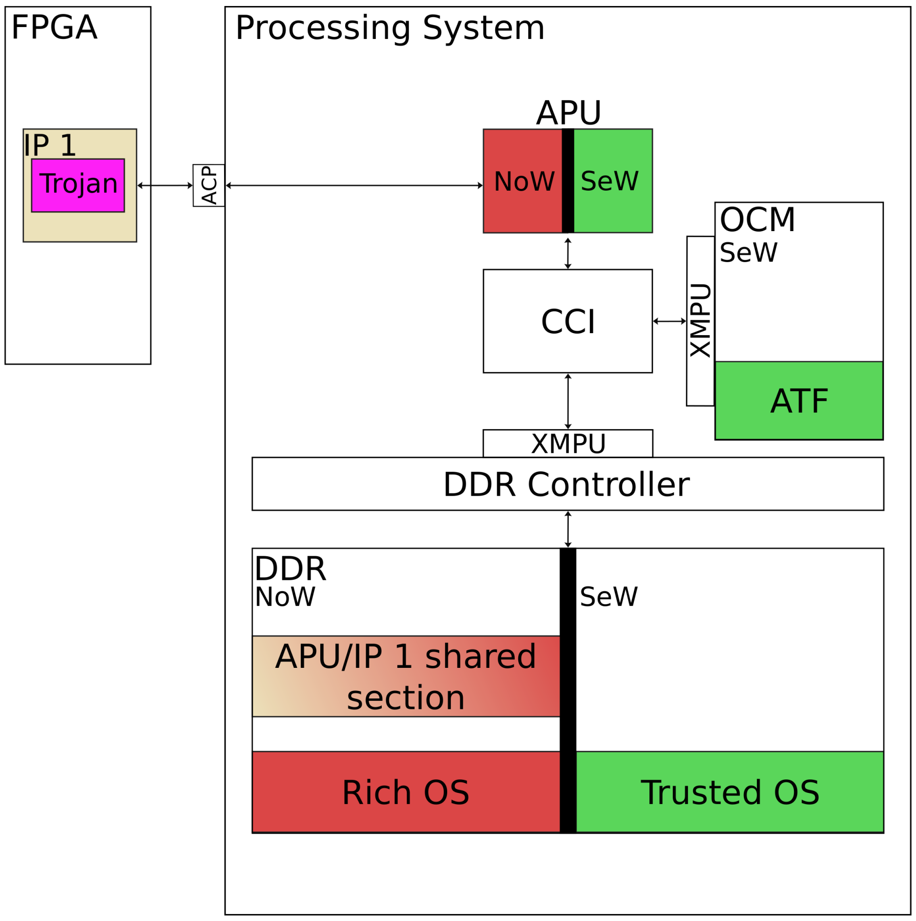


Figure 3.2 System block design

Software stack description: On the software side, Petalinux 2018.2, an Embedded Linux solution designed for Xilinx devices, is running as the NoW Rich OS. OP-TEE 3.4.0 is used

Master	Start Address	Size	TrustZone	Memory Type
APU Non-Secure subsystem				
APU	0x0	1500 MB	NoW	DDR
APU Secure subsystem				
APU	0x60000000	8 MB	SeW	DDR
APU	0xFFFC0000	256 kB	SeW	OCM
ACP subsystem				
S_AXI_ACP	0x30000000	100 MB	NoW	DDR

Table 3.2 XMPUs configuration

as the SeW Trusted OS. OP-TEE relies on the TEE Client API v1.0 and the TEE Internal Core API v1.1 to implement a TEE. The use of OP-TEE offers isolated execution of security critical software inside the FPGA-SoC.

3.2.2 Compromising the Signature Verification of Trustlets before their Execution inside OP-TEE

Trustlets are binaries running inside the SeW at EL0. These applications access the core function of OP-TEE running at EL1 via the TEE Internal Core API. Trustlets can be developed by third parties and integrated inside a system. Therefore, it is crucial to ensure the authenticity and integrity of a trustlet before executing it. To achieve this, the trustlets are stored as signed binaries inside the Rich OS Root File System (RootFS) (see figure 3.3). The private key used for signing the trustlets is not present inside the Rich OS RootFS. This prevents the modification of trustlets and the insertion of new trustlets in case of a compromised Rich OS.

The start of a trustlet is initialized by a client application. A special component (tee-supplicant) will then take care of loading the trustlet into the SeW. Once loaded in the SeW, the signature verification of the trustlet is performed. This verification checks the integrity and authenticity of a trustlet before executing it. If the signature verification fails, the client application is notified and the execution of the trustlet stops. In the other case, the trustlet is executed in EL0. The first PoC of this work aims at compromising the signature verification of a trustlet via a DMAA (shdr_verify_signature function contained in OP-TEE core) such that non-authorized trustlets can be executed on the system.

Assuming the system setup described in section 3.2.1 and the XMPUs configuration in table 3.2, a DMAA is possible because of the memory isolation issue described in section 3.1.2. In order to take advantage of the isolation issue for attacking TrustZone protected memory, the HT must find a way to access SeW APU memory. As explained in section 3.1.2, XMPUs' registers are locked after being configured by the FSBL. On ZU+ devices, the FPGA fabric is also loaded after the boot of the processor. Therefore, a manipulation of the XMPUs' registers from the HT is not possible. Instead, the HT can simply set the security bit to 0 during read and write transactions. By doing so, the generated transactions are tagged as secure and the XMPUs return no security error. This privilege escalation performed inside the FPGA fabric is necessary to access the APU SeW memory. To the best of our knowledge, Xilinx does not provide any means to define a fixed security policy of AXI masters inside the FPGA fabric via a policy table containing the security profile and access type possible for of each master.

The exploitation of these two issues enables an attacker to write arbitrary code and data inside TrustZone memory and thereby making code injection inside SeW DDR memory possible. Our implementation of the DMAA on the signature verification function should enable the execution of trustlets authenticated with an untrusted private key. The attack consists of an offline and online phase. The steps of the attack are outlined below:

- Identify the code of shdr_verify_signature function (*codeToReplace*) by disassembling the OP-TEE binary (offline).
- Modify the C code of shr_verify_signature so that all signature verifications are valid (offline).
- Recompile OP-TEE and identify the code of the modified shdr_verify_signature (codeToInject) by disassembling the OP-TEE binary (offline).
- Dump the SeW DDR memory and identify the start address of *c*odeToReplace (online).
- Write the *codeToInject* over of the *codeToReplace* via the ACP (online).

To verify the success of our attack, we tried to execute a trustlet which is signed with an untrusted private key. If OP-TEE is not compromised, the execution of the trustlet is not possible because the signature verification mechanism detects a security violation. After the injection of the malicious code, non-authorized trustlets could be executed without any error notification from OP-TEE. This type of attack becomes relevant for an attacker which manages to insert a malicious trustlet inside the Rich OS RootFS. Such a scenario corresponds for instance to the download of a trustlet from malicious sources on the Internet. Alternatively, an adversary that has obtained control of the Rich OS can replace existing trustlets with malicious trustlets compiled with her own private key.

3.2.3 Retrieving an AES Key Securely Stored with OP-TEE Software Support

Use case description: The second PoC considered in this work is the decryption of sensitive files inside the FPGA-SoC. Since the Rich OS is prone to attacks, a good security practice consists in using a dedicated hardware module in the FPGA to perform the decryption. Alternatively, the designer can leverage the TEE capabilities to implement the decryption in a secure way in software. This work uses the second option as a design choice. We assume that the file is encrypted with AES-GCM-128. The AES key (*K0*) is securely stored in an encrypted form inside the Rich OS RootFS via the secure file storage feature integrated inside OP-TEE. *K0* is only accessible to a specific trustlet (*trustlet_0*). This access limitation prevents a compromised Rich OS to access *K0*. Moreover, unauthorized trustlets cannot get information about *K0*. The interested reader can find complementary information regarding OP-TEE secure file storage capabilities in appendix A.0.2.

In addition to a trustlet specific secure key storage, OP-TEE provides isolated AES-GCM-128 decryption via the cryptographic functions contained inside the OP-TEE core. OP-TEE core relies on the use of *LibTomCrypt* to perform the AES-GCM decryption. This implementation precomputes the AES key schedule and stores it in a contiguous memory buffer

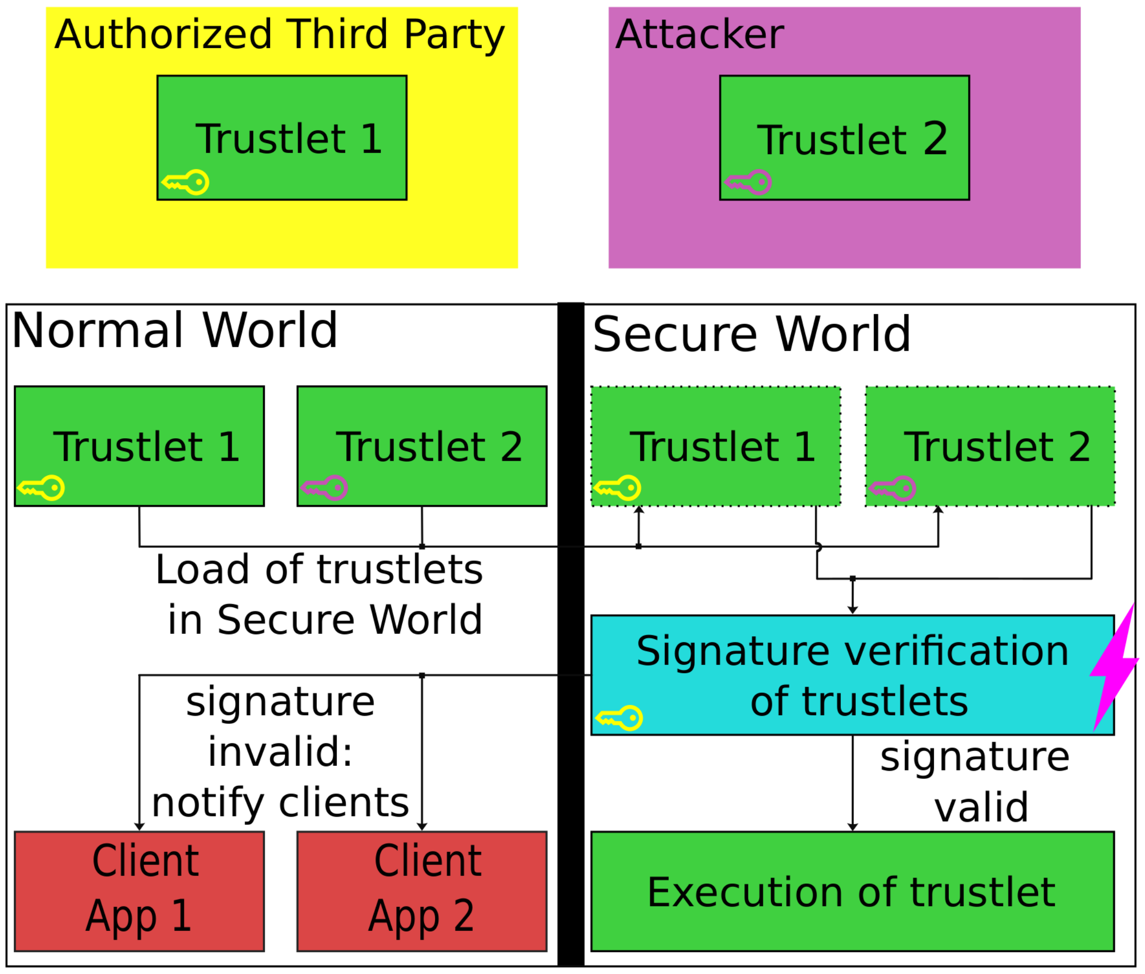


Figure 3.3 Attack on the trustlets signature verification

to increase performance.

Trustlet_0 implements the access to the secure key file and the AES-GCM decryption via the TEE Internal Core API. The NoW client application provides the encrypted file, a 12 Bytes Initialization Vector (IV), and the key_id. These inputs are processed according to the algorithmic description shown in figure 3.4. If a key (*key0*) associated to key_id0 exists in an encrypted form inside the Rich OS RootFS (see appendix A.0.2), it is loaded from the Rich OS RootFS and decrypted inside *trustlet_0*. Otherwise, *K0* is randomly created and securely stored in the REE RootFS for further usage. Once decrypted or generated, *K0* is used for decrypting the sensitive file. Before sending the plaintext back to the client application, a tag verification ensures the authenticity and integrity of the file. If the file has been tampered, an error message is sent back to the client application. In the other case, the plaintext is sent back to the client application.

DMAA description: For the second PoC of this work, it is assumed that the attacker has access to one sensitive encrypted file. This file can be obtained by compromising the server generating it or by eavesdropping the communication between the server and the FPGA-SoC. The attacker's goal consists in finding the AES key necessary for decrypting the file with the help of a SeW memory dump. The system setup described in section 3.2.1 and

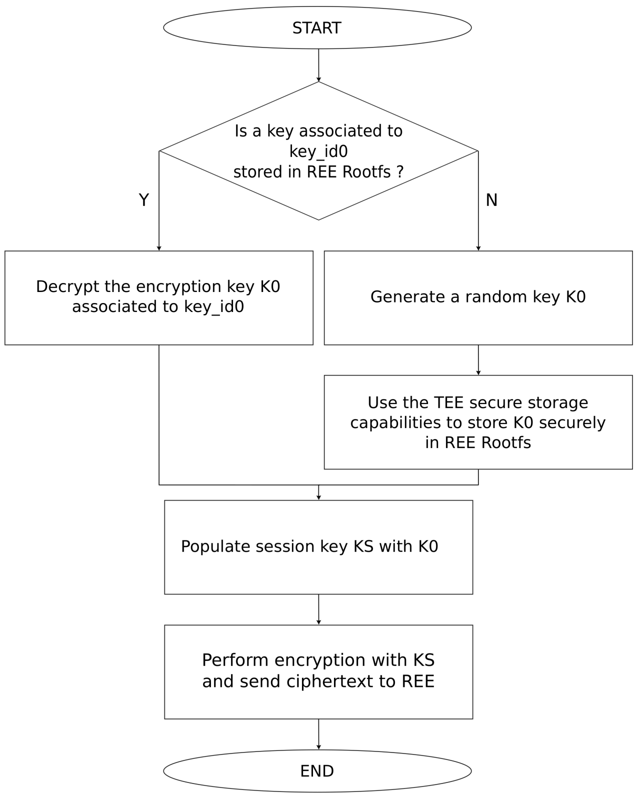

START
Is a key associated to key_id0 stored in REE Rootfs ?
Y
N
Decrypt the encryption key K0 associated to key_id0
Generate a random key K0
Use the TEE secure storage capabilities to store K0 securely in REE Rootfs
Populate session key KS with K0
Perform encryption with KS and send ciphertext to REE
END

Figure 3.4 *Trustlet_0* description

the XMPUs' configuration described in table 3.2 is assumed to be run on the FPGA-SoC. The first step of the attack consists in dumping the whole SeW memory (8 MB) via the HT contained inside IP 1 (see figure 3.2). This is done by generating secure read transactions (ARPROT[1]=0) on the SeW memory via the ACP. The next step is to scan the obtained memory dump. Since *LibTomCrypt* stores a precomputed AES key schedule in memory, this structure should be observable in a memory dump. Similar to [36], we identify an AES key inside a memory dump by searching for a specific key schedule. The pseudocode for finding an AES key inside a memory dump is explained in algorithm 1.

We verified the success of our approach for different AES keys. Figure 3.5 corresponds to the portion of the memory dump containing the key schedule (in little endian representation) associated to the AES key 8b 94 06 88 eb 6b d4 48 0f e5 6a 33 ac 2f f8 07.

In order to decrypt the sensitive file, the knowledge of the IV is an additional requirement. This parameter is usually not secret but should not be used multiple times with a same key to prevent IV reuse attacks [47]. We assume that this parameter is known to the attacker.

3.3 Compromising Secure Boot and Secure Device Update via the Accelerator Coherency Port

This section shows that a HT contained inside an ACP master can compromise the secure boot of the ZU+ via peripherals manipulation. This is achieved by exploiting the possibility for a HT to program the BBRAM and eFuses via the ACP. After programming an RSA

Algorithm 1 AES key finder from memory dump

```
1: procedure AES KEY FINDER(in memory_dump, out key_found)
2:     word_iterator [31:0]
3:     key_cand [127:0]
4:     key_schedule_cand [351:0]
5:     $key\_found \leftarrow 0$
6:     while $key\_found \neq 1$ OR $word\_iterator \neq endOfFile$ do
7:         $key\_cand \leftarrow$ 16 Bytes following word_iterator
8:         $key\_schedule\_cand = AESKeySchedule(key\_cand)$
9:         if $key\_schedule\_cand \subset memory\_dump$ then
10:            $key\_found \leftarrow 1$
11:        end if
12:        word_iterator++
13:    end while
14: end procedure
```

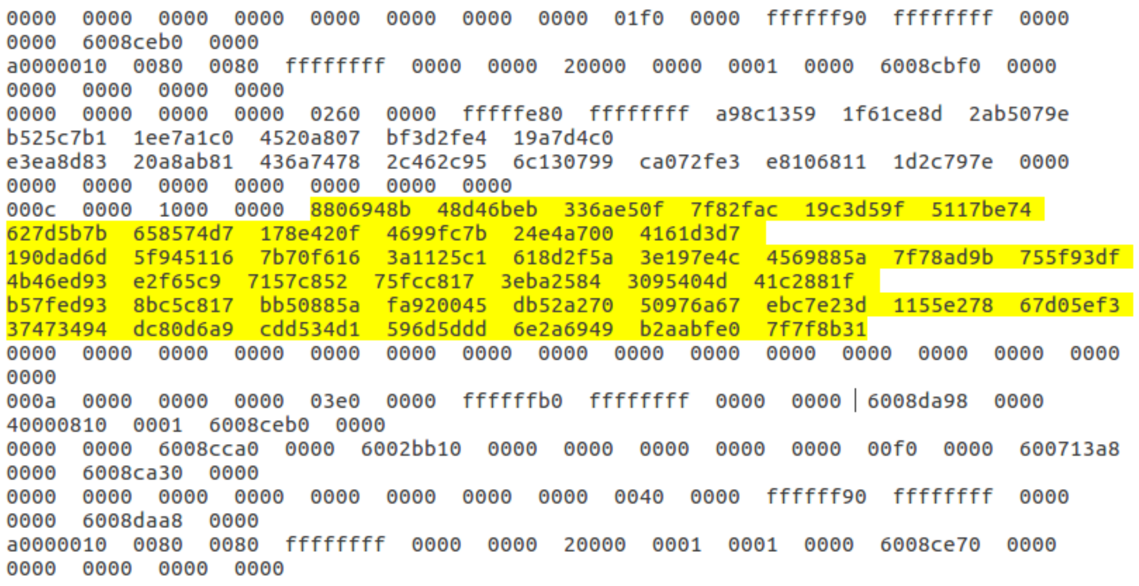
public key hash in the eFuses and an AES key in the BBRAM, our PoC shows that the attacker is able to start her own authenticated and encrypted boot image in the hardware root of trust secure boot scheme of the ZU+.

3.3.1 System Description

The system architecture is shown in figure 3.6. It consists of the FPGA-SoC and one server which is used to provide configuration updates. In order to transmit the updates securely to the device owner, the configuration update files are authenticated with RSA signatures and encrypted with AES-GCM. This scheme is compatible with the hardware root of trust secure boot mode of the ZCU102 Evaluation Kit which is used in this work. In order to use this secure boot scheme, the device owner has programmed the hash of the server's public key and an AES key inside the eFuses reserved for PPK0 and the AES device key. The device owner has deliberately not configured the set of eFuses used for storing PPK1, so that it is possible to program a new key if the private key of the server gets compromised. By doing this, it is also possible to program the public key of another trusted source inside PPK1 later on.

The FPGA-SoC configuration which is booted is shown in figure 3.6. It consists of the PS and a third party IP located inside the FPGA fabric (IP 1) which is connected to the PS via the ACP. Similarly to section 3.2.1, the XMPUs are enabled to restrict the memory access of the accelerator. Since IP 1 only requires access to a restricted memory subsection and not to the APU peripherals, the XPPU is in addition configured to prevent an access to peripherals. Among those peripherals is the eFuses controller, which is accessed by the FSBL during the authentication of the boot image.

Despite the use of secure boot, IPs obtained from third parties may still contain a hidden HT. Moreover, as explained in section 2.4.4.1, the relaxed trust assumptions in the FPGA reconfiguration interfaces enable the load of non-authenticated bitstream after an FPGA-SoC secure boot. We assume that the attacker has managed to include a HT inside IP 1 which she intends to use for taking control of the device. Sections 3.3.2 and 3.3.3



```
0000  0000  0000  0000  0000  0000  0000  0000  01f0  0000  ffffff90  ffffffff  0000
0000  6008ceb0  0000
a0000010  0080  0080  ffffffff  0000  0000  20000  0000  0001  0000  6008cbf0  0000
0000  0000  0000  0000
0000  0000  0000  0000  0260  0000  fffffe80  ffffffff  a98c1359  1f61ce8d  2ab5079e
b525c7b1  1ee7a1c0  4520a807  bf3d2fe4  19a7d4c0
e3ea8d83  20a8ab81  436a7478  2c462c95  6c130799  ca072fe3  e8106811  1d2c797e  0000
0000  0000  0000  0000  0000  0000  0000
000c  0000  1000  0000  8806948b  48d46beb  336ae50f  7f82fac  19c3d59f  5117be74
627d5b7b  658574d7  178e420f  4699fc7b  24e4a700  4161d3d7
190dad6d  5f945116  7b70f616  3a1125c1  618d2f5a  3e197e4c  4569885a  7f78ad9b  755f93df
4b46ed93  e2f65c9  7157c852  75fcc817  3eba2584  3095404d  41c2881f
b57fed93  8bc5c817  bb50885a  fa920045  db52a270  50976a67  ebc7e23d  1155e278  67d05ef3
37473494  dc80d6a9  cdd534d1  596d5ddd  6e2a6949  b2aabfe0  7f7f8b31
0000  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000  0000
0000
000a  0000  0000  0000  03e0  0000  ffffffb0  ffffffff  0000  0000 | 6008da98  0000
40000810  0001  6008ceb0  0000
0000  0000  6008cca0  0000  6002bb10  0000  0000  0000  0000  0000  00f0  0000  600713a8
0000  6008ca30  0000
0000  0000  0000  0000  0000  0000  0000  0000  0040  0000  ffffff90  ffffffff  0000
0000  6008daa8  0000
a0000010  0080  0080  ffffffff  0000  0000  20000  0001  0001  0000  6008ce70  0000
0000  0000  0000  0000
```

Figure 3.5 Little endian representation of the AES key schedule found in a Secure World memory dump

explain the attack vectors that are exploited by the HT. Once these steps are performed, we explain how the attacker is able to start her own authenticated and encrypted boot image in section 3.3.4.

3.3.2 Programming of an RSA Public Key Hash into the eFuses from the ACP

As explained in section 2.2.2.1, the hardware root of trust secure boot relies on RSA authentication with public parameters (PPK hash and SPK_{ID}) contained inside the boot image and a comparison with a value stored inside eFuses. In this experiment, we assume that the device owner has only programmed one PPK hash inside the eFuses and investigate whether a HT contained inside an ACP master can program the second PPK hash.

From the device owner perspective, altering unburned eFuses should not be possible since the XPPU has been configured in a way that the ACP cannot access it. Due to the malfunction described in section 3.1.3 the ACP can however access the eFuse controller. The procedure used for programming a PPK hash from the ACP into the eFuses is described in algorithm 2. After following these steps, the HT can read back the PPK value programmed inside the $PPK1_{0..11}$ registers, which confirms the success of the attack.
In section 3.3.4, we show that this attack primitive enables an attacker to start a boot image that is authenticated with her own private key and thereby allows her to bypass the hardware root of trust secure boot configuration set by the device owner.

3.3.3 Programming of an AES Key into BBRAM from the ACP

The BBRAM stores a 256 bit AES device key which can be used for decrypting a boot image and authenticating it in the encrypt only secure boot. In contrast to eFuses, BBRAM can be reprogrammed multiple times. Xilinx provides code snippets which enable the programming of BBRAM from a processor (APU or RPU). The BBRAM registers are

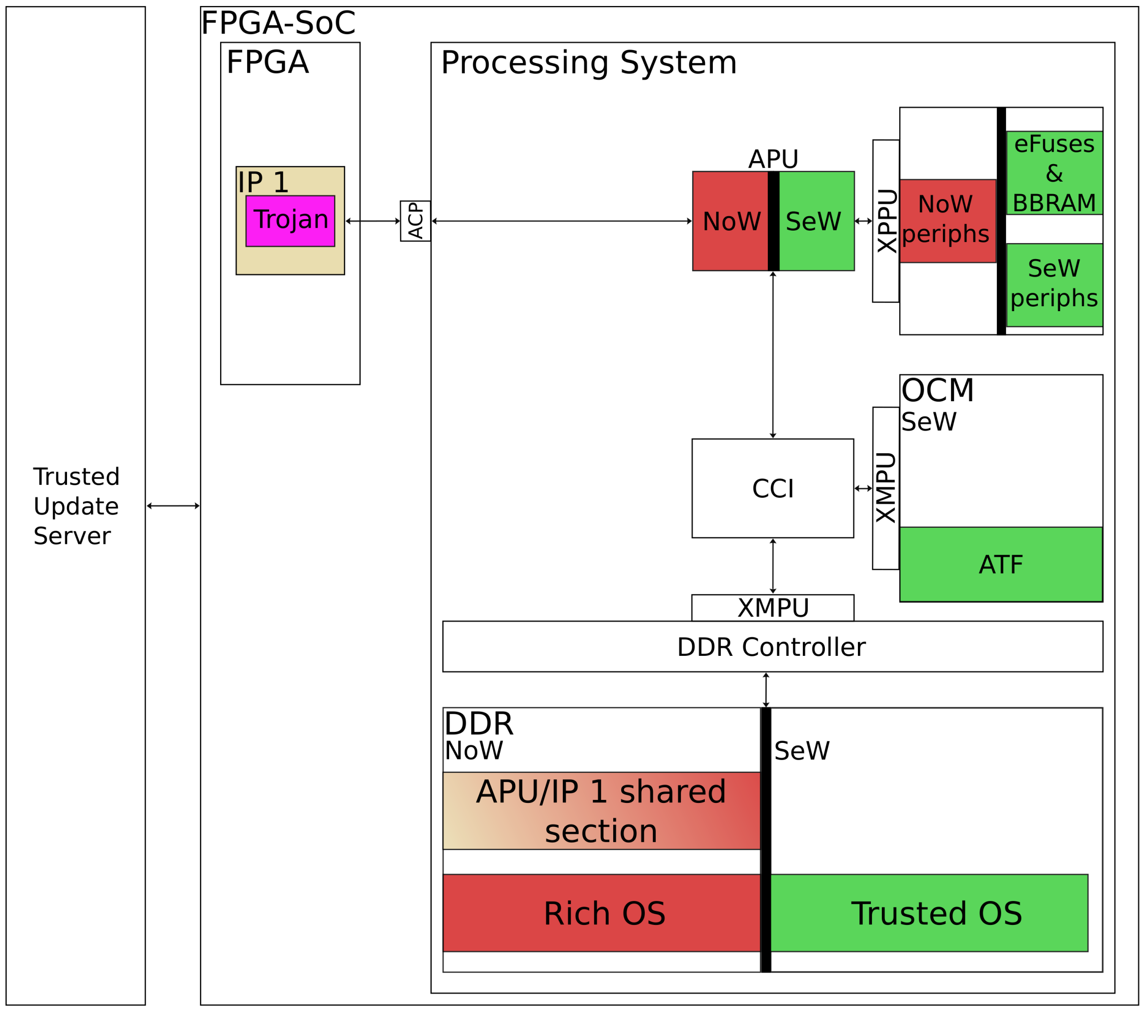


Figure 3.6 Secure boot with a configuration update obtained from a third party

accessible from the ACP. To verify the possibility of programming a BBRAM key from the ACP we performed the steps mentioned in algorithm 3.
By doing so, we found out that an ACP master is capable of programming an AES key into BBRAM. Since we assume that the device owner is decrypting the boot image with an AES key stored inside the eFuses, it is possible for a HT to reprogram BBRAM without preventing the device from booting. In section 3.3.4, we explain how this attack primitive can be used for booting an encrypted boot image, which is successfully decrypted with a device key that is not the one of the device owner.

3.3.4 Attack Description

In order to bypass the hardware root of trust secure boot configuration set by the device owner, the attacker needs to program an RSA public key hash to the second set of eFuses used for that purpose (see section 3.3.2) and optionally to program an AES key into BBRAM (see section 3.3.3). Programming an AES key into BBRAM is optional, because the hardware root of trust secure boot can work with boot images that are only authenticated, not encrypted. Both of these steps are performed by the HT contained inside IP 1 (see figure 3.6). After having done this, these keys are going to persist across device reboots. From a device owner point of view, the device is still booting without any

Algorithm 2 Programming of an RSA public key hash into eFuses

```
1: procedure eFuses RSA PPK programming(in RSA_PPK_HASH)
2:     $efuse\_wr\_lock \leftarrow 0xDF0D$                ▷ Unlock the eFuse controller
3:     $efuse\_cfg\_pgm\_en \leftarrow 1$                  ▷ Enable programming mode
4:     Set timing constraints and initialize sysmon.
5:     for ($i = 0; i < 384; i \leftarrow i + 1$) do
6:         if $RSA\_PPK\_HASH[i] == 1$ then
7:             $efuse\_pgm\_addr \leftarrow (row(RSA\_PPK\_HASH[i]),$
               $column(RSA\_PPK\_HASH[i]))$  ▷ Burn a fuse corresponding to the $i^{th}$ bit
8:         end if
9:     end for
10:    $efuse\_cfg\_pgm\_en \leftarrow 0$                  ▷ Disable programming mode
11:    $efuse\_wr\_lock \leftarrow 0$                      ▷ Lock the eFuse controller
12: end procedure
```

Algorithm 3 Programming an AES key into BBRAM

```
1: procedure BBRAM programming(in AES_KEY,in AES_KEY_CRC, out status)
2:     $bbram\_pgm\_mode\_reg \leftarrow 0x757BDF0D$     ▷ Put BBRAM in programming mode
3:     $bbram\_\{0..7\}\_reg \leftarrow AES\_KEY$  ▷ Write the 256 bit AES key in BBRAM registers
4:     $bbram\_aes\_crc\_reg \leftarrow AES\_KEY\_CRC$
5:     while $bbram\_status\_aes\_crc\_done \neq 1$ do
6:     end while
7:     if $bbram\_status\_aes\_crc\_pass == 1$ then
8:         $status \leftarrow success$
9:     else
10:        $status \leftarrow failure$
11:    end if
12: end procedure
```

errors in the hardware root of trust secure boot, because the hash of PPK0 and the AES key are still programmed inside the eFuses.

In order to take control of the device, the attacker must also be able to provide her own boot image to the device owner, in which she has specified to use the PPK1 for authentication and BBRAM as source for the device key. This can be achieved by tricking the device owner into downloading the boot image from malicious sources or by compromising the communication between the device owner and the server. Once the attacker has achieved the previous step, the device owner will then start the attacker's boot image successfully with the impression that the image is validated with the keys he programmed inside the device. In reality, these steps were realized with the keys that the attacker programmed in the non-volatile storage via the HT. Once the attacker has managed to boot her own image on the device, it is also possible for her to authenticate and decrypt partial bitstreams with the AES-GCM device key stored inside BBRAM. Again, the device owner is expecting the device key to be stored in the AES eFuses, however the compromised boot image has specified BBRAM as device key source.

3.4 Mitigations and Portability of the Attacks on other FPGA-SoC Platforms

In this section, we discuss possible countermeasures against the attack vectors described in sections 3.1.2 and 3.1.3 and the PoCs described in sections 3.2.1 and 3.3. We also evaluate if the attacks presented in this work might be applicable on other FPGA-SoC platforms.

3.4.1 Mitigations of the Attacks Presented in this Work

For the rest of this section, we use the ✓ symbol for indicating that a mitigation is effective, the ✗ symbol to indicate that it is not, and the ★ symbol to indicates that it partially addresses the issue. The notation ✓DMAA/✗secure boot indicates that a preventive technique effectively mitigates the DMAAs described in section 3.2.1 but not the attack against the hardware root of trust secure boot described in section 3.3.

Manual modification of the XMPUs'/XPPU's configuration (★DMAA/✓secure boot): The XMPUs/XPPU fail to isolate APU private memory/peripherals from an ACP master because of the mask value associated with the APU regions. We observed this vulnerability after using the Vivado Isolation Configuration (VIC) to configure the XMPUs/XPPU. Despite the existence of the VIC, the user can still configure the XMPUs'/XPPU's registers manually by modifying the psu_init.c file.

According to table 3.1, the 5^{th} Least Significant Bit (LSB) of the mask should be set such that the XMPUs/XPPU can distinguish a transaction originating from the APU and the ACP. Therefore, we modified the mask value 960 to 976 such that the master ID filtering can work properly and keep the FPGA-SoC working. During the research for a possible maske value, we observed that changing the mask value in the XMPUs registers could prevent the system from booting. This means that there is at least one incoming APU transaction for which the second condition in equation 3.1 is not met. Our hypothesis is that the "Core nn read/write" transaction ID is implemented with the 5^{th} LSB set and therefore by considering the master ID 128, which is stored for APU core 0 in the XMPUs registers, an incoming APU transaction is not going to be filtered with the result 128. The right approach consists in finding a solution which allows "core 0 exclusive read/write" and "core 0 read/write" to access APU memory regions while preventing it for the ACP. Given the ID encoding of these transactions (see table 3.1) we chose to modify the XMPU configuration according to table 3.3. With this approach, the ZU+ is booting successfully and meanwhile the ACP cannot access the APU private memory. Given the memory isolation described in table 3.2, we had to manually define two new memory regions for the APU (one for the SeW and one for the NoW).

Changing the XMPU configuration only is however not sufficient for solving the isolation problem fully. As depicted in figure 3.1, the XMPU cannot prevent an accelerator from accessing data located inside the L2 cache. Therefore, in order to protect the TEE from the DMAAs presented in this work, cache maintenance operations should be used after a SeW to NoW switch. We verified that the approach is also working for the XPPU. However, in that case replacing a mask value was sufficient (see table 3.4). Since an access to a peripheral from the ACP always goes through the XPPU, the caching problem encountered with the XMPU does not apply here.

Old configuration	*New configuration*
Core 0 (128, 960)	Core 0 (128, **976**)
Core 1 (80, 1022)	**Core 0 (160, 1008)**
Core 2 (197, 1023)	Core 1 (80, 1022)
Core 3 (98, 1023)	Core 2 (197, 1023)
	Core 3 (98, 1023)

Table 3.3 XMPUs configuration (ID, mask) for APU memory regions

Old configuration	*New configuration*
Core 0 (128, 960)	Core 0 (128, **976**)
Core 1 (80, 1022)	Core 1 (80, 1022)
Core 2 (197, 1023)	Core 2 (197, 1023)
Core 3 (98, 1023)	Core 3 (98, 1023)

Table 3.4 XPPU configuration (ID, mask) for the APU profiles

Use of another FPGA fabric to PS memory interface (✓DMAA/✓secure boot): The attack described in this work assumes a hardware accelerator interfacing DDR memory via the ACP. The ZU+ MPSoC provides alternative high performance memory interfaces. The ACP is however the only interface which enables a hardware accelerator to allocate cache lines inside the L2 cache.

Design of a specific isolation mechanism for the ACP (✓DMAA/✓secure boot): In contrast to most of the PS slave ports (see figure 3.1), ACP transactions are not filtered by an SMMU. As an alternative, Olson et al. [80] propose Border Control, a mechanism that can substitute an SMMU by sandboxing accelerators and protecting the memory from a malicious or misbehaving accelerator. Similarly, a special AXI wrapper as used in [46] can be an efficient mechanism for providing memory/peripherals isolation in a system where an untrusted hardware block interfaces memory via the ACP. This wrapper acts like a firewall and can be configured to prevent memory/peripherals access to a specified address space.

Definition of a security policy table for hardware accelerators (★DMAA/★secure boot): A hardware accelerator can arbitrarily configure the security of a transaction via the ARPROT[1] and AWPROT[1] bit. A firewall associated to a security policy table containing the security configuration of each master can ensure that a master generates transactions matching the security policy stored inside the table. This alone is not enough to ensure memory/peripherals isolation, however guaranteeing least privilege execution is a common practice in software and its extension to the FPGA fabric can help in blocking some attack scenarios.

Use OCM instead of DDR memory for the TEE (✗DMAA/✗secure boot): Depending on the compilation options, OP-TEE can be lightweight enough to fit in the 256 kB OCM. This choice enables even better isolation compared to DDR memory partitioning between the NoW and the SeW. In addition it can protect the TEE against Rowhammer [51] and cold-boot [36] attacks. However, we verified that the XMPUs also fails in preventing a hardware accelerator from accessing APU private OCM regions because of the same reasons explained in section 3.1.2.

Execute sensitive code in caches instead of DDR (✗DMAA/✗secure boot): CPU bound execution relies on having critical data and code only in the processor registers and caches, not in the DDR memory. Originally designed to mitigate cold-boot and DMAA, the Sentry [17] and CaSE frameworks [109] enable the execution of a critical application inside the cache only via the ARM lockdown features. The protection against a DMAA adversary is further achieved by executing cryptographic operations in an isolated environment provided by ARM TrustZone. Both solutions, however, are ineffective against the attacks presented inside this work; an adversary that can perform DMAA via the ACP can indeed snoop data in the processor L1 and L2 caches and arbitrarily set the security bit of a transaction.

Use of hardware support for secure key storage and cryptography (★DMAA/✗secure boot): This work uses the Rich OS RootFS secure storage features provided by OP-TEE (see appendix A.0.2). Alternative possibilities on a ZU+ FPGA-SoC are BBRAM or eFuses. However, none of these features can help to protect against the PoC described in section 3.2.3 if the AES-GCM decryption is executed in software. An effective mitigation against our attack is to perform the AES decryption with hardware support. The ZU+ boards already contain a dedicated AES-GCM module that can be used for this purpose, however, the integration of this module inside a TEE requires additional work as described in chapter 6.

Program both set of eFuses used for storing the RSA PPK hash values (✗DMAA/✓secure boot): The attack presented in section 3.3 requires that only one PPK is programmed into the eFuses. If the two sets of eFuses are programmed, the attacker cannot program her own key into the second set. Xilinx recommends programming both PPK hashes before fielding a system but also specifies that this is not required [107]. Programming only one of the PPK hashes also has some advantages from a security point of view. If the private key of a boot image provider gets compromised, it is possible to revocate the corresponding public key hash and to program a new one into the device.

Use the encrypt only secure boot (✗DMAA/★secure boot): An alternative to the hardware root of trust secure boot is the encrypt only secure boot. This scheme requires that all partitions contained in the boot image are encrypted and authenticated with AES-GCM. Xilinx specified that this secure boot mode is only compatible with an AES-GCM authentication with a key stored in the eFuses [99]. Therefore, a variant of the attack against secure boot for this particular scheme is not possible. However, besides the attack considered in this work, the encrypt only secure boot is also vulnerable to boot header manipulation attacks [20].

Prevent the load of un-authenticated bitstream (★DMAA/★secure boot): The attacks presented in this chapter rely on the presence of a HT inside the FPGA logic. A HT can either be hidden in a bitstream obtained from a third party or be inserted by partially reconfiguring the FPGA at runtime with a malicious partial bitstream. As explained in section 2.4.4.1, Xilinx allows the reconfiguration of the FPGA with un-authenticated bitstream even after a secure boot of the FPGA-SoC. In chapter 6, we present a framework which forces the usage of authenticated bitstream load, which contributes to partially mitigate the attack vectors presented in this chapter.

3.4.2 Portability of the Attacks on other FPGA-SoC Platforms

The PoC described in section 3.2.1 was tested on a Xilinx ZU+ MPSoC ZCU102 Evaluation Kit (Production Silicon). To verify the portability of the PoC on other ZU+ boards, the same design was implemented for the ZCU104, ZCU106, and Ultra96-V2 variants. Tests on these boards were not directly performed, instead, a comparison of the generated psu_init.c file with the file generated for the ZCU102 reveals that the APU private memory regions are configured with the same mask. Similarly, the APU apertures are configured with a mask that does not allow filtering between APU and ACP. The two previous observations make the attacks presented in this work portable to other ZU+ boards.

Stratix 10 [43] is the Intel equivalent to the Xilinx ZU+. However, this architecture does not contain an ACP bus interface to the ARM Cortex-A53. An inspection of the technical reference manual reveals that all FPGA to processor memory interfaces present on the Stratix 10 go through an SMMU. Additionally a system of firewalls enables the protection of memory and peripherals. To take advantage of this architectural specificity, the user must nevertheless be careful when selecting the order in which the FPGA fabric and the processor are booting. A good prevention of DMAAs from malicious logic consists in configuring the Cortex-A53 and the SMMU before loading the FPGA fabric. By doing so, the user can effectively prevent the FPGA fabric to access processor private memory. The opposite configuration is insecure and could lead to DMAAs scenarios during the boot of the processor.

Concerning authentication, Stratix 10 relies on Elliptic Curve Digital Signature Algorithm (ECDSA) signature verifications with a root public key (equivalent of the PPK on the ZU+) hash stored in the eFuses [42]. Only one root public key can be programmed into a Stratix 10 device and root key revocation is not possible. Therefore, the attack performed in section 3.3 seems not to be applicable on this architecture.

3.5 Summary

This chapter shows two approaches for compromising an FPGA-SoC via malicious hardware. The first one consists in manipulating memory in order to bypass some security mechanisms of a TEE. In contrast to previous works [15, 46, 61], our experiments were carried out on an FPGA-SoC based on the modern ZU+ architecture from Xilinx. This architecture contains more mechanisms for memory and peripherals isolation inside the FPGA-SoC. Despite the presence of more sophisticated separation mechanisms, we show that malicious hardware can still compromise memory via the ACP. This interface is usually considered for scenarios where a hardware accelerator requires fast and cache coherent memory access.

The second approach consists in the manipulation of the FPGA-SoC peripherals via malicious hardware hidden inside an accelerator which uses the ACP. Our experiments reveal an issue in the peripheral protection unit which enables the malicious logic to access peripherals it is not supposed to. We use this vulnerability to demonstrate a proof of concept attack in which an attacker can bypass the secure boot configuration set by a device owner and boot her own authenticated software. This is achieved by programming an RSA public key hash into the eFuses and an AES key into BBRAM via malicious logic.

Before using the ACP for hardware accelerators requiring fast and cache coherent memory access, we strongly recommend to perform a security risk assessment considering our detected attacks. If the usage of the ACP is necessary, the attack vectors presented in this work can be mitigated by manually changing the configuration of the XMPUs and XPPU registers and flushing the L2 cache when switching from the secure world to the normal world. As a more practicable solution, we would instead recommend the use of sandboxing for ACP accelerators [80], or to use a wrapper as done in [46].

3.6 Responsible Disclosure

Xilinx has been informed about the XMPU vulnerability we discovered in July 2019 and responded via the Answer Record 72654 [100]. The memory isolation issue that we observed is due to an unrestricted access to memory located inside the L2 cache together with a configuration of a particular (mask, ID) value in the XMPUs' registers after the use of the Vivado Isolation Configuration [1]. The XMPUs configuration issue extends the issue further and enables the ACP to access data which is not located inside the APU's L2 cache.

In addition to the first disclosure, we have informed Xilinx about the extension of the ACP isolation issue with the peripherals of the FPGA-SoC. Xilinx recognized the second issue on January 26th 2021, with no particular comments from their side. As a general recommendation, we would recommend a careful usage of the ACP in security critical designs requiring isolation. This recommendation has also been added to Xilinx ZU+ documentation, which enable a user to be easily informed about our findings.

4 Fault Attacks on a CPU through FPGA Logic

Besides the memory and peripherals manipulation attacks described in chapter 3, fault attacks are another class of attacks that can alter the execution of software on FPGA-SoCs. In comparison to the attacks presented in chapter 3, the fault attacks presented in this chapter require a less powerful adversary model. More precisely, we demonstrate that fault attacks targeting the CPU of an FPGA-SoC can be performed remotely by relying on a voltage drop that is generated via dedicated FPGA logic. This attack primitive corresponds to the remote electrical threat introduced in section 2.4.4.4. Our experiments demonstrate the possibility of compromising the data transfer from external DDR memory to the processor cache hierarchy. Furthermore, we were also able to fault and skip instructions executed on an ARM Cortex-A9 core. The FPGA based fault injection is precise enough to recover the secret key of an AES T-tables implementation found in the *mbed TLS* library.

The remainder of this chapter is organized as follows: section 4.1 contains the threat model and background information related to fault attacks via FPGA logic. Section 4.2 describes the power-hammering circuit used in this chapter and the fault model. Section 4.3 presents the experimental setup and the results. Section 4.4 discusses possible countermeasures and future work. Finally section 4.5 contains a summary of this chapter.

The results presented in this chapter were part of the publication *FPGANeedle: Precise Remote Fault Attacks from FPGA to CPU in the 28th Asia and South Pacific Design Automation Conference - ASPDAC 2023* [33].

4.1 Remote Fault Injection on Software and Threat Model for the FPGA-SoC Scenario

This section provides an overview of existing works related to remote fault injection on CPUs and describe the threat model considered in this chapter.

4.1.1 Remote Fault Injection on Software

In the last decade, many research works demonstrated the severity of remote fault injection on software. In 2014, Kim et al. [51] described the Rowhammer attack, which enables the flipping of bits in DRAM modules by repeatidely accessing adjacent rows located near a row to attack. This attack has been demonstrated to be feasible from the browser via Javascript [34] and even via the injection of network packets with no attacker code running on a machine [65]. Despite the latest technological advancements of DDR4 to mitigate this threat, the attack is still possible on recent DDR4 DRAM modules [49, 65].

Besides the Rowhammer attack, DVFS, a performance optimization feature found on ARM and Intel processors has been also shown to be usable for fault attacks. By lowering the voltage of a CPU and/or increasing its frequency, fault attacks were shown to be possible on ARM [91] as well as Intel processors [77].

Remote fault attacks were also considered on FPGA-SoC platforms, but generally the attack has targetted a victim located in the FPGA fabric (see section 2.4.4.4). In parallel to our work, Mahmoud et al. [69] demonstrated the possibility of injecting faults on software executing on an FPGA-SoC platform by combining DVFS, with an increase of the CPU frequency together with voltage drop generated from the FPGA logic. In this chapter, we use a similar threat model and attack target (see section 4.1.2) and achieve fault injection on a CPU by relying on fault injection via FPGA logic while running the CPU at its default frequency.

4.1.2 Threat Model

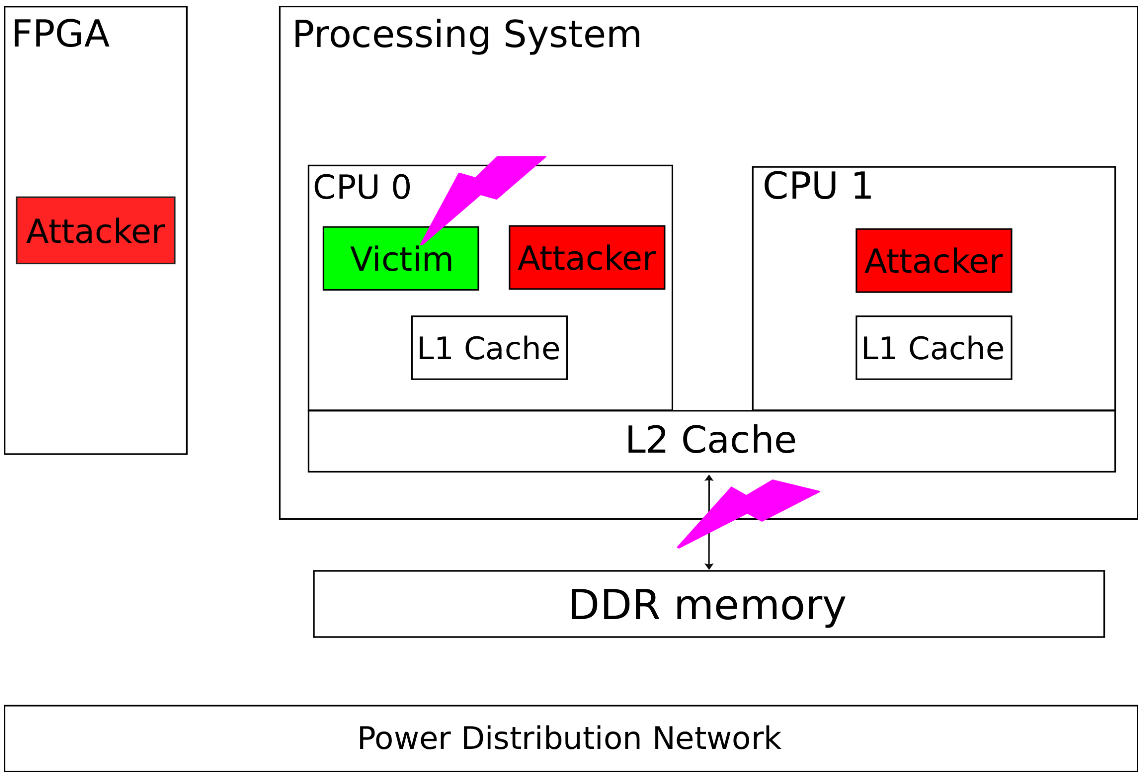


Figure 4.1 Threat model for the FPGA to CPU fault attacks

In this chapter we consider an FPGA-SoC platform where an FPGA and multiple processing units are located within the same SoC. We assume that the SoC contains a PDN which is shared among the FPGA and the processing units (see figure 4.1). Furthermore, we assume that the attacker can execute code on one of the processing units and reprogram partially the logic located inside the FPGA. The attacker can either be a process located on the same core of the victim or on another core. Security mechanisms from the OS or ARM TrustZone [6] guarantee an isolation between the attacker and victim process. The goal of the attacker is the injection of faults in a victim process through the exploitation of the shared PDN. This is achieved by generating voltage drops within the FPGA logic that can affect the correct operation of processor instructions or fault the data transfer from DDR memory to the processor cache hierarchy. In contrast to the work of [69], we make no special assumption regarding DVFS. The attacker is not capable of overclocking the processor cores, which can be used for making the fault injection easier or even be sufficient for injecting faults [91]. For the two platforms considered in this chapter, the processor cores are run at the default frequency settings recommended by the respective FPGA-SoC manufacturer.

4.2 Fault Injection Circuit and Attack Targets

This section first introduces the power-hammering circuit based on the PRESENT cipher that is used in this chapter as well as the fault injection parameters. In a second time, the fault injection targets that are used for the evaluation of the fault model are described.

4.2.1 Power-hammering Circuit and Parameters Description

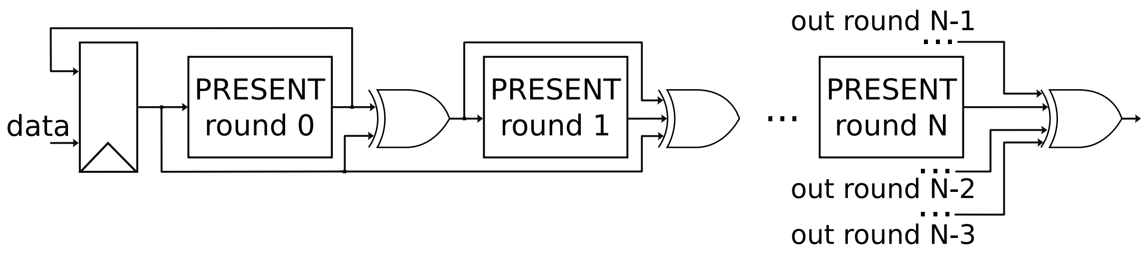


Figure 4.2 Power-hammering circuit based on a chain of PRESENT rounds

In our threat model introduced in section 4.1.2, an attacker can reprogram partially the FPGA with her own logic while letting the victim design unmodified (see figure 4.1). In this chapter, the attacker logic is the power-hammering circuit based on rounds of the PRESENT cipher, as depicted in figure 4.2. PRESENT [12] is a lightweight block cipher which has a similar operating mode as AES on a smaller internal state. One PRESENT round consists of applying an *AddRoundKey*, a *sBoxLayer*, and a *permutationLayer* to the 64 bit state. Our PRESENT power-hammering circuit contains a chaining of PRESENT rounds and XORs of 4-previous round outputs between two consecutive rounds to generate glitches, which is known as an efficient technique for consuming significant dynamic power in FPGAs [58]. A similar design using AES instead of PRESENT rounds was introduced in [82]. By using PRESENT rounds instead of AES rounds, we verified through initial experiments that a more precise fine tuning of the voltage drop can be obtained. This is due to the fact that a PRESENT round is implemented with fewer logic resources and generate less switching activity in comparison to an AES round.

In order to adjust the voltage drop duration and magnitude and exploit possible resonance effect of the PDN and voltage regulators, an attacker can control the following parameters:

- The number of rounds per PRESENT power-hammer and the number of PRESENT power-hammer instances
- The activation delay offset after a trigger signal
- The total duration of the fault injection
- The period of the enable signal

- The duty-cycle of the enable signal

By finding optimal parameters, we were able to inject faults on the data transfer from DDR to the processor's cache hierarchy (section 4.2.2) and faulting instructions on the processor (section 4.2.3) for two FPGA-SoC platforms using ARM Cortex-A9 processors.

4.2.2 Faulting the Data Transfer from DDR to the Processor Cache Hierarchy

To demonstrate the possibility of faulting data during the transfer from DDR to the cache hierarchy, we tried to modify the values written inside an array via a fault injection. The victim pseudo-code used for verifying the validity of this attack vector is contained in listing 4.1. The analysis of the results shows the possibility of producing multiple faulty outputs on a word used for filling the array. We observed 4 or 8 consecutive faulty words, depending if a L1 or L2 cache line was affected. Among those faulty outputs, we observed faults affecting a single byte of a 32-bit word on the Pynq-Z1 platform used in this chapter (see section 4.3). We used this particular type of fault for implementing a DFA on an AES T-tables implementation in section 4.3.3.

```
#define ARRAY_SIZE 1024
#define FILL_PATTERN 0xFFFFFFFF
...
uint32_t array_attacked[ARRAY_SIZE];

fill_array(array_attacked,FILL_PATTERN);
flush_caches();
// Attacker starts injecting faults from here
verify_fill_pattern(array_attacked,FILL_PATTERN);
...
```

Listing 4.1 Faulting data transfer from memory to the L2 cache

4.2.2.1 Application Usecase: Differential Fault Attack on AES

To demonstrate the capabilities of FPGANeedle, we chose to use the DFA on AES proposed by Piret et al. [81]. The AES implementation we attack is based on T-tables, as in the established *mbed TLS* library [63], optimized for 32-bit architectures. This T-table based implementation abstracts the AES round transformations *SubBytes (SB)*, *ShiftRows (SR)*, and *MixColumns (MC)* into 4 T-tables. An AES round is then implemented via 16 T-table lookups (4 per T-table) and XORs. We will summarize the attack steps for discovering four key-bytes, this can be easily extended to the whole key as in [81].

Fault Model The attack of Piret et al. assumes a manipulation of a single byte after the *MC* operation of round 8 and right before the *MC* transformation of round 9 as shown in figure 4.3. This fault propagates to a full AES column due to the *MC* transformation of round 9. Finally, the *SR* from round 10 leads to 4 faulty bytes in C.

Fault Attack In order to recover four key-bytes an attacker needs to obtain a tuple of correct and faulty ciphertext. As the attacker does probably not know which bytes has been attacked it becomes necessary to compute the 4×255 possible output differentials of *MC* of round 9, which only needs to be done once. Afterwards, the attacker is required to test 2^{32} possible key candidates of the last round key which are influenced by the

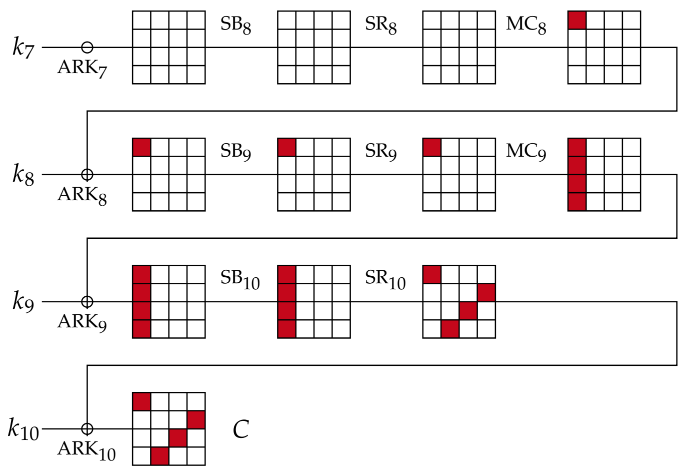


Figure 4.3 Fault propagation in the AES state matrix

fault injection of the faulty ciphertext which lead to valid output differentials of the *MC* operation of round 9. After processing of the first ciphertext tuple, a unique key candidate for the four bytes of the last round key can be found. If that is not the case, the process is repeated with another tuple of faulty and correct ciphertext.

4.2.3 Faulting Instructions on the Processor

```
#define N 500
#define NUMBER_OF_NOPS 100
...
int j = 0;
/*Attacker starts injecting faults from here*/
NUMBER_OF_NOPS*nops();
j++;
...  // N consecutive j++ instructions
j++;
NUMBER_OF_NOPS*nops();
...
```

Listing 4.2 Faulting add instructions

```
#define N 5
#define MULTIPLIER 11
#define NUMBER_OF_NOPS 500
...
uint32_t j = 3;
// Attacker starts injecting faults from here
NUMBER_OF_NOPS*nops();
j *= MULTIPLIER;
... // N consecutive j *= MULTIPLIER
j *= MULTIPLIER;
NUMBER_OF_NOPS*nops();
...
```

Listing 4.3 Faulting a victim code based on a multiplication instruction

4.3 Experimental Setup and Results

This section presents the two experimental setups considered in this chapter and the fault injection results obtained during listings 4.2 and 4.3 execution.

4.3.1 FPGA-SoC Setups

In this chapter, two FPGA-SoCs both containing a dual-core ARM Cortex-A9 processor are used. The first platform is the Pynq-Z1 board from Digilent. This platform is based on the Zynq-7000 SoC and features a Xilinx Artix-7 FPGA. We run the Cortex-A9 at 666 MHz, which is the maximal frequency recommended by the manufacturer for this particular Zynq-7000 SoC. The second setup is the Terasic DE1-SoC which is based on the Intel Cyclone-V SoC. The Cortex-A9 on the Intel board runs at its default frequency of 800 MHz.

The power-hammering parameters were chosen such that faults can occur while having a low number of crashes. This is particularly important for a fault injection attempt with a Linux system running. According to previous works [56, 69], faults can be observed with a minimum number of crashes when the power-hammering activation frequency is located in the MHz range with a duty-cycle located within the 30% and 40% range. We verified this for the Pynq-Z1 platform. For the Terasic DE1-SoC, fault injection was performed with a lower activation frequency and a longer total duration. We used the parameters listed in table 4.1 for the fault injection experiments performed in section 4.3.2 and 4.3.3. To evaluate the voltage drop resulting from the activation of the power-hammering circuits, we used one TDC sensor consisting of 16 CARRY 4 primitives as delay-line and 2 Look-Up Tables (LUTs) as initial delay (see section 2.4.4.4 for an explanation of the working principle of a TDC). Figure 4.4 depicts the TDC sensor propagation in comparison to a baseline during the activation of the power-hammering circuit on the Pynq-Z1 platform. The periodic activation of this circuit leads to three consecutive voltage drops within a 450 cycles duration. After power-hammering, the voltage regulators contained inside the PDN gradually lead to the return to the baseline.

Platform	FPGA clock freq. (MHz)	PRESENT power -hammer (number, rounds)	Duration (cycles)	Activation freq. (MHz)	Duty cycle
Pynq-Z1	222	(13,16)	450	1.48	40 (bare-metal) 30-40 (best 31) (Linux)
Terasic DE1-SoC	250	(14,13)	10 000	0.408	99 (bare-metal)

Table 4.1 Fault injection parameters chosen during the experiments

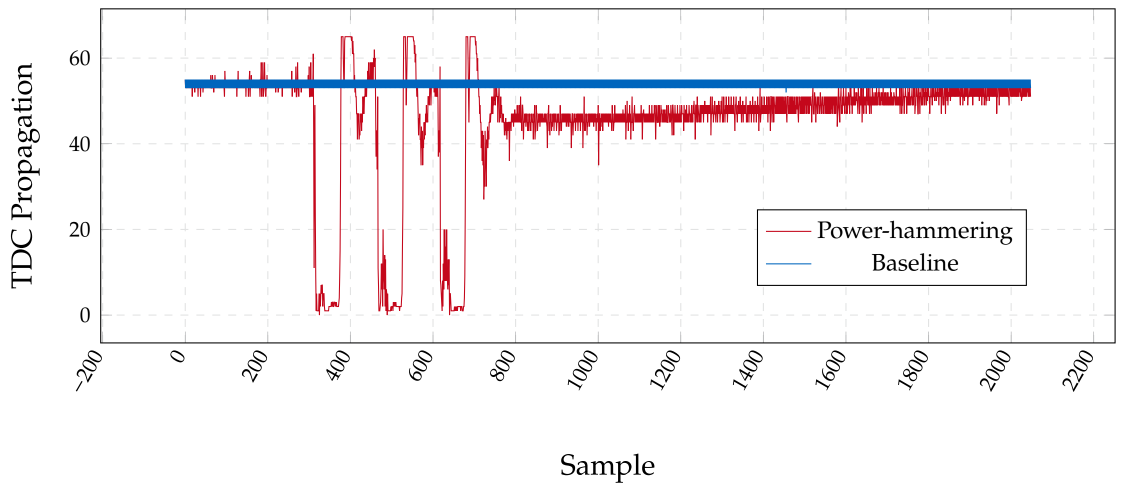


Figure 4.4 Sensor delay measurements during the activation of 13 PRESENT power-hammer of 16 rounds on the Pynq-Z1

Faulty output range	[0 : 450[	[450 : 500[	[501 : 599[	[600 : 1000[	[1057739 : 1058020[
Distinct faulty outputs	12	35	6	8	5
Faulty output distribution	14	868	102	11	5

Table 4.2 Faulty outputs distribution during listing 4.2 execution on the Pynq-Z1

4.3.2 Faulting Processor Instructions

4.3.2.1 Faulting Additions

For both platforms, we evaluate the distribution of 1000 faults obtained during multiple executions of listing 4.2. The results for the Pynq-Z1 platform are depicted in table 4.2. 88.2% of the faulty outputs are located within the range [0 : 499], which correspond to the skip of one (42.8% of the faulty outputs) or several executions. We also observe the value 501 in 9.3% of the faulty outputs, resulting from the double execution of an add instruction. Besides the skip and double execution of additions, we also obtain faults outside of the range [490 : 502]. Our hypothesis is that these results correspond to a fault on the add instruction and not in the Program Counter (PC) register or the processor's pipeline. Table 4.3 contains the results of the same experiment for the Terasic DE1-SoC. Similarly, we observe 37% of the faulty outputs in the range [497 : 499] and 501 in 0.2% of the faulty outputs. Besides the faults in the range [503927 : 504058], the faults observed on the Terasic DE1-SoC seem also to be caused by a fault in the PC register or the processor's pipeline.

Faulty output range	[0 : 450[	[450 : 500[	]500 : 600]	[503927 : 504058]
Distinct faulty outputs	39	14	11	4
Faulty output distribution	150	463	23	364

Table 4.3 Faulty outputs distribution during listing 4.2 execution on the Terasic DE1-SoC

4.3.2.2 Faulting Multiplications

Similarly to the additions, we evaluated the distribution of 1000 faulty outputs during multiple executions of listing 4.3 on the two FPGA-SoC platforms used in this chapter. The results for the Pynq-Z1 are contained in table 4.4. A total of 56 different faulty outputs can be generated. To evaluate the effect of a fault during listing 4.3, we grouped the faulty output values according to their greatest power of 11 divisor. This classification can give an insight regarding the number of multiply instruction skipped. According to the faulty output distribution, most of the faults occur on an intermediate multiply computation with the end result still being dividable by a power of 11. Instructions skips could be further confirmed by observing faulty outputs which correspond to an intermediate result during the exponentiation. Besides instruction skips, we observe faults which are not dividable by 11 and the value 0 which might come from a multiplication with 0 during the exponentiation process due to a fault on one of the operands. The distribution of the faulty outputs for the Terasic DE1-SoC is shown in table 4.5. For this setup, we obtained 28 different faulty outputs. The analysis of the results similarly reveals the possibility of skipping as well as faulting multiply operations on this platform. In summary, the results of this experiment suggest the possibility of faulting the execution of a multiply operation as well as the PC register or the processor's pipeline.

Faulty output value OR max (11^N) divisor	0	3	11	11^2	11^3	11^4	11^5	11^6	others
Distinct values	1	1	6	12	11	11	1	1	12
Values distribution	141	17	29	279	314	179	1	1	39

Table 4.4 Faulty outputs distribution during listing 4.3 execution on the Pynq-Z1

4.3.2.3 Fault Model Deduced from the Experiments

The results presented in this section suggest the possibility of faulting additions and multiplications as well as skipping instructions. This could be achieved due to a fault occurring within the processor's pipeline or in the PC register. Skipping instructions is a powerful fault model which has been used on an ARM Cortex-A9 in previous work to implement privilege escalation in Linux [93]. A future work should investigate if our fault

Faulty output value OR max (11^N) divisor	0	3	11	11^2	11^3	11^4	11^5	11^6	others
Distinct values	1	0	4	1	6	7	0	0	9
Values distribution	1	0	241	3	276	281	0	0	198

Table 4.5 Faulty outputs distribution during listing 4.3 execution on the Terasic DE1-SoC

injection setup is capable of producing similar results on the two platforms used in this chapter.

4.3.3 Differential Fault Attack on AES

To demonstrate the severity of the fault attack vector described in section 4.2.2 we implemented a DFA on the AES T-tables implementation used in *mbed TLS* [63]. Through our experiments, we figured out that faults could occur during an AES encryption if at least one memory block containing T-tables (32 Bytes in our set-ups) is fetched from the DDR memory to the processor caches. The readback of the T-tables in the event of a faulty encryption reveals a modification of the T-table values. This is related to the possibility of faulting a read from DDR memory as described in section 4.2.2. The probability of obtaining faults is dependent on the number of T-table elements which are already in the L1 or L2 caches before an encryption. An analysis of the obtained faults reveal that single and multi-bytes faults as well as faults affecting the whole AES state can be obtained through a faulty T-table lookup. In conformity to our fault-model described in section 4.2.2.1, only single byte faults occurring between *MC* in round 8 and round 9 are used for the key elimination. This specific kind of fault can be easily identified by observing a 4 bytes difference between the faulty and non-faulty ciphertexts at specific locations. For the rest of this section, we assume that no T-table elements are contained inside the L1 and L2 caches before an AES encryption.

4.3.4 Application to the T-tables Implementation Found in *mbed TLS* in a Bare-Metal Setup

The DFA in a bare-metal setup is done on the Pynq-Z1 board by using the parameters contained in table 4.1 and by varying the activation offset in a window, such that the activation offset summed with the activation duration corresponds to a fault injection point located between *MC* in round 8 and round 9. For each plaintext, we tested 15 different fault injection configurations and took 10 measurements for each configuration. By using this strategy with 100 plaintexts, we were able to recover 10 AES keys with a 100% success rate. To evaluate the fault injection quality with the chosen power-hammering parameters, we computed the ratio of exploitable faults in comparison to the total number of faults (see table 4.6).

4.3.5 Extension of the Attack in a Linux Setup

In comparison to the bare-metal setup, a Linux system is more prone to crashes during a fault injection attempt. Through our experiments, we tried to find fault injection param-

	Total number of faults	Number of exploitable faults	Ratio
Worst	595	40	6.72%
Average	620	61	9.88%
Best	607	74	12.19%

Table 4.6 Fault injection results across 10 random AES-128 keys. Best and worst results are evaluated by considering the ratio

eters which enable us to take a maximum number of measurements and observe faults during AES encryptions before the crash of the Linux OS. We used a STM32F3 discovery board to remotely reset the Pynq-Z1 after detecting a crash of the board and vary the power-hammering parameters after 10 crashes. By using this approach, we find the fault injection parameters contained in table 4.1 as a configuration which enabled us to take up to 243 measurements with 7 faulty ciphertexts. Among our measurement sets, we observed at most one exploitable fault before the crash of the Linux OS, which is not sufficient for a DFA.

4.4 Discussion and Future Work

The results presented in section 4.3 show the possibility of injecting faults for two FPGA-SoC platforms using an ARM Cortex-A9 processor via a power-hammering circuit based on rounds of the PRESENT block cipher [12]. Through our experiments, we tested several parameters and opted for a configuration allowing the injection of faults in a short duration. This is particularly important for attacking code snippets that have a short execution time, such as an AES encryption. As a future work, the power-hammering parameter space can be explored further in order to achieve a fault injection in Linux, where avoiding crashes is more challenging than on a bare-metal setup. In parallel to this parameter optimization criteria, reducing the size of the power-hammering circuit in the FPGA is another interesting direction of research.

Through our experiments, we demonstrated that fault injection can compromise the data transfer integrity from external DDR memory to the cache hierarchy (see section 4.2.2). This has been used for mounting a DFA on an AES T-tables implementation for the Pynq-Z1 platform. Besides this fault model, we also show the possibility of attacking addition and multiplication instructions running on an ARM Cortex-A9 and compromise the processor execution flow for the two platforms considered in this chapter. A future work should investigate more in detail which further instruction can be affected by our on-chip fault injection setup and demonstrate more advanced attacks exploiting the skip of processor instructions [93]. Furthermore, it should be investigated whether more recent variants of ARM cores contained in high-end FPGA-SoCs from Xilinx and Intel are vulnerable to the attacks presented in this chapter.

Another direction of research are countermeasures. Related work has shown the difficulty of detecting power-hammering circuits at the bitstream level [89, 54]. Alternatively, the system level security approach presented in chapter 6 could be used for making the insertion of power hammering circuits through partial reconfiguration more difficult (see attack vector introduced in section 2.4.4.1). This consists of restricting the trust assumptions of the FPGA reconfiguration interfaces so that only authenticated bitstream can be loaded in the FPGA during partial reconfiguration. We believe that online detection and prevention of attacks via on-chip sensors as done in [82, 78] should be further investigated. The challenge in that line of work is a fast detection and response to an attack. An interesting approach for raising the reaction time deadline would be to temporally reduce the clock of the processor via DVFS in the event of a fault detection, which is a similar approach to the one presented in [67] or the reliability optimization for mainframe processors [60].

4.5 Summary

In this chapter we demonstrate the generation and possible exploitation of faults induced by FPGA logic on an ARM Cortex-A9 processor for two FPGA-SoC platforms. Our experiments reveal that the data transfer integrity from external DDR memory to the processor's cache hierarchy can be compromised via a voltage drop. Furthermore, we demonstrate the possibility of disturbing the execution flow of the processor by skipping or compromising the result of instructions. These attack vectors are used for recovering an AES key via a DFA on the AES T-tables implementation from *mbed TLS* that we used in a bare-metal setup. Future work should optimize the fault injection parameters for mounting a complete DFA on Linux and consider other attack scenarios such as a privilege escalation.

5 Electrical Covert Channel between CPU and FPGA

As mentioned in section 2.4.4.4, the capability of observing and modulating the PDN lead to the threats of information leakage and fault attacks. Remote fault attacks were demonstrated in chapter 4. In this chapter, we describe the implementation of a covert channel implemented via the PDN between a CPU and an FPGA.

We show that the PDN can be effectively modulated from the CPU via a sequence of divisions and `nanosleep` operations running in a single thread on one CPU core. In addition to having a low demand on the CPU usage, our covert channel also achieves a high transmission rate of up to 16.7 kbit/s and a corresponding bit error rate of 2.3% without requiring an explicit synchronization between the transmitter and the receiver. As an application usecase, we discuss the usage of the covert channel for the activation of a HT. The covert activation of a HT complements the results presented in chapters 3 and 4, where we demonstrated possible usages of a HT for compromising the software executed on one of the CPUs of an FPGA-SoC.

The remainder of this chapter is organized as follows: section 5.1 explains the background information such as the threat model and the vulnerabilities resulting from a shared PDN in FPGA-SoCs. Section 5.2 describes the implementation of the power covert channel between the CPU and the FPGA. Section 5.3 characterizes the covert channel implemented in this chapter. We compare our covert channel to similar work and discuss its limitations and countermeasures in section 5.4. Finally, section 5.5 summarizes the results presented in this chapter.

5.1 Background

This section contains the background information related to this chapter. It first introduces the threat model and the assumptions we made for the implementation of the covert channel. Finally some information regarding Manchester code, the code we used for encoding a message through the PDN are introduced.

5.1.1 Threat Model

The attacker model considered in this chapter is depicted in figure 5.1. We consider a scenario where an attacker capable of executing unprivileged code on one CPU core wants to communicate with a HT module located in the FPGA fabric. The HT purpose is to mount an attack on the CPU, which cannot be mounted from unprivileged code executed on the CPU due to the isolation mechanisms of the operating system, lack of attack primitives available in software or due to primitives that require higher privileges for being used. Such attacks include direct memory access attacks, which have been shown to be feasible on FPGA-SoCs from the FPGA fabric because of a poor protection of memory interfaces that are accessible from the FPGA logic (see [46] and chapter 3 of this thesis). Another scenario could be, an unprivileged software adversary who wants to mount side-channel through power sensors implemented in the FPGA logic [28] or use special circuits such as ROs for generating voltage drops that can compromise software execution on a CPU (see chapter 4 of this thesis).

For executing one of the previously mentioned attacks, an attacker can reprogram partially the FPGA from the userspace via the *libdfx* library [102], that interacts with the *FPGA Manager* kernel driver [105]. On Xilinx FPGA-SoCs, the FPGA reconfiguration interfaces are considered as trusted under secure boot assumption [101]. Therefore, the runtime reconfiguration of the logic does not force the usage of encrypted or authenticated bitstream load even after secure boot. This relaxed trust assumption contributes to facilitate the HT insertion for those platforms. Once the attacker has placed a malicious IP inside the FPGA logic, she needs to communicate with the FPGA for activating the HT in a covert way.

In an FPGA-SoC, a CPU core can use the AXI bus for communicating the activation signal, however this communication channel is system wide accessible and is therefore not suitable for activating a HT covertly. Furthermore, a suspicious communication on the AXI bus can be easily blocked by a firewall placed on the AXI bus as proposed in [46]. For that purpose, we present a methodology to covertly communicate between the CPU and the FPGA with the help of a PDN modulator software running on the CPU (see section 5.2.2) and a decoding logic implemented within the FPGA logic (see section 5.2.3).

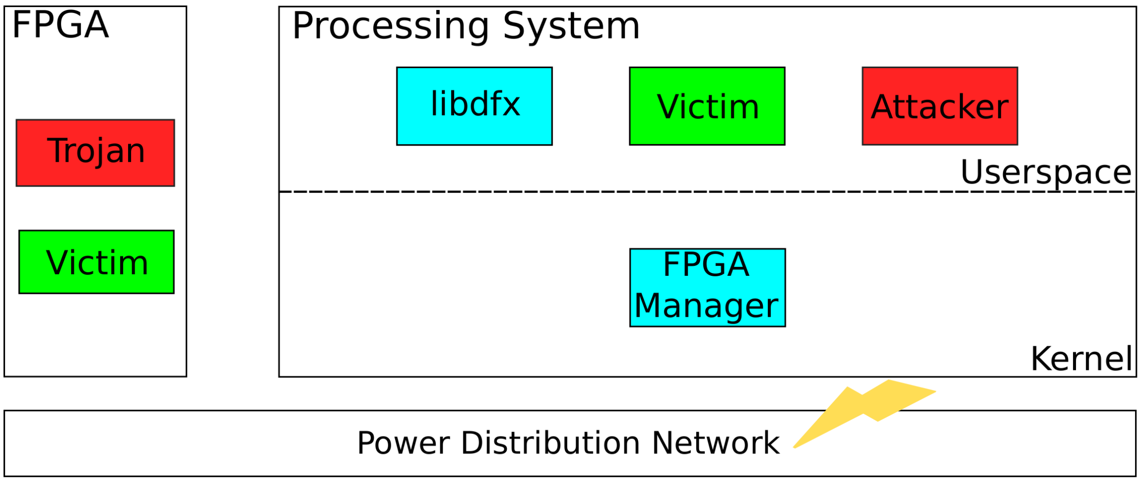


Figure 5.1 Threat model for the CPU to FPGA power covert channel

In summary, our covert channel can be utilized for activating the HTs used in the attacks presented in the chapters 3 and 4 of this thesis by performing following steps:

- Partial reconfiguration of the FPGA logic with the Trojan circuit (if not already included in the fixed partition of the FPGA) and insertion of the receiver logic in a reconfigurable partition of the FPGA.
- Covert activation of the Trojan by transmitting an activation pattern in the PDN through the execution of specific instructions on the CPU (see section 5.2.2).
- Decoding of the transmission pattern encoded in the PDN and activation of the Trojan for an FPGA assisted attack on the CPU (see section 5.2.3).

5.1.2 Manchester Code

The transmission of a message inside the PDN requires an encoding scheme. For FPGA platforms, the On-Off keying encoding has been demonstrated to be efficient for a temperature covert channel [92]. In the context of a PDN covert channel, this simple encoding scheme has also shown to be efficient for a pure FPGA-to-FPGA power covert channel [27]. For a more generic power covert channel implementation involving FPGAs, Giechaskiel et al. suggest the usage of the Manchester code which is less prone to transmission errors [22]. Based on this evaluation, we opted for the Manchester encoding scheme for implementing a PDN covert channel in this chapter. The Manchester code defines a format to physically represent bits on a transmission line. In the Manchester code, logical zeros are encoded in a falling signal edge, whereas logical ones are represented by rising signal edges. The level transition occurs in the middle of the bit-period i.e., it is aligned with the rising clock edge of a shared clock signal. Due to this alignment capability, the Manchester code is a so-called self-clocking code. The clock signal can be extracted from the data signal itself. Thus, depending on the implementation, a shared clock signal or synchronized clocks on the transmitter and receiver side are not required [88].

In this chapter, we have implemented a covert channel using the shared PDN, meaning only a single transmission line for communication is available. Hence, we make use of the self-clocking property of the Manchester code, synchronizing the transmitter and receiver without a shared clock signal.

5.2 Power Covert Channel Implementation

This section describes the implementation details of the PDN covert channel presented in this chapter. After a brief description of the experimental setup, a more detailed explanation of the transmitter and receiver design is presented.

5.2.1 Experimental Setup

The experimental setup used in this chapter is the Pynq-Z1 board from Digilent. This platform contains a Xilinx Zynq-7000 SoC which features a dual-core ARM Cortex-A9 CPU running at 650 MHz together with a Xilinx Artix-7 FPGA clocked at 300 MHz in our experiments. The SoC is connected to an external 512 MB DDR3-RAM chip. We supplied the board with power through micro-USB instead of using an external power supply.

The transmitter software is running on an Ubuntu 18.04 operating system. For an evaluation of the transmission quality, we read the decoded bitstream from Linux and store them inside logfiles. These logfiles are then downloaded for an offline analysis on a standard PC.

5.2.2 Transmitter Design

The transmitter software aims at modulating the usage of the PDN by varying the CPU load. In [22], the open-source application `stress` has been used for generating voltage drops on the shared power supply by imposing load on several CPU cores during a given duration with matrix multiplication operations. We verify that this methodology can also be applied to our platform, however it has the downside of leading to a low transmission rate, which is inherent to the usage of the `stress` tool. The tool indeed needs to be run in seconds granularity which prevents the achievement of a high transmission rate.

Initial experiments revealed that a sequence of division instructions were sufficient for generating a voltage drop which is significant enough for implementing a covert communication. After testing several strategies for encoding a high voltage level, we opted for the usage of the `nanosleep` system call. During the execution of `nanosleep` we observed an initial voltage drop for 15 µs followed by a raised and a constant high level which depends on the given duration and a final voltage drop for another 15 µs (see figure 5.2). Fortunately, this behavior matches the Manchester code specification which requires a level transition after either a half or a full period.

The PDN response to `nanosleep` prevents the transmission of discrete bits by iterating through the bitstream to transmit. Instead of that, we used the translation table (see table 5.1) which takes the current bit value to transmit, its predecessor and successor to encode a sequence of instructions which modulates the PDN according to a Manchester encoding (see section 5.1.2). Additionally, for a bitstream of size n we assume that the predecessor of the first bit is 0 and the successor of the last bit is 1. The sleeping durations of 30 µs and 60 µs, as well as the number of divisions, are derived from the low voltage level between two sequential `nanosleep` syscalls. From figure 5.2 the duration of this voltage level can be read off to be 30 µs. As a result, the period of the Manchester signal must be double this time. Thus, the software must modulate high voltages for 30 µs and 60 µs by executing the `nanosleep` syscall with the respective duration. Low voltage levels of 30 µs and 60 µs are modulated by the delay between two `nanosleep` executions and an extension of this delay by executing 1200 integer divisions.

Figure 5.3 depicts the waveform corresponding to the transmission of the bitstream *011*, which is obtained by executing instructions on one ARM Cortex-A9 core following the encoding contained in table 5.1. In conformity to the Manchester code specification, only transitions occurring at the middle of a bit-period (60 µs) carry information (cf. section 5.1.2). Therefore, the falling edge occurring after 120 µs in figure 5.3 does not encode a bit.

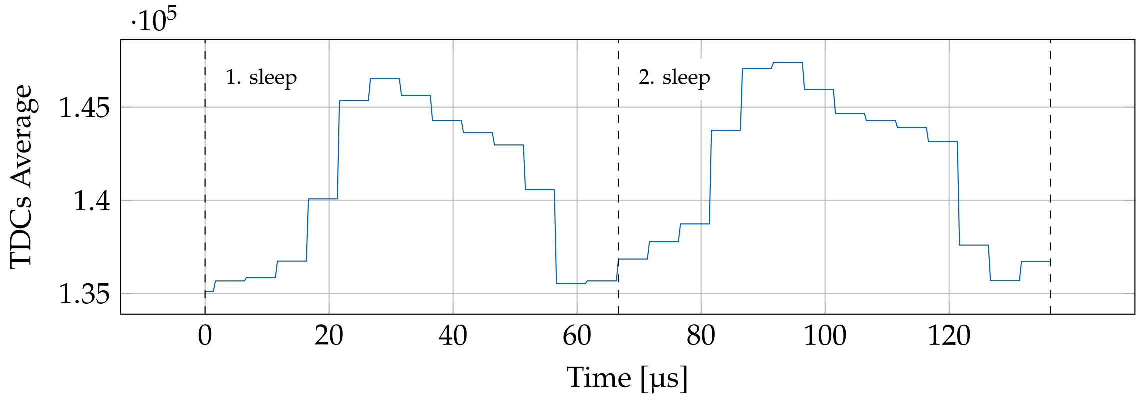


Figure 5.2 Averaged TDCs' measurements during two consecutive `nanosleep` executions using Linux

Predecessor	*Value*	*Successor*	*Instruction*
0	0	don't care	sleep for 30 µs
0	1	0	divide 1200 times, sleep for 60 µs
0	1	1	divide 1200 times, sleep for 30 µs
1	0	don't care	-
1	1	0	sleep for 60 µs
1	1	1	sleep for 30 µs

Table 5.1 Translation of a bit under consideration of its direct neighbors into an instruction list, used for voltage modulation

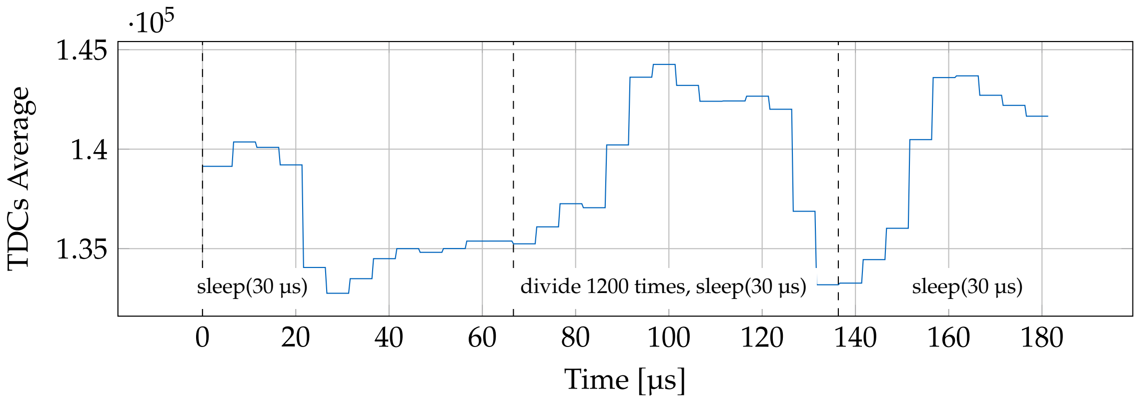


Figure 5.3 Signal waveform resulting from the execution of the instruction list translated from the bitstream *011*

5.2.3 Receiver Design and Message Decoding

This section describes the components involved in the receiver block. The receiver logic running at 300 MHz consists of TDC sensors and a decoding logic which is represented in figure 5.4. A particular focus is made on the Finite-State Machine (FSM), which is used for detecting edges corresponding to a valid bit transmission and the corresponding bit value according to the Manchester code specification (see section 5.1.2).

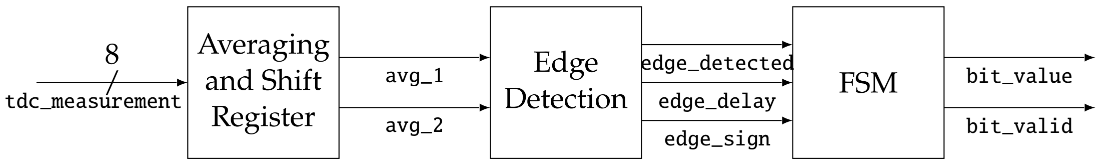


Figure 5.4 Block diagram of the decoder logic with intermediate signal names

5.2.3.1 PDN Monitoring

For monitoring the variations of the PDN, we use TDC sensors. One TDC sensor consists of a chain of 16 CARRY-4 elements as delay-line and 2 LUTs-6 elements as initial delay (see figure 2.7 and section 2.4.4.4 for the explanation of the PDN monitoring via TDC sensors). The output of a TDC sensor is further fed into an encoder so that it can be represented in binary code. Using multiple TDCs and averaging the measurements lead to a better voltage fluctuation coverage and measurement quality. However it also results in additional noise generated by the TDC sensors which in turn decreases the measurement quality. Therefore a trade-off has to be found with the number of TDCs and the resulting measurement quality. Previous work [28] investigating side-channels on a Zynq-7000 Processing System via TDC sensors found that the usage of 8 adjacent TDC sensors placed at the left-hand side of the FPGA produce the best measurement quality. We used this configuration as a baseline and performed further experiments in section 5.3.3, to evaluate the influence of the placement of the sensors on the covert channel quality.

5.2.3.2 Averaging and Shift Register

The first block of the decoding logic (visible in figure 5.4) is a block-averaging mechanism which can be seen as a low pass filter that is applied to the noisy TDCs' measurements. The approach consists in summing 1500 samples from the 8 TDCs. Using this approach rather than a more complex low pass filter is sufficient since the covert channel transmission frequency is much lower than the TDCs' sampling frequency. The averaging block is followed by a shift register of size 2, which keeps the current (`avg_1`) and previous block-averaged values (`avg_2`).

5.2.3.3 Edge Detection

The signal edge detection is implemented as a gradient-based mechanism which compares the difference between two block-averaged values. If the absolute value of this difference is higher than a fixed threshold, the `edge_detected` signal is set high for one clock cycle. Falling edges are encoded as 0 whereas rising edges are encoded as 1 in the `edge_sign` signal. Furthermore, a counter is used to reflect the delay between two consecutive edges. According to the Manchester code specification, a signal can only show a constant level for either a half or a full bit-period. Therefore, this threshold is set to 3/4 of a bit-period. In practice, no clock is shared between the decoder logic and the transmitter code running on the CPU. Moreover, the non-determinism of the Linux operating system might cause slight variations in the bit-period. Consequently, we determined the threshold value empirically. If the delay between two detected edges exceeds this threshold, the `edge_delay` signal is set high for one clock cycle.

5.2.3.4 Finite State Machine

The final step of the decoder logic is an FSM that gets the decomposed signal and returns the decoded bit values. To determine whether an edge in the Manchester code actually encodes a bit, we require knowledge about the previous edges in the signal (cf. section 5.1.2). Hence, we use an FSM since it stores the information about the previous edges in the currently active state.

Figure 5.5 visualizes the Mealy FSM with two input and two output bits. The first input represents the delay passed since the previous edge with zero meaning a half bit-period passed and a one signalizing an entire bit-period passed. The second input bit shows the edge sign where falling edges are represented by a zero and rising edges by a one. The first output bit finally stores the decoded bit value and the second output is set high whenever the decoded bit is valid. The FSM is triggered asynchronously whenever the upstream edge detection logic detects an edge.

The main function of the FSM is to decode the Manchester encoded signal. If an edge is detected and it does not encode a bit, one of the two *pre-** states is entered and the validation output is set low. This prevents subsequent logic blocks from accepting the current decoder output. Contrarily, if an edge is detected that encodes a bit the corresponding *valid-** state is entered and the validation output is set high. The distinction of whether an edge encodes a bit or not is accomplished based on the currently active state. According to the Manchester code definition, edges in the data signal that encodes bit values must be aligned with the rising edge of the shared clock signal. Since our implementation only has a single transmission line and no shared clock signal we extract a virtual clock signal from the data signal. This virtual clock signal is stored in the first input bit which stores the delay passed since the previous edge was detected. If one of the *valid-** states is active it means the last detected edge did encode a bit and was consequently aligned with the rising edge of the virtual clock signal. If the next edge is detected after a half bit-period it will be aligned with the falling edge of the virtual clock signal. Hence, it does not encode a bit and the corresponding *pre-** state is entered. But in case the next edge is detected after an entire bit-period it will be aligned with a virtual rising clock edge again thereby encoding a bit value.

Moreover, the FSM has the function to resynchronize the receiver with the transmitter. If an edge is missed or an additional one is detected the decoder FSM might enter a false state and incorrectly decode subsequent edges. To cope with this issue, we use a property of the Manchester code that unambiguously shows if an edge decodes a bit or not. If a delay of an entire bit-period between two edges is detected, both edges must encode a bit according to the Manchester code. Consequently, whenever an edge after a delay of an entire bit-period is detected the FSM enters the corresponding *valid-** state, regardless of the currently active state.

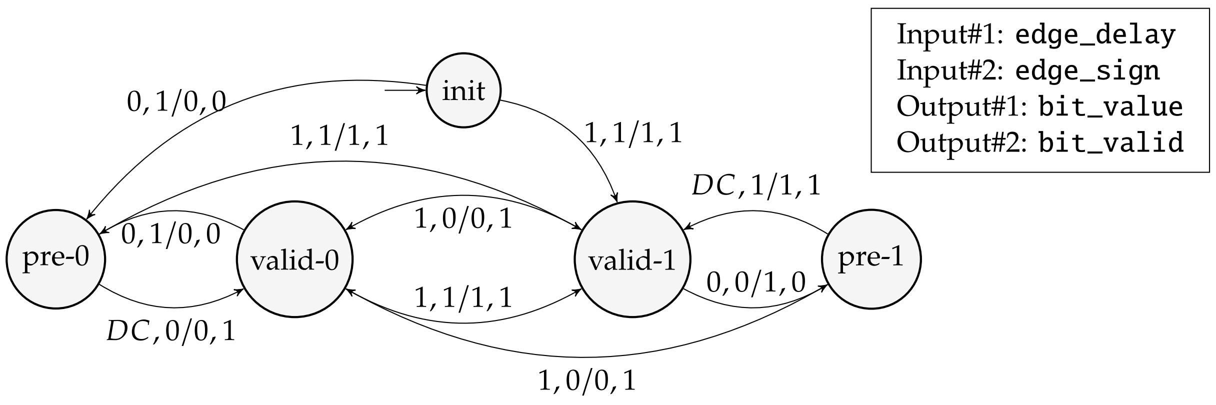


Figure 5.5 FSM to determine which signal edges encode to an actual bit value

5.2.3.5 Decoder Control

The current implementation uses control signals for configuring the decoder and starting and stopping the decoder logic. The configuration of the decoder consists in specifying the block size used for averaging, the thresholds for the detection of edges, and the delay between two consecutive edges. All these parameters are configurable via software by writing into specific AXI-addressable registers. This enables flexibility in the decoder configuration without the necessity of regenerating a bitstream for a different decoder configuration.

In addition, the start and stop signals used for a transmission are also encoded into AXI-addressable registers. These registers are system-wide accessible, which violates the concept of covert channel presented in section 5.1.1. In future work, the start and stop signals used for the covert communication should be encoded in the PDN via a specific start and stop bit pattern.

5.3 Power Covert Channel Characterization

In this section, the performance and transmission quality of the covert channel are evaluated. Furthermore, we analyze the influence of the sensors placement on the covert channel characteristics and present the FPGA resource usage of the sensors and decoder logic.

5.3.1 Data Rate Limitations

The communication performance is represented by the achieved transmission rate. The bit-period of the covert communication is derived from the transmitter implementation. As shown in section 5.2.2 and table 5.1 the minimal duration of a constant signal level is 30 µs. Using the Manchester code results in a bit-period of 60 µs since every bit is transmitted as a combination of a high and low signal level of equal duration. Consequently, this results in a maximal data rate of 16.7 kbit/s.

The lower bound of 30 µs for a constant signal level is due to the presented behavior of the `nanosleep` syscall (cf. figure 5.2). It is important to note that this bound does not correspond to the physical limits of the used device. Implementing the transmission software as a bare-metal program instead of a Linux application results in an increased transmission rate of 47.1 kbit/s.

5.3.2 Transmission Quality

To determine the quality of the communication channel we examine the ratio of falsely detected bits i.e., bit error. This allows a fine granular analysis of the conditions under which errors are especially likely to occur. Additionally, we determine the word success rate, meaning the ratio of correctly transmitted words. Both metrics are calculated from a set of 10 000 word transmissions.

5.3.2.1 Bit Error vs. Word Size

Figure 5.6 shows the bit error for the four different word sizes from 8 to 64 bit. As expected, the smallest evaluated word size of 8 bit results in the lowest ratio of falsely transmitted bits of 2.3 %. Furthermore, figure 5.6 shows an almost linear increase of the bit error with increasing word sizes. This matches the results of Gnad et al. [27]. Their FPGA-to-FPGA covert channel also shows a nearly linear dependency between the bit error and the width of the transmitted word.

5.3.2.2 Bit Error Distribution

As a metric to evaluate the communication quality, we measure the bit errors occurring in a set of 10 000 transmissions. For that purpose, we transmit 64-bit wide words in sequence

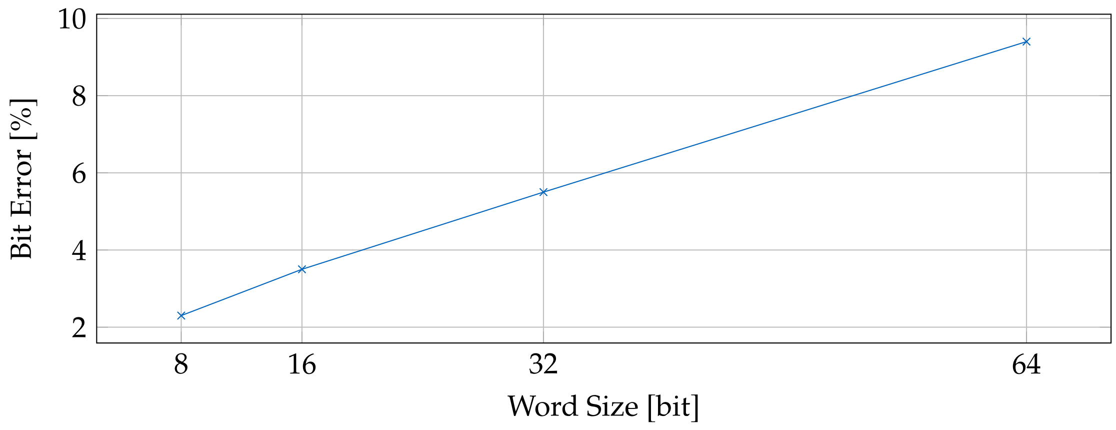


Figure 5.6 Relative bit error in percent against different word sizes, calculated form a set of 10000 samples

and calculate the relative amount of falsely detected bits for every index of the words.

Figure 5.7 presents the bit error evaluation results. The bar plot depicted in blue shows the bit error distribution for a set of randomly generated words. Since every single word in this set is independent of every other word, it is possible to derive the general dependence of the bit error on the index and word size. The bit error distribution measured

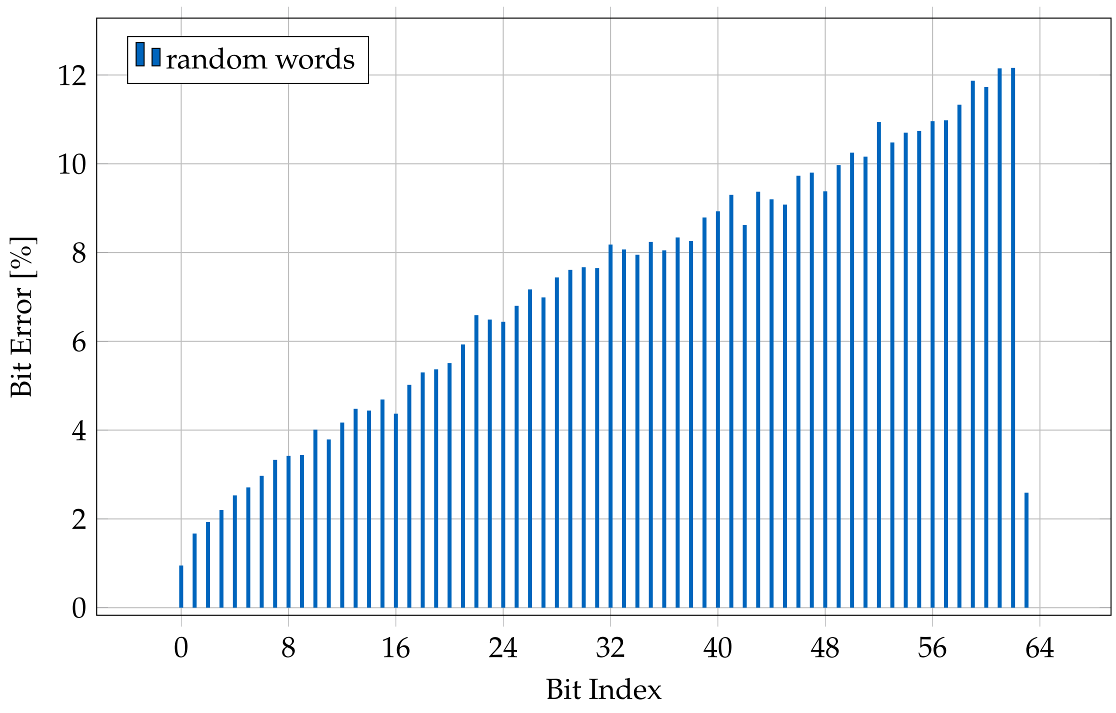


Figure 5.7 Relative bit error in percent against the position of the respective bit in a 64-bit wide word

after transmitting random words in figure 5.7 shows a linear increase with rising index. This effect can be explained by the occurrence of synchronization errors. The PDN covert channel uses only a single transmission line and does not have a shared clock signal. The Manchester code in combination with the implemented decoder FSM enables the communication without synchronized clocks on the transmitter and receiver sides. However, this approach is prone to detecting unintentional edges and missing intentional edges in the data signal. If such a detection error occurs, the FSM might enter a false state and is not able to decode the following edges correctly. Furthermore, even after a resynchronization of the FSM, the signal pattern is decoded correctly but the data might be aligned at an incorrect index. Consequently, bit errors at one specific position in the data word induce further errors at the subsequent indices. Summing the individual bit error at an index and the bit errors at upstream indices results in the linear increase shown in figure 5.7.

5.3.2.3 Word Success Rate

A further metric that represents the quality of the communication channel besides the bit error is the word success rate. This characteristic describes the relative number of successfully decoded words in comparison to the total number of transmitted words.

Figure 5.8 shows the word success rate against different word sizes. Transmitting byte-sized words results in the highest success rate of 94.5 %. It is visible that the success rate decreases almost linearly with increasing word sizes. The moderate gradient is surprising since two known effects contribute to a decrease in the word success rate. First, with a longer word size the probability that at least one bit error occurs increases. Secondly, the bit error distribution shows higher numbers of errors in wider words suggesting a decrease in signal quality.

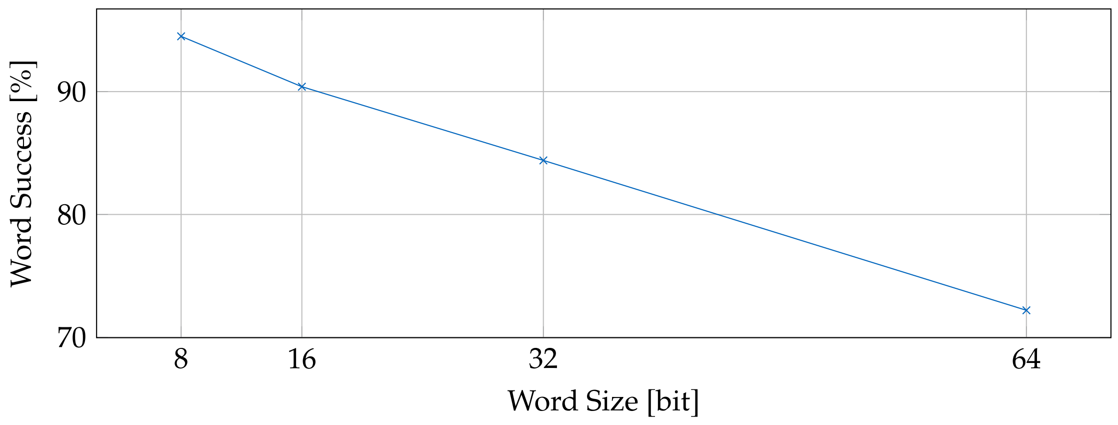
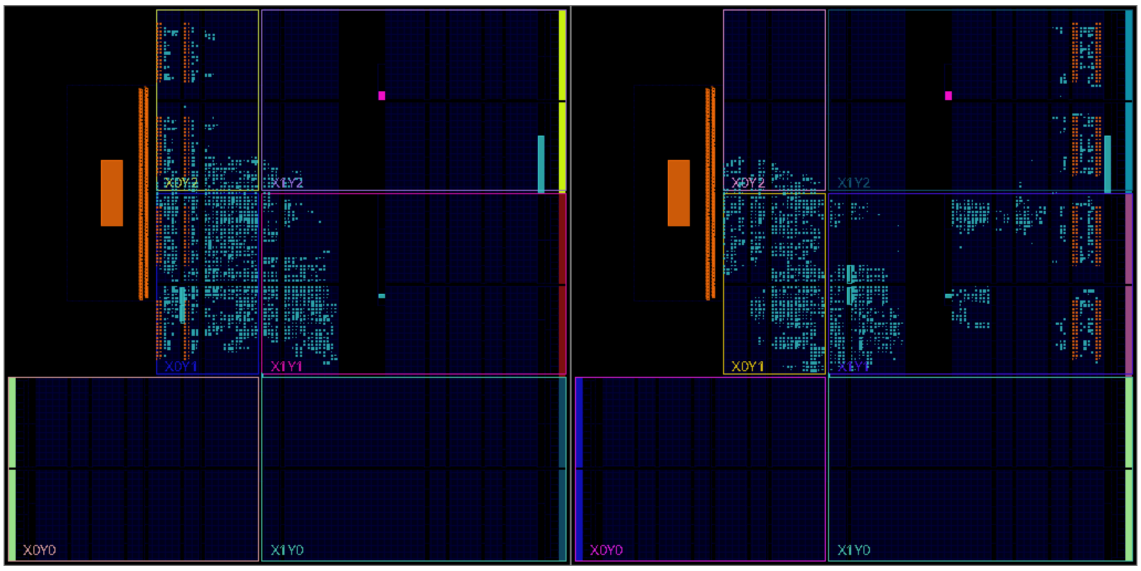


Figure 5.8 Word success rate in percent against different word sizes, calculated form a set of 10000 samples

5.3.3 Influence of the Sensors Placement

To determine whether the physical location of the TDC sensors inside the FPGA influences the communication quality, we chose two distinct placements on opposite sides of the FPGA (see figure 5.9). In both positions, all 8 TDCs are placed near each other. For the first setup, the slices directly next to the CPU are used to implement the TDCs. This leads to a placement of the sensors on the left-hand side of the FPGA fabric. For the second location, the sensors are placed on the far-most right-hand side of the FPGA. The two positions are 80 slices apart horizontally.



(a) **(b)**

Figure 5.9 Placement of the eight TDCs a) next to the CPU and b) far away from the CPU

Figure 5.10 shows the comparison of these two placements in terms of communication quality. Figure 5.10a depicts the signal waveforms measured by the TDCs. The upper blue curve corresponds to the TDCs' placement next to the CPU. The lower orange curve shows the waveform measured with the sensors placed on the opposite side of the FPGA. Comparing both measurements shows that the orange waveform is scaled down by 25% in comparison to the blue waveform. Consequently, the signal edge height stays constant relative to the signal level. This shows that in both positions the TDCs measure a significant voltage variation when the PDN is modulated by the CPU. The downscaled voltage measurements can be explained by a non-uniform PDN. Hence, the supply voltage varies slightly due to the design of the PDN across the SoC. Comparable results are presented in the work of Krautter et al. [55] who exhaustively analyzed the influence of the transmitter and receiver placement on the quality of an intra-chip side-channel attack. They found that the power distribution is not uniform within the chip, which can result in a different transmission quality, that is not necessarily influenced by the physical distance to the transmitter.

Since we use a gradient-based approach to detect edges in the data signal and the downscaling of the TDCs' measurements results in a reduced gradient value, a negative influence on the communication quality is expected. A comparison between the word success rate, i.e. the relative number of correctly transmitted data words of the two different TDC placements is shown in figure 5.10b. It shows that the downscaling of the edge height due to the different sensor positions does not present a problem for small word sizes. In contrast, transmitting wider words result in a significantly decreased word success rate. Here, the high number of bits and therefore the increased error probability in combination with a more susceptible signal-to-noise ratio result in a lower communication quality.

5.3.4 Resource Utilization

As shown in table 5.2, the receiver and decoder logic can be implemented with a small resource usage, using only 2.35% of the available FPGA LUTs.

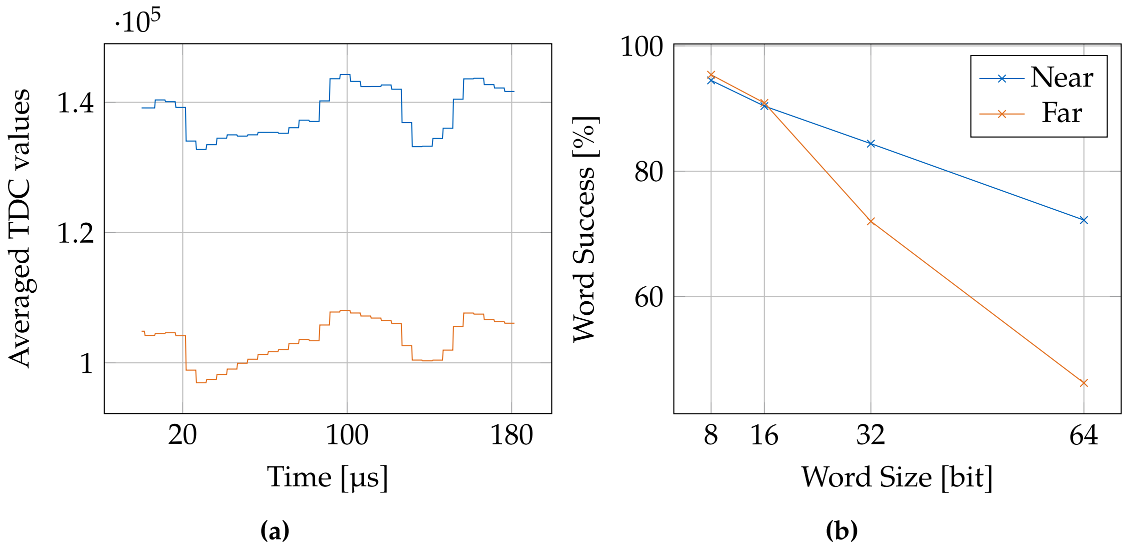


Figure 5.10 Comparison of a) the signal waveforms and b) word success rate between the placement of the TDCs next to the CPU (blue) and far away (orange)

On the CPU side, the transmitter is implemented with only one thread performing a sequence of divisions and `nanosleep` system calls. Moreover, the capability of achieving a good transmission at a high bandwidth without having to resend a message multiple times contributes to make the transmitter code stealthy on the CPU usage.

Type	*Amount*	*Utilization*
Slices	555	4.17 %
LUTs	857	2.35 %
Registers	1364	1.28 %

Table 5.2 Resource utilization caused by the receiver and decoder logic

5.4 Discussion

The following section first presents a comparison between the CPU to FPGA covert channel implemented in this chapter and other state-of-the-art covert channels shown on FPGAs. In a second time, the usage of the implemented covert channel as an activation function for a HT is discussed. Finally some considerations regarding noise and potential countermeasures are presented.

5.4.1 Comparison with Other Power Covert Channels Involving FPGAs

Temperature [39, 92] and power consumption [27] are the most promising transmission medium that can be used as covert channel on FPGA platforms since they can bypass FPGA isolation mechanisms. The temperature covert channel uses high and low temperature level to encode bits and has been shown to be practicable on standard FPGA platforms [39] upto FPGA platforms integrated in the cloud [92]. Its downside is mainly the achieved transmission speed, with which several minutes are required to transmit a

128 bit AES key [92]. A faster transmission medium relying on the FPGA power consumption is presented in the work of Gnad et al. [27]. Their implementation uses ROs in a custom logic circuit to modulate the supply voltage. A TDC sensor, programmed into the same FPGA fabric, observes the PDN. Since they also use the Pynq-Z1 as their evaluation platform, we can use their results to classify our covert channel. In comparison to our implementation, their covert channel achieves data rates up to 8 Mbit/s. They use the same transmission medium on the same hardware and an equivalent receiver circuit. Therefore, we can derive that the transmitter software executed on the CPU is the main performance limitation. This supports the observation presented in section 5.3.1, showing an increased transmission rate when executing the software as a bare-metal program instead of a Linux application. Moreover, Gnad et al. are able to generate three distinct voltage levels whereas our covert channel uses rising and falling level edges between two voltage levels. Consequently, using a custom logic circuit to stress the PDN results in fine granular control over the PDN in terms of timing and level modulation.

A different approach towards power covert channels was taken by Giechaskiel et al. [22]. Instead of implementing both the transmitter and receiver on the same chip and using the shared PDN as the transmission medium, they have used a computer Power Supply Unit (PSU). The transmitter and receiver are implemented using discrete devices which are either placed directly on the motherboard or connected via Peripheral Component Interconnected Express (PCIe) acceleration cards. This setup results in a more complex transmitter and receiver design. While we are able to modulate the supply voltage using only a single core of an embedded CPU, they have required multiple threads of a desktop-class CPU. Moreover, the receiver that is implemented on a discrete FPGA requires additional circuitry to measure deliberate voltage variations. They have used additional ROs, stressing the voltage regulators to make the supply voltage more vulnerable to high CPU loads. A receiver of this complexity is not required for our covert channel. As shown in section 5.2.3, voltage variations are directly measurable using TDC sensors with a simple block averaging scheme to filter high-frequency noise.

In conclusion, the threat of a power covert channel heavily depends on the transmitter implementation and the nature of the transmission medium. The shared PDN of a single chip is especially vulnerable to covert channel communication. In comparison to a common PSU, exploiting the shared PDN by implementing the transmitter and receiver on the same chip shows significant improvements in the achievable transmission rate. Furthermore, it allows a simplified transmitter and receiver design.

5.4.2 Activation of a Hardware Trojan via the Covert Channel

HTs consist of malicious circuits hidden within a benign design. In the context of FPGA-SoCs, they can typically be contained in IP cores obtained from third parties. The insertion of HTs in FPGA-SoCs is also facilitated due to the relaxed trust assumptions on the FPGA reconfiguration interfaces made on Xilinx FPGA-SoCs [101]. Xilinx considers the PCAP and the ICAP as trusted in the context of secure boot (see section 2.4.4.1). Therefore, it is possible to load un-authenticated and un-encrypted bitstram after a secure boot process due to these relaxed trust assumptions [101].

After its insertion, a HT should remain discrete and only activated under specific conditions, which are not reproducible during normal operation. The activation signal should also be communicated via an indirect communication channel.

In this chapter, we have the capability to encode a chosen bitstream in the PDN via the bit to instruction mapping presented in section 5.2.2. To ensure that the transmitted bitstream cannot be reproduced during normal conditions and still be transmitted reliably, the activation bitstream should be large enough, transmittable in a short duration, and have a good word success rate. With the analysis presented in sections 5.3.2.1 and 5.3.2.3, we think that a trigger signal of 16 bits can be a good trade-off for ensuring those requirements. In section 5.4.3, the influence of noise which may degrade the activation signal transmission quality is discussed together with techniques that can be investigated in future work for improving the transmission quality in presence of noise.

5.4.3 Influence of Noise and Countermeasures

Besides the analysis in terms of performance and quality, further topics are still worth investigating. Currently, the used FPGA-SoC is operated in an ideal state for the covert channel implementation. Gnad et al. [27] showed that a PDN covert channel can be implemented with noise sources located within the FPGA. In future work, we should verify if noise sources generated by logic inside the FPGA can disturb the reception of the message encoded in the PDN by the CPU.

In addition to noise on the receiver side, noise on the transmitter software should also be considered. In the current evaluation, no major application is running on the Linux operating system during the experiments. One transmitter thread is executed using only one of the two available cores of the integrated ARM Cortex-A9 and is not running in parallel or being preempted by another thread. A promising strategy to deal with thread preemption could be the multi-threading of the transmitter signal, as presented in [22]. While the results presented in [22] haven't explicitly considered preemption of the transmitter code, they showed an increase in the transmission quality by considering multiple threads running *stress*, which is the PDN stressor the authors used for encoding bits in their covert channel implementation. Future work should investigate if running multiple transmission threads increase the transmission quality at the price of a decrease in the implementation stealthiness and if it enables the run of parallel victim workload during the covert transmission.

The implementation of countermeasures in the context of a PDN covert channel remains challenging. In contrast to side-channel attacks, where the implementation to be protected is clearly defined and can be masked with adequate techniques [79, 29], equalizing a hidden PDN transmitter on a CPU or within the FPGA logic is more difficult. One approach could be to use a bitstream analyzer scheme which prevents the insertion of power sensors within the FPGA logic [57]. However, it was shown that power sensing circuits that are harder to detect can still be used despite this approach [89].

From a system level point-of-view, another countermeasure which can partially the threat of HT insertion and limit the risks of covert channels between a CPU and an FPGA consists in restricting the trust assumptions of the FPGA reconfiguration interfaces (see section 2.4.4.1). Doing this consists in restricting the reconfiguration of the FPGA with authenticated bitstreams only. While this doesn't mitigate the electrical threat inherent to the covert channel, it hinders the insertion of the receiver logic which is necessary for the implementation of the covert channel. The implementation of this countermeasure is discussed in the chapter 6 of this thesis.

5.5 Summary

In this chapter, we have shown that the PDN is a vulnerable resource that can be used for implementing a covert channel between a CPU and an FPGA on an FPGA-SoC platform. This communication channel is particularly interesting for the activation of HTs in FPGA-SoCs. By using simple sleep system calls and integer divisions, we were able to modulate the PDN usage in a stealthy way. The hidden message encoded in the PDN can then be decoded within the FPGA using TDC sensors and a decoder logic. Overall, the presented covert channel achieves a transmission rate of up to 16.7 kbit/s and a bit error rate of 2.3%, which is a significant improvement in comparison to other CPU to FPGA covert channels. Future research should evaluate the robustness of the covert channel to noise sources running in parallel to the transmitter software and investigate possible countermeasures.

6 Enhancing the Security of FPGA-SoCs via the Usage of a Trusted Execution Environment and a Hybrid-TPM

In this chapter, we investigate the security benefits obtained from the integration of a TEE and an fTPM on FPGA-SoC platforms. To this end, we improve an existing fTPM implementation from Microsoft and adapt it for the ZU+. This consists in adding an entropy source and cryptographic accelerators of the ZU+ to the reference implementation, thereby making the pure software TPM a hybrid-TPM, which relies on a software TPM implementation enhanced with hardware security features. As an application usecase, we demonstrate how ARM TrustZone combined with the hybrid-TPM presented in this chapter help in protecting the FPGA reconfiguration interface via a secure on the fly bitstream loading framework integrated within ARM TrustZone. By using such an FPGA reconfiguration framework, the usage of authenticated bitstream load can be forced on the ZU+. This protection technique makes the insertion of HTs in the FPGA logic harder and thereby partially mitigate the attack vectors presented in the chapters 3, 4, and 5 of this thesis.

The remainder of this chapter is organized as follows: section 6.1 presents the threat model considered in this chapter. Section 6.2 explains the methodology used for deriving entropy out of on-chip SRAM contained inside the ZU+ FPGA-SoC. Section 6.3 describes the changes we made to fTPM's implementation for transforming it into a hybrid-TPM. Section 6.4 introduces a framework that enables a secure usage of the bitstream remote programming interface with the help of ARM TrustZone and our hybrid-TPM. Section 6.5 contains the performance evaluation of this work. Section 6.6 discusses the advantages and limitations of our hybrid-TPM in comparison to dTPMs. Section 6.7 contains a summary of the results presented in this chapter.

The results presented in this chapter are part of the publication: *Enhancing the Security of FPGA-SoCs via the Usage of ARM TrustZone and a Hybrid-TPM in the ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol 15, November 2021 [32].*

6.1 Threat Model and Limitations of a Software TPM Running on an FPGA-SoC

The definition of the threat model aims in identifying the list of attacks that affect our hybrid-TPM and those against which it is protected. To delimit the threat model, we first present the attacker model considered in the original fTPM paper [84]. In a second part we present additional attacks which should be considered due to the introduction of an FPGA inside the original system. Finally we list the security mechanisms of the ZU+ which help in defending against the previously identified attacks and outline those, which are out of the scope of this work.

A summary of the relevant threats for our hybrid-TPM is detailed in table 6.1. In comparison to the relevant threats identified for FPGA-SoCs in table 2.1, our hybrid-TPM offers further protection against bitstream readback/manipulation attacks, SCA, and downgrade attacks. Furthermore it provides hardware entropy for the initialization of a Pseudo-Random Number Generator (PRNG) as well as protected execution for security critical software executed in a TEE.

6.1.1 Attacks Covered by ARM TrustZone and fTPM Original Implementation

The main security objective of a TEE is to provide isolation of a small subset of the software stack. This is achieved by partitioning memory and peripherals between two execution worlds and preventing the NoW execution environment to access the SeW assets (see section 2.2.3.2). This partitioning ensures that even if an attacker can compromise an OS, the code subset contained inside a TEE will still operate securely inside the SeW. Similarly it can protect the execution environment by preventing peripherals such as the DMA from accessing the SeW. fTPM is designed as a trusted application running inside an ARM TrustZone TEE and as such it relies on a similar threat model. In comparison to a standard TEE, it offers additional protection against timing and cache SCA by using constant time cryptographic implementations and by performing cache flush operations on a security world switch. However, Denial-of-Service (DoS) attacks mounted by a malicious OS and physical attacks are considered to be out of scope [84]. The authors also explicitly requires the provisioning of fTPM with a good quality entropy source. Without such an entropy source, the fTPM implementation is vulnerable to attacks exploiting random number predictability.

6.1.2 Attack Vectors due to the Introduction of an FPGA Inside the fTPM Environment

Recent works have shown that a HT can mount powerful attacks on the processing system via the manipulation of DDR memory which is possible through the usage of unprotected memory interfaces [46, 31]. Fortunately, these attacks are more difficult to perform on the ZU+, since the FPGA-SoC can take advantage of memory isolation mechanisms. However, as explained in chapter 3, memory manipulation attacks via a HT are still possible on this platform, if the ACP is not disabled. HTs have also been used to mount remote power SCA [111, 28] or remote fault attacks [25, 56] on FPGA-SoCs. In power SCA, an attacker tries to exploit the leakage of cryptographic algorithms through the power consumption with the goal of obtaining information about a cryptographic key. In remote fault attacks, an attacker tries to compromise the execution flow of software by injecting faults through voltage or temperature variations. These attacks were carried out by implementing a power measurement circuit or a fault triggering circuit inside the FPGA. In this work we offer a partial protection against remote power SCA by taking advantage of the SCA resistance of the cryptographic hardware accelerators available on the ZU+.

Finally another kind of attacks on FPGAs considered inside this work are related to bitstream security. With the framework described in section 6.4, we protect the FPGA configuration from being readback for IP theft purposes. Furthermore we prevent an attacker from programming a bitstream on a compromised device via the requirement of loading an authenticated bitstream and the sealing feature of our hybrid-TPM.

6.1.3 Attacks Covered with the Hybrid-TPM and the Zynq UltraScale+ Security Mechanisms

The integration of some of the ZU+ security mechanisms inside our hybrid-TPM contributes in relaxing the threat model considered in the original paper (see section 6.1.1). The secure boot mechanism available on the ZU+ consists of an authentication of the boot image via the help of RSA signatures. Before a signature verification, the authenticity and integrity of the RSA public key contained inside the boot image is verified against a SHA-3-384 digest of that key, which is stored inside the device's eFuses. Authenticating the software and the bitstream of our hybrid-TPM as part of the boot image guarantees its integrity and authenticity before its execution. In the event of a security vulnerability discovered within a boot image, the ZU+ enables public key revocation via the use of eFuses. This contributes in securing our hybrid-TPM against downgrade attacks, where an attacker tries to exploit vulnerabilities contained in an outdated software.

The ZU+ platform contains cryptographic hardware accelerators for AES-GCM-256, RSA-2048/4096, and SHA-3-384. By using these accelerators together with a secure key storage in BBRAM or eFuses, memory access attacks such as cold-boot attacks can be mitigated [36]. In those attacks, an attacker exploits the possibility of reading cryptographic keys from DRAM by cooling down the memory with a cooling spray such that the data remanence effect lasts longer. An attacker must in addition rely on a custom bootloader for reading-back the memory values. The installation of such a bootloader is prevented by secure boot. The cryptographic hardware accelerators also help in defending against SCA. The AES engine is equipped with a rekeying defense mechanism which forces the usage of a new cryptographic key after a certain amount of encryptions. This mechanism increases the difficulty of SCA, however a recent work succeeded in exploiting leakage of the AES engine and managed to obtain partial information about the cryptographic key via an advanced SCA [37].

6.1.4 Further Attacks not Covered

Fault attacks that are performed by a physical attacker or done remotely via malicious FPGA logic (see chapter 4) or via DVFS are considered to be out of the scope of this work. Although the ZU+ contains mechanisms to detect temperature and voltage variations via on-chip sensors, their limited sampling rate prevent them from detecting a DoS or fault attack fast enough. We also exclude the Rowhammer attack [51] from the scope of this work.

6.2 Building a Truly Random Seed from on-chip SRAM

The National Institute of Standards and Technology (NIST) recommends using a truly random number for seeding a DRBG [9]. By doing this with a well designed DRBG, the output bits produced are unpredictable for an attacker. The derivation of a device fingerpint and randomness from on-chip SRAM was first studied by Holcomb et al. [38] in 2009. Their work demonstrated that SRAM start-up patterns are a suitable primitive to build a fingerprint of a device via a PUF. In addition, they also showed that SRAM start-up patterns are noisy, and that it is possible to derive a truly random seed by processing this noise via a conditioning algorithm such as a hash function. Wild et al. [97] investigated the possibility of building an SRAM PUF on a Zynq-7020 via Block RAM

Physical	Software attacks	FPGA
Cold-boot ✓	Memory corruption attacks ★	DMAA/cache SCA through malicious logic ✓ ✚
Power SCA ★	SCA (cache, timing) ✓ ✚	FPGA bitstream readback/modification ✓ ✚
Fault attacks ★	Fault attacks via DVFS ✗	On-chip power SCA through malicious logic★
DMAA ✓	Rowhammer ✗	On-chip fault/DoS attacks through malicious logic ★
	FPGA bitstream readback/modification ✓ ✚	
	Downgrade attack ✓ ✚	
	DoS ✗	
	Attack exploiting random number generator predictability ✓ ✚	
	Integrity and authenticity of execution through TEE and TPM ✓ ✚	

✓ = covered; ★ = partially covered; ✗ = not covered
✚ = security gain obtained through the TEE/hybrid-TPM;

Table 6.1 Relevant attack vectors affecting the hybrid-TPM

(BRAM) start-up patterns. Their work showed that this approach is not straightforward, since BRAMs contained in Xilinx FPGAs are automatically cleared on power-up, making the access to the start-up pattern not possible.

For the hybrid-TPM implemented in this chapter, we exploit start-up values from on-chip SRAM available on the ZU+. We also found that accessing these values was not straightforward and use a special technique described in this section to achieve that. Although these FPGA-SoCs contain a built-in PUF, the noise contained in the PUF response cannot be exploited for obtaining hardware entropy, because this information is not accessible to the user. Instead of using the PUF noise as entropy source, our methodology consists in exploiting the entropy contained inside the start-up pattern noise of one of the SRAM banks from the RPU Tightly Coupled Memory (TCM). In section 6.2.2 we evaluate the seed quality on four ZU+ devices.

6.2.1 Methodology

In our experiments, we found out that reading the TCM start-up values was not straightforward, because of memory initialization occurring during boot. Therefore, we developed a methodology to power-up and power-down a memory bank on demand via the help of the PMU. When using this technique, the memory bank is not reinitialized by software and thus it is possible to read back the start-up values of the TCM.

The methodology used for deriving a truly random seed out of the R5_1_BTCM bank (which we will refer to as B4) of the TCM is depicted in figure 6.1.

By interacting with some global registers of the PMU (REQ_PWRDWN_INT_{EN, DIS, TRIG, STATUS} and REQ_PWRUP_INT_{EN, DIS, TRIG, STATUS}), the APU can power-

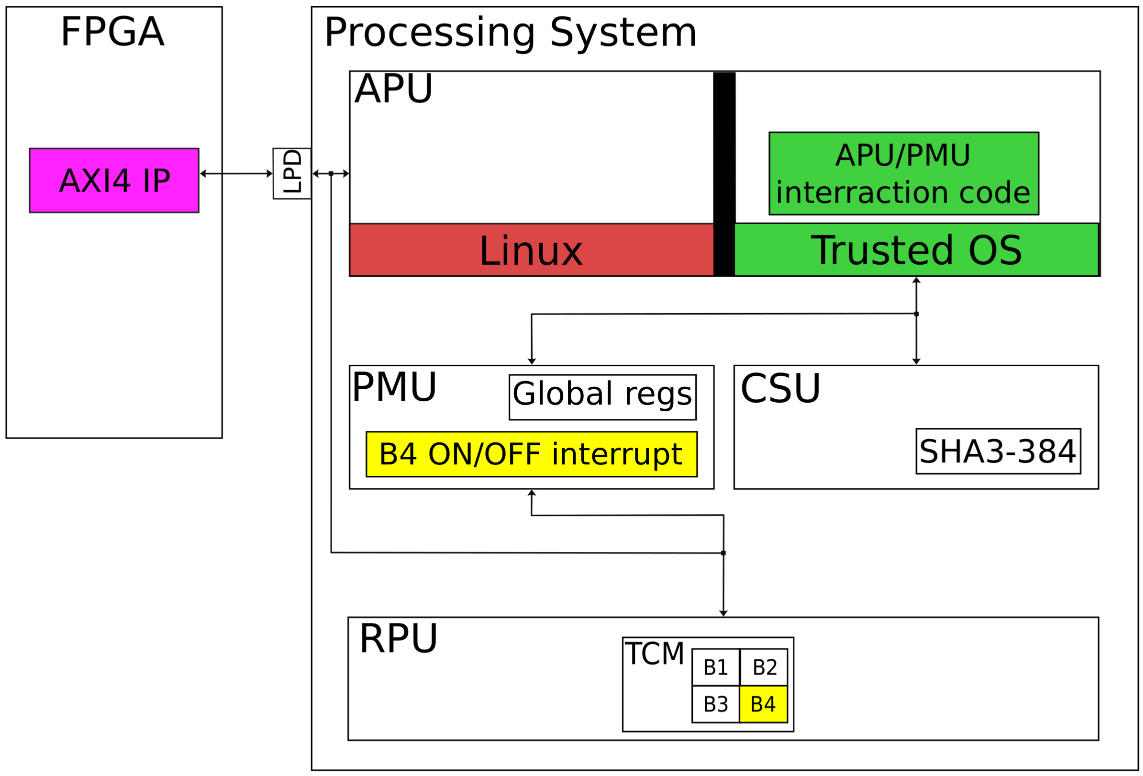


Figure 6.1 Methodology used for deriving a truly random seed

up and power-down the TCM bank B4 via the *B4 ON/OFF interrupt* occurring inside the PMU. The access to the PMU global registers is restricted to the SeW, therefore the APU code used for interacting with the PMU is running inside a trusted OS.

Once B4 has been restarted, its start-up pattern must be read. We found out that no explicit initialization of the bank is performed when powering it ON and OFF via the *B4 ON/OFF interrupt*. Therefore it is possible to exploit the randomness contained in B4's start-up pattern by reading back the start-up values. However, this is not directly possible from the APU or the RPU due to a memory access error which is propagated on the AXI bus while doing this. To overcome this issue, B4 is read back from the FPGA logic via the Low Power Domain (LPD) interface with a custom AXI4 IP. This module ignores the errors encountered while reading back B4 and write the start-up pattern in a reserved area of the SeW DDR memory (possible because the data is visible on the bus even in the case of an AXI transaction error). The last step consists in obtaining a SHA-3-384 digest of B4's start-up pattern. This is necessary in order to extract entropy out of the SRAM start-up pattern. The hashing is performed by the SHA-3-384 hardware module contained inside the CSU.

In this work we didn't consider the case of a real-time application or real-time operating-system running on the RPU. If such a scenario is required, the linker script of the application and the kernel of the Real Time Operating System (RTOS) should be adapted such that B4 is not used. Preventing the usage of a memory bank may lead to a performance penalty which is compensated by an increase in security. One advantage of our methodology in comparison to related work is that the process described in this section can be used multiple times without having to restart the system. In section 6.2.2, we perform an evaluation of the seed randomness quality across four ZU+ devices. This study gives some insights concerning the quality of the generated entropy and determines the number of TCM start-up bits required for deriving a 384 bit true random seed from B4's start-up pattern.

6.2.2 Evaluation of the Seed Randomness Quality

The seed quality is evaluated by computing the fractional hamming distance and the min-entropy. These two metrics are commonly used in works exploiting SRAM properties for the design of TRNGs and PUFs [38, 59]. The first metric gives an insight regarding the variability of the start-up patterns on a same device. The second enables us to determine the number of SRAM bits required for the derivation of a truly random seed.

6.2.2.1 Fractional hamming distance:

As described in the work of [38], SRAM start-up patterns contain a stable part that can be used for device identification and a part which is noisy. In this work, we exploit the noisy part to construct a truly random seed after processing it (see section 6.2.1). Our methodology induces a sequence of power-up and power-down operations on an SRAM bank. This sequence of operation is susceptible to reduce the noise of SRAM start-up patterns due to the data remanence effect affecting SRAM during the power off period. To estimate the noise contained in our measurements, we measure the fractional hamming distance among the collected start-up patterns. This metric quantifies the differences in the bit sequence of SRAM start-up values. In the PUF context, this would be denoted as the reliability metric [70]. We used 100 start-up patterns for four different devices compliant to the ZU+ architecture: two ZCU102 Evaluation Kits Revision 1.1 (devices 1 and 2), one ZCU104 Evaluation Kit Revision 1.0 (device 3), and one Ultra96-V2 development board (device 4). The 32 first kB of B4 were used for the evaluation. All the measurements were collected at ambient temperature.

Devices	Average fractional hamming distance
Device 1	9.80%
Device 2	10.67%
Device 3	10.08%
Device 4	10.70%

Table 6.2 Average fractional hamming distance between 100 R5_1_BTCM start-up patterns

The results from table 6.2 demonstrate a sufficient variable part among the different start-up patterns. These results also suggest that the methodology described in section 6.2.1 is resilient enough to the data remanence effect. In comparison to related work, our results suggest that the TCM start-up pattern is more noisy than the SRAM chips which were considered in [86]. This would be a problem for designing a PUF out of the TCM. For the design of a TRNG out of the noisy part, a higher noise proportion in the SRAM start-up patterns is a desirable feature.

6.2.2.2 Min-entropy:

To estimate the quality of the randomness generated via the technique presented in section 6.2.1, it is necessary to determine the entropy in the worst case scenario, which is referred by NIST as the min-entropy. NIST defines the min-entropy of a random variable X as the greatest lower bound for the information content of potential observations of X.

Previous work have proposed different methodologies for obtaining this metric from the SRAM start-up pattern noise depending on a bit-wise [59] or byte-wise [38] independence hypothesis. Similar to the work of [59], we assume that all bits of the SRAM start-up pattern are independent from each other. Therefore, the min-entropy computation of an SRAM start-up pattern of length N is obtained by calculating the maximum bit value probability (p_i^{max}) for each of the bits individually and combining those probabilities according to equation 6.1:

$$min\_entropy = \sum_{i=1}^{N} -log_2(p_i^{max}) \tag{6.1}$$

The evaluation of the min-entropy is done on 100 start-up patterns on the four different devices introduced earlier in this section. The entropy is evaluated before the computation of a SHA-3-384 digest from B4's start-up pattern. The min-entropy results are grouped in table 6.3.

Devices	min-entropy
Device 1	12.04%
Device 2	13.08%
Device 3	11.86%
Device 4	12.65%

Table 6.3 Min-entropy evaluation of the R5_1_BTCM start-up pattern

The results listed in tables 6.2 and 6.3 give a first insight regarding the applicability of our methodology for deriving a truly random seed out of the R5_1_BTCM bank start-up pattern. In comparison to the works of [59, 38], our results suggest that the TCM presents a slightly higher min-entropy value. To be more complete, a future work should extend this study by using more devices and varying environmental conditions and chip voltages as done in [59, 38].

For the rest of this chapter, we consider a min-entropy rate of 8%. This should be lower than the actual min-entropy rate computed on a ZU+ device according to our first measurements and thus it gives a safety margin which is necessary given the incompleteness of our statistical study. With this value, the left-over hash lemma [40] requires 4800 bits of SRAM start-up pattern to produce 384 random bits via a SHA-3-384 digest. We used this configuration in the rest of this chapter.

6.3 Enhancing Microsoft's fTPM Implementation towards a Hybrid-TPM for the Zynq UltraScale+ Architecture

Microsoft implemented fTPM [84] purely in software so that it can be deployed easily on many mobile devices. However, the reference implementation [73] can be adapted for a particular hardware platform, which makes the integration of device specific features inside fTPM possible. By doing this, it is possible to enhance fTPM implementation's with platform specific features such as cryptographic hardware accelerators. The original work [84] also explicitly requires the use of an entropy source accessible to the secure world and relies on the platform manufacturer for providing it.

In this section, we describe the modifications made to the original fTPM implementation which transforms the original software implementation into a hybrid hardware/software implementation for the ZU+. This is achieved by enhancing fTPM with hardware cryptographic support and by providing hardware entropy to fTPM via the technique described in section 6.2.

6.3.1 System Description

This chapter uses a Xilinx ZU+ ZCU102 Evaluation Kit. The software components running on the Cortex-A53 processor are compliant with the fTPM software architecture represented in figure 2.5. Petalinux 2018.3 is running as the NoW Rich OS together with the version 2.10 of the TPM 2.0 software stack. The fTPM-TA is running inside the TEE OP-TEE. We are using an enhanced version of OP-TEE 3.4.0 which integrates hardware security features of the ZU+ architecture. The TEE is also further extended to include access to the hardware entropy contained inside on-chip SRAM start-up patterns. The techniques used for integrating CSU cryptography and hardware entropy inside fTPM are presented in sections 6.3.2 and 6.3.3. The list of TPM 2.0 commands which can take advantage of hardware support can be found in appendix A.0.1.

6.3.2 Integration of the Hardware Cryptographic Support in OP-TEE and fTPM

Providing hardware support cryptography for fTPM has two advantages over a pure software implementation relying on cryptographic libraries (*wolfSSL* or *OpenSSL*). Several works have shown that cryptographic software implementations running inside ARM TrustZone might be vulnerable to SCA [110, 74] and cold-boot attacks [36]. Hardware cryptography in ZU+ is hardened against SCA and the keys are not stored in memory which prevents memory read out attacks. A second advantage is a possible performance improvement. For these reasons, hardware supported cryptography is preferable over a pure software implementation in the context of fTPMs.

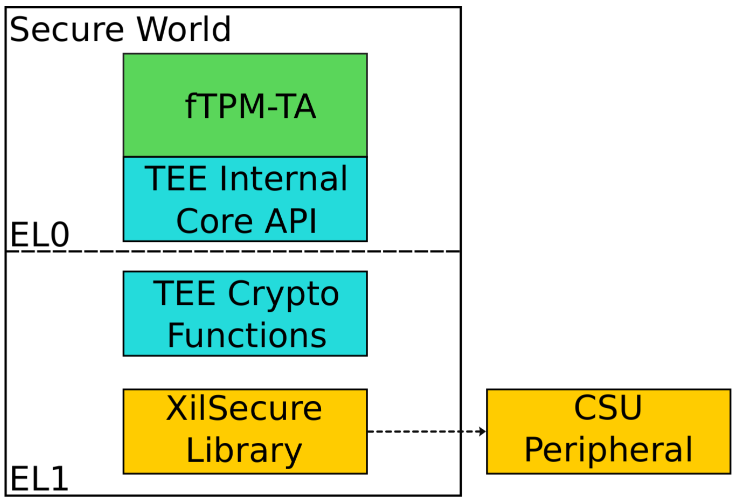


Figure 6.2 Integration of the CSU hardware cryptographic support in fTPM

The ZU+ FPGA-SoC contains a CSU that includes cryptographic hardware accelerators for AES-GCM 256, RSA 2048/4096, and SHA-3-384. These components are mainly used for the secure boot of the FPGA-SoC but are also accessible to the user via the *XilSecure* library. Since this library is intended for bare-metal usage, it cannot be integrated directly inside OP-TEE, where the ARM processor accesses memory via virtual addresses. In this chapter, we use an enhanced version of the TEE OP-TEE developed by Missing Link Electronics [19]. Their work enabled the integration of ZU+ security features inside OP-TEE. This was achieved by rewriting the bare-metal *XilSecure* library such that it can be used inside OP-TEE and adding drivers for the cryptographic accelerators to the TEE. The integration of the CSU cryptographic components is shown in figure 6.2. It consists in adding calls to the *XilSecure* library inside the TEE cryptographic functions. By doing so, the cryptographic accelerators are available to the OP-TEE OS. The hardware cryptographic modules can then be accessed from the user space via the TEE Internal Core API.

The integration of the hardware cryptographic support inside fTPM is achieved by rewriting the reference implementation such that cryptographic functions are not directly accessed via the libraries *wolfSSL* or *OpenSSL* but from the TEE Internal Core API instead.

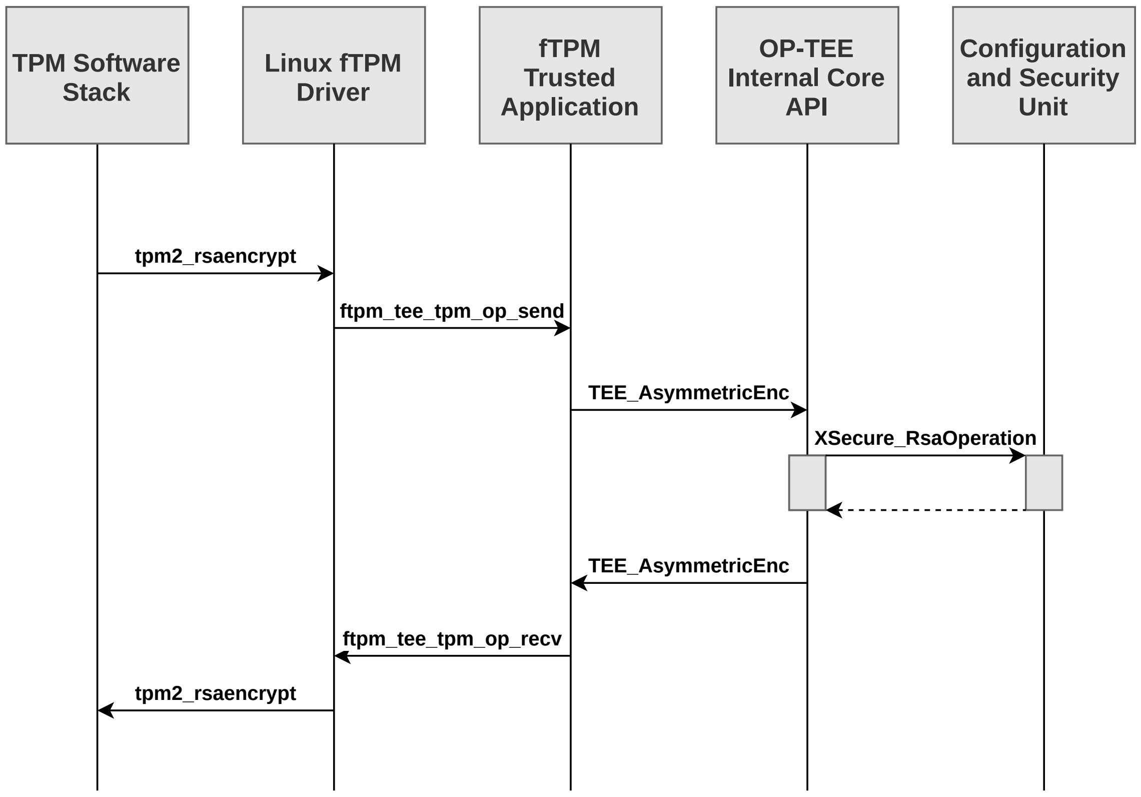


Figure 6.3 Handling of a TPM 2.0 RSA encryption command with hardware support

The handling of a TPM 2.0 RSA encryption command is depicted in figure 6.3. The user expresses its intention of performing an RSA encryption via the *tpm2_rsaencrypt* command. The Linux fTPM driver receives this command and transmits it to the fTPM-TA via the *ftpm_tee_tpm_op_send* command. fTPM interprets the received command and uses OP-TEE OS for performing the encryption via a call to the *TEE_AsymmetricEncrypt* function. Finally OP-TEE processes the encryption command by accessing the CSU RSA core via the *XilSecure* library.

6.3.3 Integration of the TRNGs Inside OP-TEE and fTPM

OP-TEE and fTPM handle the generation of random numbers via a DRBG (the Fortuna DRBG [21] is used for OP-TEE while fTPM relies on AES in counter mode). The output sequence produced by a DRBG is predictable to an attacker who manages to get access to the seed used for the initialization of the DRBGs. Therefore, NIST recommends the usage of a seed derived from a high entropy source and to periodically reseed the DRBG [10]. Both OP-TEE and fTPM rely on a low entropy source (physical counter registers) and recommend to use entropy obtained via hardware support instead. In this chapter, we use a high entropy source (see section 6.2 for details concerning the entropy extraction and its quality evaluation) that we XOR with the entropy source which is used by default. By doing so, we protect the DRBG seed from an adversary who tries to probe the SRAM start-up values which are readback by the AXI4 IP from figure 6.1.

The integration of the hardware entropy in OP-TEE OS consists in enhancing the reference implementation such that OP-TEE can use the procedure described in section 6.2.1. OP-TEE OS accesses the hardware entropy through an APU/PMU interaction code (see figure 6.1). This function is used to replace the DRBG initialization code of OP-TEE (*plat_rng_init* function in core/tee/tee_cryp_utl.c).

The APU/PMU interaction code cannot be used directly from the TEE user space inside a TA, because it requires to map some virtual addresses to specific PMU global registers. Therefore, we define a new syscall (*syscall_get_hw_rng_entropy* in tee_svc_cryp.c) and extend the internal core API with a new operation (*TEE_ZynqMPGetSRAMEntropy*), such that the hardware entropy can be obtained inside a TA via the call to this new operation. The last step consists in overwriting the *_plat__GetEntropy* function contained in the fTPM Entropy.c file, such that the new entropy source is used for the initialization and reseeding of the DRBG.

On most systems relying on the exploitation of SRAM start-up patterns, the hardware entropy can only be accessed once, because the restart of a system without disturbing the functionality is not always possible. With our methodology however, we verified that the bank B4 of TCM is indeed powered OFF and ON at each call to *TEE_ZynqMPGetSRAMEntropy*, resulting in new start-up values. This capability of regularly obtaining fresh SRAM entropy from the B4 bank increases the security of the DRBG used in fTPM.

6.4 Protecting the Bitstream Reconfiguration Interfaces via the Usage of ARM TrustZone and our hybrid-TPM

As explained in section 2.4.4.1, Xilinx considers the PCAP and ICAP as trusted under the assumption of secure boot [101]. Therefore, the user is allowed to readback the configured logic after the loading of an encrypted bitstream via this interface. Bitstream readback via the PCAP is an interesting debugging feature, however it should not be available on a fielded system where secure boot together with the loading of an encrypted bitstream is used. Even more surprising is the fact that this interface enables the loading of any bitstream type, regardless whether secure boot is enabled or not [101]. Due to these two security flaws, an adversary who achieves execution of its code after the secure boot process can introduce a malicious functionality inside the reconfigurable logic or potentially steal an intellectual property by reading back the configuration logic.

Several works described concepts for allowing a more secure remote configuration of FPGAs [95, 94, 48]. The works from [95, 94] introduced methods for secure remote partial reconfiguration of the FPGA fabric via the ICAP. Their design consisted in implementing the ICAP in a trusted static partition of the FPGA which cannot be modified by a user. A user who dynamically reconfigures an FPGA is then unable to use the ICAP in its partition. This prevents an attacker from loading an unauthenticated malicious bitstream or reading back the configuration logic via the ICAP. Recently, the work of [48] also points out the security limitations of the PCAP on the ZU+ architecture and propose a framework that disables the software access to the PCAP. Instead of that, they rely on the ICAP and the partial reconfiguration controller from Xilinx inside the reconfigurable logic. Their design uses a TrustZone logic checker between the ARM Cortex-A53 processor and the partial reconfiguration controller, to prevent the normal world OS from programming and reading back bitstreams in the secure area of the FPGA.

In this section, we present a solution that similarly prevents the normal world OS from using the PCAP, but instead of disabling the interface completely, we allow the secure world OS to use the PCAP and integrate the *XilFPGA* library inside the secure world OS. By doing this, we not only prevent insecure usage of the PCAP, but also integrate all the security features contained in the *XilFPGA* library inside a TEE.

6.4.1 Secure on the Fly Bitstream Loading via ARM TrustZone

The Xilinx software environment for loading bitstreams from Linux is depicted in figure 6.4. The framework uses the FPGA manager feature contained inside the kernel as well as the *libdfx* library [102] in the userspace. FPGA manager relies on the kernel features *sysfs* and *debugfs* for loading a bitstream and reading back the configuration logic. Xilinx also developed a specific driver *fpgautil* [106] which provides a more user friendly interface to the FPGA manager capabilities. Once a user has started an interaction with the Linux FPGA environment, ARM Trusted Firmware forwards the command to the *XilFPGA* library via a inter-processor interrupt with the PMU processor. This library implements the interaction with the PCAP which is located inside the CSU.

In a secure boot scenario, the *XilFPGA* library running on the PMU is authenticated and therefore Xilinx includes it in the trusted code base [101]. The problem with the scheme depicted in figure 6.4 is that the *XilFPGA* library enables the load of any bitstream types and allows a configuration readback of the FPGA even if the bitstream was loaded encrypted in a secure boot mode. Therefore, in this chapter we developed an alternative bitstream loading scheme which is depicted in figure 6.5.

This scheme uses a secure bitstream loading via OP-TEE, which is a feature integrated in the enhanced OP-TEE from Missing Link Electronics [19]. This is achieved by removing the *XilFPGA* library from the PMU and by enabling a direct communication between the APU SeW and the CSU, which contains the PCAP. The approach consists in adapt-

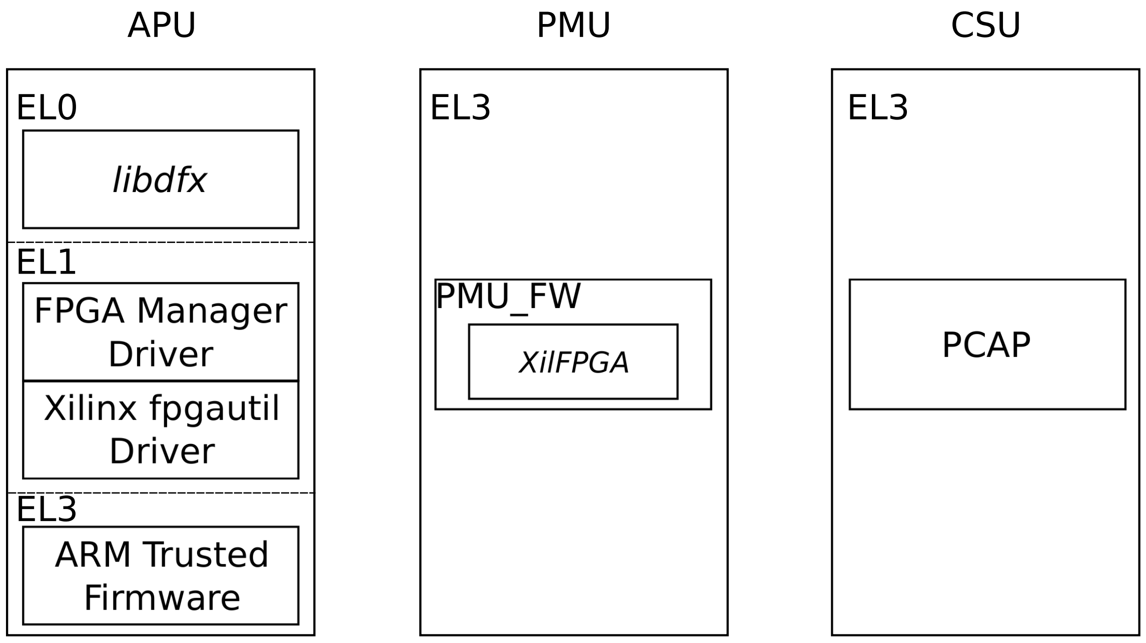


Figure 6.4 Software environment used for performing bitstream loading from Linux

ing the *XilFPGA* library such that it can be used inside OP-TEE. To this end, OP-TEE is enhanced with a control file system (*ctl_fs*) which can be seen as an analogy to *sysfs* and *debugfs*. Together with *ctl_fs*, an authentication state machine is used to perform bitstream authentication and decryption. The authentication state machine implements the security features contained inside the *XilFPGA* library. In practice, a user has to provide a bitstream from the NoW via a client application. The corresponding trusted application is using the *ctl_fs* backend for FPGA reconfiguration together with the authentication state machine to fetch the bitstream from the NoW, authenticate it, and decrypt it before sending it to the PCAP for reconfiguration.

A performance evaluation of the two frameworks is presented in section 6.5.3.

6.4.2 Combining Secure Bitstream Loading with fTPM Key Sealing

6.4.2.1 Motivation:

Data sealing consists of encrypting data with a key bounded to a given TPM device and under a specific TPM state. The TPM state is identified by particular PCR values. Once the data is sealed, it can only be unsealed with the same TPM device and TPM state.

In this chapter, we use data sealing to protect the decryption key of an encrypted partial bitstream, which is loaded after the successful completion of the secure boot mechanism. This decision is motivated by recent attacks targeting the secure boot of FPGA-SoCs [46, 20]. If an attacker manages to compromise the secure boot mechanism, she can program a bitstream that is encrypted with a key stored inside the device's eFuses or BBRAM. With sealing however, a bitstream decryption key can only be accessed if the device has booted a specific boot image which is measured by the TPM's PCRs.

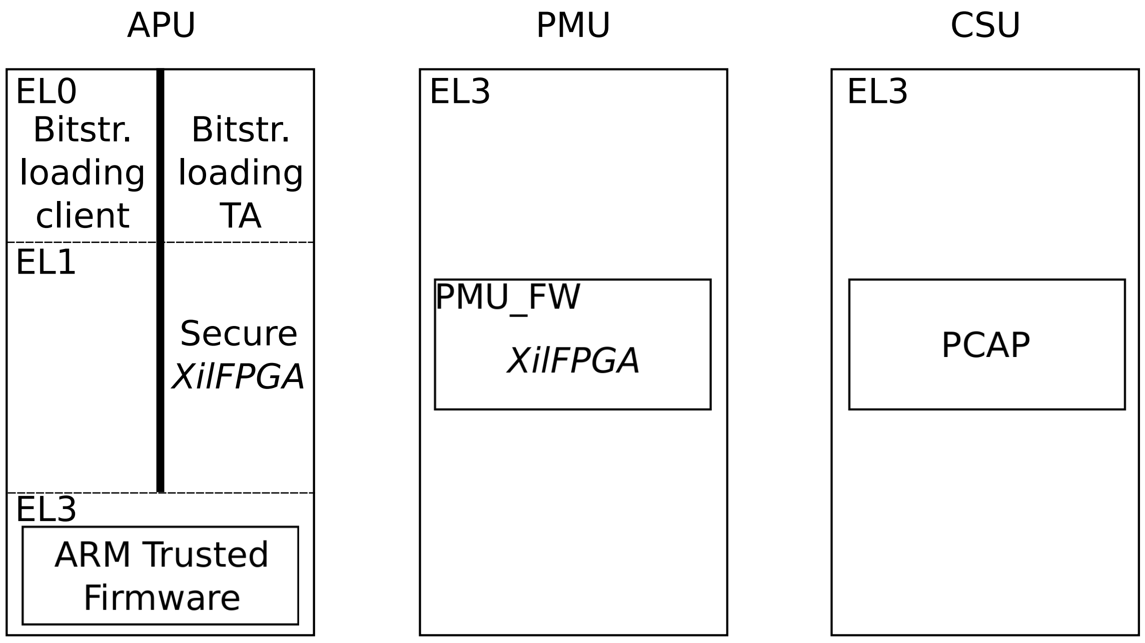


Figure 6.5 Software environment used for performing secure bitstream loading from OP-TEE

6.4.2.2 Sealing of a bitstream partial decryption key:

During the secure boot process, all the components depicted in figure 6.6 are authenticated with a device specific key. This mechanism ensures that our hybrid-TPM cannot be used on another device. The code used for the implementation of the sealing concept is an adaptation of the Xilinx Application note 1342: *Measured Boot of Zynq UltraScale+ Devices* [104] adapted for fTPM.

Upon power on, the CSU ROM is the first code that gets executed. This code is stored on chip in an immutable way and is responsible for device initialization, authentication of the FSBL before loading it into OCM. The FSBL is then performing authentication and loading of the subsequent boot partitions: a minimal static bitstream that configures the FPGA, PMU-FW, ATF, OP-TEE, and u-boot (which loads and measures a Linux Kernel and its root file system). In comparison to the standard FSBL that is generated from Xilinx tools, we have performed similar code modifications as the ones done in [104]. With these modifications the FSBL can measure each of the partitions it loads via a SHA-3-384 measurement. These measurements cannot be directly written to fTPM PCRs, since the FSBL is responsible for loading OP-TEE, which contains the fTPM-TA. Therefore the measurements of the components are stored in TrustZone protected memory before being usable by fTPM. After fTPM is loaded, it accesses the physical addresses containing the PCR values via a syscall and extends these values into PCRs[0 : 6]. With the correct boot measurement values stored in the PCRs, the sealing of a bitstream decryption key can be performed with the sequence of commands contained in listing 6.1.

In this example, the partial bitstream decryption key is sealed with the boot measurements contained inside the PCRs[0 : 6]. This is achieved via the help of commands from *tpm2-tools* version 2.10. Sealing requires the creation of a PCR policy based on the PCR[0 : 6] values and a primary key. Both of these objects are used to seal the object with the help of the TPM. Once the sealed object is created, the primary key is saved in

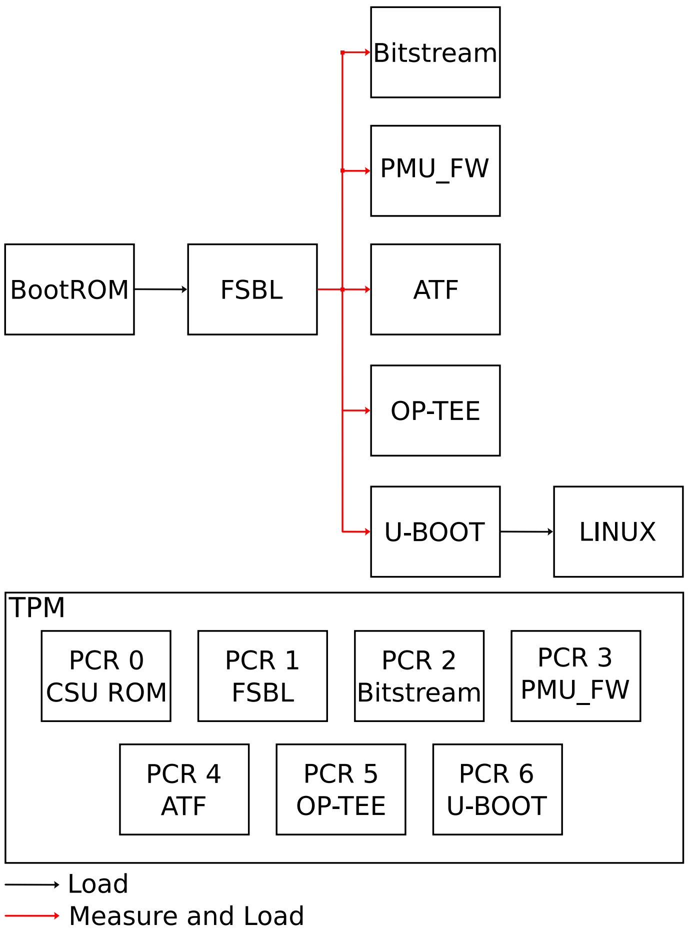


Figure 6.6 Creation of a partial bitstream decryption key with the measurements of the FPGA-SoC boot components

the persistent memory of the hybrid-TPM, such that it can be re-accessed after restarting the system.

6.4.2.3 Decryption of a partial bitstream with a sealed key:

Decryption of the partial bitstream first involves the recovery of the sealed key from the hybrid-TPM persistent memory through the load of an RSA key pair and the unsealing process. The sequence of commands for unsealing an AES key is precised in listing 6.2. We verified that the sealed partial bitstream decryption key can be unsealed after rebooting the FPGA-SoC, with the requirement that PCRs[0 : 6] contain the correct value. Once unsealed, this key is going to be used for loading an encrypted bitstream with a user provided key within the framework from section 6.4.1.

```
tpm2_pcrlist -L sha384:0,1,2,3,4,5,6 -o pcr.bin
tpm2_createpolicy -P -L sha384:0,1,2,3,4,5,6 -F pcr.bin -f policy.
    ↪ digest
tpm2_createprimary --hierarchy e -g sha256 -G rsa --out-context
    ↪ primary.context
tpm2_create -g sha256 -C primary.context -u obj.pub -r obj.priv -L
    ↪ policy.digest -I partial_bitstream_key
tpm2_evictcontrol -a o -c primary.context -p 0x81000001
```

Listing 6.1 Sealing a bitstream decryption key with fTPM

```
tpm2_load -C 0x81000001 -u obj.pub -r obj.priv -n load.name -o load.
    ↪ context
tpm2_unseal -c load.context -L sha384:0,1,2,3,4,5,6
```

Listing 6.2 Unsealing a bitstream decryption key with fTPM

6.5 Performance Evaluation

This section first compares the performance of the standard implementations of AES-GCM, RSA, and SHA-3-384 in OP-TEE and fTPM to a hardware accelerated version. Afterwards, we compare the performance of an encrypted bitstream load via the framework proposed in section 6.4 against the classical approach suggested by Xilinx.

6.5.1 Hardware Accelerated Cryptography vs Software Implementation in OP-TEE

The performance of the OP-TEE software cryptographic implementation (*LibTomCrypt*) is compared to the CSU enhanced implementation done by Missing Link Electronics (see section 6.3.2) in tables 6.4, 6.5, and 6.6. The execution times are measured inside the secure world via the *TEE_GetSystemTime* function from the TEE Internal Core API. This utility measures the system time by using the CPU frequency and the ARM Physical Count register. When possible, the performance of the two cryptographic implementations are also compared with the ARMv8-A cryptographic extension. In a real world scenario, the end user accesses cryptographic services of the TEE from a NoW client application. In that case, it is necessary to add a latency for switching between the NoW and the SeW. We measure an estimation of this time overhead to be 234 microseconds with a TA implementing a simple counter incrementation.

The performance measurements done in this chapter are comparable to the values obtained by Xilinx in [98]. In addition to the security advantages of a hardware accelerated solution against memory and SCA, our measurements show that CSU accelerated cryptography can significantly speedup RSA operations and increase the SHA-3-384 throughput for input data bigger than 1 kB due to input buffering [103]. Concerning AES operations, the CSU offer speedup over a pure software implementation from *LibTomCrypt* but the communication overhead for interacting with the AES hardware accelerator is outperformed by AES operations executing directly via special instructions on the ARM Cortex-A53.

Input size (bytes)	Software (MB/s) only	ARM v8 cryptographic extensions (MB/s)	CSU (MB/s)
16	0.59	**1.06**	0.51
528	5.07	**40.61**	22
1024	5.59	**73.14**	39.38
4112	6.10	**178.78**	95.62
7696	6.19	**233.21**	124.12
15 888	6.24	**283.71**	147.1

Table 6.4 AES-GCM-256 encryption throughput

Input size (bytes)	Software only (MB/s) [98]	CSU (MB/s)
16	**41.58**	0.94
528	**59.63**	35.2
1024	**60.80**	56.88
4112	62.03	**152.29**
7696	62.36	**150.90**
15 888	62.29	**220.66**

Table 6.5 SHA3-384 hashing throughput

6.5.2 Hardware Backed Cryptography vs Software Implementation in fTPM

The reference implementation of fTPM relies on the *wolfSSL* cryptographic library. This library provides a fast software implementation of the AES, SHA-3-384, and RSA primitives.

TPMs were initially designed as external chips which integrate a low performance microcontroller. Therefore, TPMs are not intended to encrypt/hash large blocks of data. Software components are typically measured outside of the TPM and only a hash of a measurement is extended to a TPM's PCRs via the hashing engine of the TPM. Similarly, TPMs are not used to encrypt or decrypt a hard drive, but rather to store an encrypted version of the hard drive decryption key. This limitation in the data size is also reflected in the maximum command and response size supported by the *tpm2-tools* software. To be compliant with this software, fTPM only allows a maximum command and response size of 4 kB. This data size constraint prevents a speedup for the SHA-3-384 hashing or the AES encryption/decryption of large data via the CSU. On the other hand, it is still possible to obtain a significant improvement of performance for the RSA 2048 decryption and signing operations (see table 6.7). Despite providing only a performance improvement for RSA, relying on the hardware cryptography is still beneficial from a security point of view as it improves the resistance of fTPM against SCA and memory read-out attacks.

6.5.3 Encrypted Bitstream Load from TrustZone vs Encrypted Bitstream Load from Linux

An encrypted bitstream of 7 MB containing an AXI General Purpose I/O (GPIO) controller accessible to the ZU+ processing system was used for the performance evaluation of the encrypted bitstream loading feature contained in the extended OP-TEE used in this chapter. We measure the time necessary for configuring a user key inside the ZU+ and

RSA operation	Software only (ms)	CSU (ms)
Encrypt 2048	1.74	**1.21**
Decrypt 2048	47.22	**26.88**
Encrypt 4096	12.04	**4.23**
Decrypt 4096	312.93	**195.39**

Table 6.6 RSA 2048 operation time

RSA operation	Software only (ms)	CSU (ms)
Encrypt	1.64	**1.26**
Decrypt	115.74	**30.36**
Verify	1.65	**1.25**
Sign	115.75	**30.38**

Table 6.7 RSA 2048 operation time (fTPM)

decrypting the bitstream with that key for the framework presented in this chapter and the one suggested by Xilinx (see section 6.4.1).

The timing measurements were done via the *time* command from Linux and averaged over 50 distinctive bitstream loads. The performance measurements obtained via the two frameworks are contained in table 6.8. The real time is the time necessary for the complete operation while the system time corresponds to the time spent in kernel space. Despite being less secure, the Xilinx framework appears to be faster and to rely less on the kernel. The OP-TEE framework on the other hand relies more on the kernel, since the *XilFPGA* library is running on the APU SeW at kernel privilege level instead of being run on the PMU-FW (see figures 6.4 and 6.5). The framework we developed appears to be slower than the standard Xilinx's framework. The main advantage of the OP-TEE framework is its security, as it prevents the load of insecure bitstream and the readback of the FPGA configuration after an encrypted bitstream load.

	Real time (ms)	System time (ms)
Xilinx framework	465	169
OP-TEE framework	867	419

Table 6.8 Timings for loading a bitstream encrypted with a user defined key

6.6 Discussion and Future Work

This section describes the advantages and limitations of our hybrid-TPM in comparison to the reference implementations of fTPM and discrete TPMs. In a second part, possible improvements reserved for future work are discussed.

6.6.1 Comparison with Other TPM Designs

A TEE is also conceivable on Intel platforms, via the use of Intel Software Guard Extensions (SGX) [18]. This technology enables an isolated execution environment by using enclaves, which are isolated memory regions only accessible to the enclave process itself.

Despite not being a TPM, Intel SGX contains several TPM features, such as remote attestation. Sun et al. [90] have shown that this technology is suitable for building TPMs inside enclaves in the cloud.

More recently, Chakraborty et al. [14] presented simTPM, a software TPM running inside a Subscriber Identity Module (SIM) card. simTPM has a good synergy with TPMs running inside a TEE. This is achieved by using the TEE-TPM as a trusted interface between the user and the SIM card. This implementation achieves better system wide isolation than fTPM. However, we chose to design a hybrid-TPM by improving fTPM, since in our opinion this implementation fits better with FPGA-SoC platforms.

Zhao et al [112] proved that secure key generation and access to hardware entropy can be added to ARM TrustZone via the usage of SRAM start-up patterns of an external chip. The hardware entropy source is then used to seed a software DRBG of the TEE. They further include the two previous features inside a software TPM compliant to the TPM 1.2 standard. By doing so, they turned a purely software TPM implementation into a hybrid hardware/software implementation. Recently, Kim et al. [50] have shown that the alternative approach, which consists in extending a hardware TPM with software support can also be beneficial, especially from a performance point of view.

The advantages and drawbacks of Microsoft's fTPM and our hybrid-TPM in comparison to dTPMs are listed in table 6.9.

One interesting benefit of fTPM and our hybrid-TPM over dTPMs is a performance improvement (see table 6.7 and performance comparison numbers from [84, 14]). This is achieved through the execution of the TPM inside a powerful ARM processor combined with cryptographic hardware accelerators. In contrast, dTPMs rely on a low-performance microcontroller. This performance improvement is interesting for integrating TPMs in applications requiring fast operations, where discrete TPMs might not be suitable.

dTPMs are usually significantly tested such that they can cope with security certifications such as Common Criteria EAL4+ or FIPS 140-2. Therefore, dTPMs are the best choice for achieving the hardware security objectives mentioned in table 6.9. fTPM is weaker from a hardware security point of view. It does not contain tamper and SCA protection mechanisms and the original paper [84] explicitly mentions that the fTPM should have access to an entropy pool within ARM TrustZone. With our hybrid-TPM, we aim in partially fulfilling some of these goals by using the security mechanisms contained inside the ZU+ FPGA-SoC. The secure boot of the ZU+ contributes in binding our hybrid-TPM to a specific device via the usage of a device specific key for authenticating the boot image. The configurable tamper protection and response mechanisms available on the device also contribute to improving the overall hardware protection. The AES cryptographic hardware accelerator available on the ZU+ contains a protocol-based countermeasure against SCA which exploits rekeying. This ensures that a given key can only be used for decrypting a certain amount of data blocks. Although a recent work [37] pointed an observable leakage of the AES cryptographic core, the sophisticated attack developed in their work couldn't break the AES decryption engine because of the rekeying mechanism. By integrating the cryptographic accelerators available on the ZU+ inside our hybrid-TPM, we increase the resistance of our hybrid-TPM against SCA and cold-boot attacks. One major contribution of our work is the integration of a good entropy source to our hybrid-TPM DRBG, such that it becomes impossible for an attacker to guess the DRBG sequence due to a poor entropy pool. The secure storage remains a feature not directly integrated inside fTPM, instead it relies on an eMMC module or a software assisted secure data storage. Our hybrid-TPM also uses the second method. A further description of the secure data storage used in this chapter is provided in appendix A.0.2.

Finally an important advantage of fTPMs and our hybrid-TPM is their adaptability. In contrast to dTPMs which could not be fixed easily after a security incident, fTPMs software can be adapted and updates can be made on systems already deployed in the field. This flexibility also enables the integration of new coming cryptographic standards, such as the ones currently reviewed in the NIST post-quantum cryptography competition.

	fTPM	**hybrid-TPM**	**dTPM**
Performance	✓	✓	✗
Hardware protection	✗	★	✓
Secure storage	★	★	✓
SCA resistance	✗	★	✓
True entropy	✗	✓	✓
Pre-boot availability	✗	✗	✓
Adaptability	✓	✓	✗

✓ = fulfilled; ★ = partially fulfilled; ✗ = not fulfilled

Table 6.9 Comparison of advantages and drawbacks of different TPM's design choices

6.6.2 Future Work

In this chapter, we described a methodology to derive a true random seed out of the noise from on-chip SRAM start-up patterns (see section 6.2). This primitive can be extended to construct a PUF or alternatively a PUF can be implemented within the reconfigurable logic. The manufacturer already integrated a PUF on the FPGA-SoC, which is used for generating a unique device key that encrypts/decrypts a user provided "red key" into a "black key" that can be stored in the eFuses or in the bitstream header [2]. However, the PUF primitive from the manufacturer cannot be accessed by the user and therefore it is not possible to integrate it into our hybrid-TPM. A relevant extension to this chapter would be to design a PUF that is exclusively accessible to the TPM.

Despite the integration of the hardware cryptographic accelerators inside our hybrid-TPM, the TPM remains executed in DDR memory with system level isolation provided by an ARM TrustZone TEE. Achieving the execution of the TEE and the hybrid-TPM in on-chip SRAM would provide better protection against the Rowhammer [51] and cold-boot [36] attacks. However, this is challenging since the OP-TEE kernel requires already more than 256 kB of memory, which is the amount of OCM available on the platform. In addition, other components such as ATF and the FSBL are executing in OCM on the ZU+ FPGA-SoC. Integrating the secure paging support from OP-TEE in the ZU+ build enables the use of virtual memory and thus might help in achieving a TEE execution inside OCM.

Last but not least, TPM still relies on asymmetric cryptographic algorithms which can be broken by a quantum computer. An interesting extension to this chapter would be to add support for post quantum algorithms to the TPM 2.0 software stack. These algorithms can be designed as hardware accelerators inside the reconfigurable logic and used to enhance the performance and security of our hybrid-TPM.

6.7 Summary

This chapter explores the security benefits achievable through the usage of a TEE based on ARM TrustZone on the ZU+ architecture. First we have adapted Microsoft's fTPM implementation, a software TPM running inside ARM TrustZone for the ZU+ platform. The modifications consist in using the cryptographic hardware accelerators inside the TPM and to extend it with an entropy source derived from on-chip SRAM. The two adaptations we made to fTPM transform the purely software implementation into a hybrid hardware/software implementation which is more resistant against side-channel and memory read-out attacks while remaining updatable in the event of a security incident.

Second, we identified some security limitations in the usage of the PCAP and designed a framework that lead to a more secure utilization of this interface. This framework consists in preventing the load of non-authenticated bitstreams and readback of the configuration logic after secure boot. Furthermore, it only allows secure bitstream loading via ARM TrustZone. We showed that both contributions of this work can be combined together, in an example that uses the sealing of a partial bitstream decryption key via our hybrid-TPM together with secure bitstream loading inside TrustZone.

Overall, the two concepts presented in this chapter are compatible and complementary with the security mechanisms implemented by Xilinx on the ZU+. They can be easily integrated in devices already deployed in the field and do not require the usage of additional hardware.

7 Conclusion

In this thesis, we have shown the practicability and impact of remote attacks performed by FPGA logic on FPGA-SoC platforms. In contrast to remote attacks relying on software only, using malicious FPGA logic enables an attacker to bypass isolation mechanisms implemented in hardware or within the OS.

As first example of such attacks, we demonstrated how malicious logic can manipulate memory and peripherals on modern FPGA-SoCs based on the ZU+ architecture from Xilinx. This architecture contains system level isolation mechanisms which should mitigate such attacks. Despite these isolation mechanisms, we showcase that malicious logic using the ACP on this architecture can compromise the memory of a TEE implemented with ARM TrustZone and alter the secure boot process.

As a second FPGA induced remote attack, we demonstrate the possibility of faulting the execution of software running on one of the FPGA-SoC's embedded processor cores. This type of attacks is possible through the sharing of the PDN between the FPGA and the other processing units available on some FPGA-SoCs. By generating a voltage drop via dedicated FPGA logic, we achieve the faulting of data during its transfer from DDR memory to the processor's cache hierarchy. This fault injection methodology is also used for compromising the execution of software by skipping CPU instructions or altering their result. From an attack practicability perspective, we implemented a DFA on AES by using this fault injection methodology.

The implementation of the attacks described in this thesis requires the insertion of malicious logic and its activation via a covert communication channel. Due to relaxed trust assumptions in the FPGA reconfiguration interfaces, the insertion of malicious logic contained in an un-authenticated bitstream is possible on Xilinx FPGA-SoCs even after the secure boot of a device. For the malicious logic activation, we demonstrated a CPU to FPGA power covert channel, where a specific sequence of instructions executed on the CPU is used to encode a malicious logic activation signal within the power consumption of an FPGA-SoC.

Finally we have presented the security advantages of using a TEE on FPGA-SoCs. Our experiments demonstrated that the built-in security features of the ZU+ and a software TPM could be used for implementing a hybrid hardware/software TPM running inside ARM TrustZone. Furthermore, we showcase how the developed hybrid-TPM and a TEE could help in improving the security of FPGA-SoCs. This is achieved through a better protection of the bitstream reconfiguration interfaces and an extension to secure boot implemented with the hybrid-TPM.

In conclusion, this thesis gives a comprehensive overview of remote threats faced by modern FPGA-SoCs, their possible impacts and discuss protection techniques. Due to the complexity of such systems, achieving isolation while sharing resources such as memories, peripherals, and a PDN make these systems hard to secure and prone to powerful attacks implemented via the FPGA logic. With the current trend, where FPGA logic IP developped by third parties can be found on cloud providers catalog, we believe that tools for verifying FPGA bitstreams become necessary. Such tools already exist for open source bitstream formats such as the one of Lattice FPGA [57] but adapting them for proprietary bitstream formats remains challenging.

A Appendix

A.0.1 Hardware Support for TPM Commands

This work uses the version 2.10 of *tpm2-tools*. With this specific version, we were able to provide hardware support (via cryptographic hardware and with the introduction of hardware entropy) to the commands listed in table A.1. As future work, the commands *tpm2_nvread* and *tpm2_nvwrite* could benefit from the user programmable eFuses available on the ZU+ FPGA-SoC.

Command	**Brief description**
tpm2_certify	Certify that an object is loaded in the TPM
tpm2_create	Create a key or sealing object inside the TPM
tpm2_unseal	Returns sealed data in clear if the PCR values are matching the PCR policy used for sealing the object
tpm2_createprimary	Create a primary key
tpm2_encryptdecrypt	Symmetric encryption/decryption
tpm2_getpubak	Generate attestation key pair and return public attestation key
tpm2_getpubkek	Generate endorsement key pair and return public endorsement key
tpm2_hash	Generate hash of supplied data
tpm2_hmac	Perform HMAC on supplied data
tpm2_quote	Provide a quote and signature from the TPM
tpm2_pcrextend	Extend PCRs with provided values
tpm2_rsadecrypt	Perform RSA decryption
tpm2_rsaencrypt	Perform RSA encryption
tpm2_sign	Generate signature of supplied data
tpm2_verifysignature	Verify signature
tpm2_getrandom	Generate random bytes

Table A.1 TPM commands which benefit from hardware support

A.0.2 Software Assisted Secure Data Storage in OP-TEE

Secure storage is a crucial functionality offered by TPMs. While dTPMs have a tamper proof non-volatile memory which is used for that purpose, this capability is not available with fTPMs. As an alternative, Microsoft implemented secure storage in fTPM by relying on the Replay Protected Memory Block (RPMB) partition of an embedded MultiMedia Card (eMMC) device. This feature is available inside OP-TEE under the condition that the software image is flashed on an eMMC device. Although eMMC devices are compatible with the ZU+ FPGA-SoCs, the user is generally using an SD-Card as an embedded storage medium on these platforms. Therefore, in this chapter we chose to provide secure storage to our hybrid-TPM via a software assisted support available in OP-TEE. With this implementation, secret data created by the TPM is going to be stored encrypted inside the normal world RootFS and modification and access to this data is only possible for the fTPM-TA. To be compliant with the TEE Internal Core API specification, the secure storage is also implemented with an integrity guarantee.

The software assisted secure storage relies on the use of several encryption keys. The Secure Storage Key (SSK), which is a per device key generated and stored in secure memory during boot. This key is derived from a Hardware Unique Key (HUK) and a ChipID as indicated in equation A.1. The HUK is a device specific key and the software to access this key should be implemented in *tee_otp_get_hw_unique_key* according to the device family OP-TEE is running on. In the context of the FPGA-SoCs considered in this thesis, the HUK can be stored in the eFuses or BBRAM. The PUF integrated by the manufacturer on the Xilinx ZU+ or the Intel Stratix 10 cannot be used for that purpose, as the access to PUF key is not permitted.

$$SSK = HMAC_{SHA256}(HUK, ChipID||''static\ string'') \tag{A.1}$$

The SSK serves as a basis for deriving other encryption keys. One of them are the Trusted Application Storage Keys (TSKs), a per TA key used to protect the different File Encryption Keys (FEKs). A TSK is obtained from the SSK and the TA_UUID according to equation A.2.

$$TSK = HMAC_{SHA256}(SSK, TA\_UUID) \tag{A.2}$$

Each generation of a TEE file inside a TA comes with the generation of a new FEK. This key is generated by a PRNG (which was seeded with a good entropy source as described in section 6.3.3) and is further used to encrypt the meta data of the file and the data blocks composing it. Meta data encryption results in the creation of the MetaData Field as explained in equation A.3:

$$\begin{cases} FEK_{crypt} = AES - ECB(FEK, TSK) \\ (MetaData_{crypt},\ TAG) = AES - GCM(MetaData,\ IV,\ FEK_{crypt}) \\ MetaData\ Field = (FEK_{crypt}\ ||\ IV\ ||\ TAG\ ||\ MetaData_{crypt}) \end{cases} \tag{A.3}$$

Similarly, the encryption of a data block results in the creation of a Data Block Field as defined in equation A.4:

$$\begin{cases} (DataBlock_{crypt}\ ||\ TAG) = AES - GCM(DataBlock, IV, FEK) \\ DataBlock\ Field = ((DataBlock_{crypt}\ ||\ TAG\ ||\ IV) \end{cases} \tag{A.4}$$

As explained in equations A.3 and A.4, a file used to store secure data is encrypted with a per TA specific key. Therefore, this file is only accessible to a specific TA. As a result, secret data created via fTPM is neither accessible to the normal world nor to the others TAs.
The encrypted file is stored inside a hash tree, where the Hash Tree Header contains the MetaData Field from equation A.3 and where each node contains the TAG and IV of a DataBlock Field. A secure data file consists of the encrypted data blocks and the generated

Hash Tree. This file (and its backup) are stored in the RichOS RootFS under /data/tee.

The ZU+ FPGA-SoC contains user programmable eFuses which can be used in combination with the secure storage implementation described in this section. eFuses offer non-volatile storage but are only one time programmable. The integration of the ZU+ user programmable eFuses into fTPM is reserved for future work.

List of publications

[30] Mathieu Gross et al. "Breaking TrustZone memory isolation and secure boot through malicious hardware on a modern FPGA-SoC". en. In: *Journal of Cryptographic Engineering* (Sept. 2021). ISSN: 2190-8516. DOI: `10.1007/s13389-021-00273-8`.

[31] Mathieu Gross et al. "Breaking TrustZone Memory Isolation through Malicious Hardware on a Modern FPGA-SoC". In: *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*. ASHES'19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 3–12. ISBN: 9781450368391. DOI: `10.1145/3338508.3359568`. URL: `https://doi.org/10.1145/3338508.3359568`.

[32] Mathieu Gross et al. "Enhancing the Security of FPGA-SoCs via the Usage of ARM TrustZone and a Hybrid-TPM". In: *ACM Trans. Reconfigurable Technol. Syst.* 15.1 (Nov. 2021). ISSN: 1936-7406. DOI: `10.1145/3472959`. URL: `https://doi.org/10.1145/3472959`.

[33] Mathieu Gross et al. "FPGANeedle: Precise Remote Fault Attacks from FPGA to CPU". In: *Proceedings of the 28th Asia and South Pacific Design Automation Conference*. ASPDAC '23. Tokyo, Japan: Association for Computing Machinery, 2023, pp. 358–364. ISBN: 9781450397834. DOI: `10.1145/3566097.3568352`. URL: `https://doi.org/10.1145/3566097.3568352`.

List of publications not included in this thesis

[87] Johanna Sepúlveda et al. "Beyond Cache Attacks: Exploiting the Bus-Based Communication Structure for Powerful On-Chip Microarchitectural Attacks". In: *ACM Trans. Embed. Comput. Syst.* 20.2 (Mar. 2021). ISSN: 1539-9087. DOI: `10.1145/3433653`. URL: `https://doi.org/10.1145/3433653`.

Bibliography

[1] Lester Sanders (Xilinx). *Isolation Methods in Zynq Ultrascale+ MPSoCs*. XAPP1320 (v4.0). July 2021.

[2] Nathan Menhorn (Xilinx). *External Secure Storage Using the PUF*. XAPP1333 (v1.0). June 2018.

[3] *AI and Compute*. `https://openai.com/blog/ai-and-compute/`. 2023.

[4] Md Mahbub Alam et al. "RAM-Jam: Remote temperature and voltage fault attack on FPGAs using memory collisions". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE. 2019, pp. 48–55.

[5] ARM. *ARM Cortex-A53 MPCore Processor Technical Reference Manual*. revision r0p2. Feb. 2014.

[6] ARM. *ARM Security Technology - Build a Secure System using TrustZone Technology*. Issue D.c. June 2016.

[7] ARM. *ARM System Memory Management Unit Architecture Specification - SMMU architecture version 2.0*. Issue C. Apr. 2009.

[8] D. Aumaitre and C. Devine. "Subverting Windows 7 x64 kernel with DMA attacks". In: *HITBSecConf Amsterdam*. 2010.

[9] Elaine Barker and John Kelsey. *NIST Special Publication 800-90A (A Revision of SP 800-90) Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. 2012.

[10] Lawrence E. Bassham et al. *SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Tech. rep. Gaithersburg, MD, USA, 2010.

[11] E. M. Benhani, L. Bossuet, and A. Aubert. "The Security of ARM TrustZone in a FPGA-Based SoC". In: *IEEE Transactions on Computers* 68.8 (2019), pp. 1238–1248. DOI: `10.1109/TC.2019.2900235`.

[12] A. Bogdanov et al. "PRESENT: An Ultra-Lightweight Block Cipher". In: *Cryptographic Hardware and Embedded Systems - CHES 2007*. Ed. by Pascal Paillier and Ingrid Verbauwhede. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 450–466. ISBN: 978-3-540-74735-2.

[13] Gaetan Canivet et al. "Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-Based FPGA". In: *J. Cryptology* 24 (2011), pp. 247–268. DOI: `10.1007/s00145-010-9083-9`.

[14] Dhiman Chakraborty, Lucjan Hanzlik, and Sven Bugiel. "simTPM: User-centric TPM for Mobile Devices". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 533–550. ISBN: 978-1-939133-06-9. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/chakraborty`.

[15] Sumanta Chaudhuri. "A Security Vulnerability Analysis of SoCFPGA Architectures". In: *Proceedings of the 55th Annual Design Automation Conference*. DAC '18. San Francisco, California: ACM, 2018, 139:1–139:6. ISBN: 978-1-4503-5700-5. DOI: `10.1145/3195970.3195979`. URL: `http://doi.acm.org/10.1145/3195970.3195979`.

[16] Yue Chen et al. *Downgrade Attack on TrustZone*. 2017. DOI: `10.48550/ARXIV.1707.05082`. URL: `https://arxiv.org/abs/1707.05082`.

[17] Patrick Colp et al. "Protecting Data on Smartphones and Tablets from Memory Attacks". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: ACM, 2015, pp. 177–189. ISBN: 978-1-4503-2835-7. DOI: `10.1145/2694344.2694380`. URL: `http://doi.acm.org/10.1145/2694344.2694380`.

[18] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Paper 2016/086. `https://eprint.iacr.org/2016/086`. 2016. URL: `https://eprint.iacr.org/2016/086`.

[19] Missing Link Electronics. *MLE OP-TEE for Zynq Ultrascale+ devices*. `https://www.missinglinkelectronics.com/security`. 2022.

[20] F-Secure. *Security advisory: Xilinx ZU+ Encrypt Only Secure Boot bypass*. `https://github.com/f-secure-foundry/advisories/blob/master/Security_Advisory-Ref_FSC-HWSEC-VR2019-0001-Xilinx_ZU+-Encrypt_Only_Secure_Boot_bypass.txt`. 2019.

[21] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. 1st ed. USA: John Wiley and Sons, Inc., 2003. ISBN: 0471223573.

[22] Ilias Giechaskiel, Kasper Bonne Rasmussen, and Jakub Szefer. "C3APSULe: Cross-FPGA Covert-Channel Attacks through Power Supply Unit Leakage". In: *IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 1728–1741.

[23] Global Platform. *Introduction to Trusted Execution Environments*. May 2018.

[24] Global Platform. *TEE Protection Profile*. Sept. 2020.

[25] Dennis R E Gnad, Fabian Oboril, and Mehdi B Tahoori. "Voltage Drop-based Fault Attacks on FPGAs using Valid Bitstreams". In: *Field Programmable Logic and Applications (FPL)*. Ghent, Belgium: IEEE, Sept. 2017, pp. 1–7. DOI: `10.23919/fpl.2017.8056840`.

[26] Dennis R. E. Gnad et al. "An Experimental Evaluation and Analysis of Transient Voltage Fluctuations in FPGAs". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.10 (2018), pp. 1817–1830. DOI: 10.1109/TVLSI.2018.2848460.

[27] Dennis R. E. Gnad et al. "Voltage-Based Covert Channels Using FPGAs". In: *ACM Trans. Des. Autom. Electron. Syst.* 26.6 (June 2021). ISSN: 1084-4309. DOI: 10.1145/3460229. URL: https://doi.org/10.1145/3460229.

[28] Joseph Gravellier et al. "Remote Side-Channel Attacks on Heterogeneous SoC". In: *Smart Card Research and Advanced Applications, 18th International Conference, CARDIS 2019*. Pragues, Czech Republic, Nov. 2019. URL: https://hal.archives-ouvertes.fr/hal-02380092.

[29] Hannes Gross, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order". In: *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security*. TIS '16. Vienna, Austria: Association for Computing Machinery, 2016, p. 3. ISBN: 9781450345750. DOI: 10.1145/2996366.2996426. URL: https://doi.org/10.1145/2996366.2996426.

[34] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez. Cham: Springer International Publishing, 2016, pp. 300–321. ISBN: 978-3-319-40667-1.

[35] Meeta S. Gupta et al. "Understanding Voltage Variations in Chip Multiprocessors using a Distributed Power-Delivery Network". In: *2007 Design, Automation & Test in Europe Conference & Exhibition*. 2007, pp. 1–6. DOI: 10.1109/DATE.2007.364663.

[36] J. Alex Halderman et al. "Lest We Remember: Cold-boot Attacks on Encryption Keys". In: *Commun. ACM* 52.5 (May 2009), pp. 91–98. ISSN: 0001-0782. DOI: 10.1145/1506409.1506429. URL: http://doi.acm.org/10.1145/1506409.1506429.

[37] Benjamin Hettwer et al. "Side-Channel Analysis of the Xilinx Zynq UltraScale+ Encryption Engine". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.1 (2021), pp. 279–304. DOI: 10.46586/tches.v2021.i1.279-304. URL: https://doi.org/10.46586/tches.v2021.i1.279-304.

[38] Daniel Holcomb, Wayne Burleson, and Kevin Fu. "Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers". In: *Computers, IEEE Transactions on* 58 (Oct. 2009), pp. 1198–1210. DOI: 10.1109/TC.2008.212.

[39] Taras Iakymchuk, Maciej Nikodem, and Krzysztof Kępa. "Temperature-based covert channel in FPGA systems". In: *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*. 2011, pp. 1–7. DOI: 10.1109/ReCoSoC.2011.5981510.

[40] R. Impagliazzo, L. A. Levin, and M. Luby.
"Pseudo-Random Generation from One-Way Functions".
In: *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*. STOC '89.
Seattle, Washington, USA: Association for Computing Machinery, 1989, pp. 12–24. ISBN: 0897913078. DOI: `10.1145/73007.73009`.
URL: `https://doi.org/10.1145/73007.73009`.

[41] Intel. *Cyclone V Hard Processor System Technical Reference Manual*. v21.2. Aug. 2022.

[42] Intel. *Intel Stratix 10 Device Security User Guide*. v22.2. July 2022.

[43] Intel. *Intel Stratix 10 Hard Processor System Technical Reference Manual*. v21.4. Aug. 2022.

[44] Intel. *Strengthening Security with Intel Platform Trust Technology*. v 1.0. June 2014.

[45] Nisha Jacob et al.
"Compromising FPGA SoCs Using Malicious Hardware Blocks".
In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '17.
Lausanne, Switzerland: European Design and Automation Association, 2017, pp. 1122–1127.

[46] Nisha Jacob et al.
"How to Break Secure Boot on FPGA SoCs Through Malicious Hardware".
In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Vol. 10529.
Lecture Notes in Computer Science. Springer, 2017, pp. 425–442.
DOI: `10.1007/978-3-319-66787-4\_21`.

[47] Antoine Joux. "Authentication failures in NIST version of GCM". In: (Jan. 2006).

[48] N. Khan et al. "Utilizing and Extending Trusted Execution Environment in Heterogeneous SoCs for a Pay-Per-Device IP Licensing Scheme".
In: *IEEE Transactions on Information Forensics and Security* 16 (2021), pp. 2548–2563.
DOI: `10.1109/TIFS.2021.3058777`.

[49] Jeremie S. Kim et al. "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques". In: *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. ISCA '20. Virtual Event: IEEE Press, 2020, pp. 638–651. ISBN: 9781728146614.
DOI: `10.1109/ISCA45697.2020.00059`.
URL: `https://doi.org/10.1109/ISCA45697.2020.00059`.

[50] Yongjin Kim and Evan Kim.
"HTPM: Hybrid Implementation of Trusted Platform Module".
In: *Proceedings of the 1st ACM Workshop on Workshop on Cyber-Security Arms Race*. CYSARM'19.
London, United Kingdom: Association for Computing Machinery, 2019, pp. 3–10.
ISBN: 9781450368407. DOI: `10.1145/3338511.3357348`.
URL: `https://doi.org/10.1145/3338511.3357348`.

[51] Yoongu Kim et al. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors".
In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 361–372. ISSN: 0163-5964.
DOI: `10.1145/2678373.2665726`.
URL: `http://doi.acm.org/10.1145/2678373.2665726`.

[52] Paul Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *Advances in Cryptology — CRYPTO' 99*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397. ISBN: 978-3-540-48405-9.

[53] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. DOI: `10.1109/SP.2019.00002`.

[54] Jonas Krautter, Dennis R. E. Gnad, and Mehdi B. Tahoori. “Remote and Stealthy Fault Attacks on Virtualized FPGAs”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 1632–1637.

[55] Jonas Krautter, Dennis Gnad, and Mehdi Tahoori. “CPAmap: On the Complexity of Secure FPGA Virtualization, Multi-Tenancy, and Physical Design”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.3 (June 2020), pp. 121–146. DOI: `10.13154/tches.v2020.i3.121-146`. URL: `https://tches.iacr.org/index.php/TCHES/article/view/8585`.

[56] Jonas Krautter, Dennis R. E. Gnad, and Mehdi B. Tahoori. “FPGAhammer: Remote Voltage Fault Attacks on Shared FPGAs, suitable for DFA on AES”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)* 2018.3 (2018). ISSN: 2569-2925.

[57] Jonas Krautter, Dennis R. E. Gnad, and Mehdi B. Tahoori. “Mitigating Electrical-Level Attacks towards Secure Multi-Tenant FPGAs in the Cloud”. In: *ACM Trans. Reconfigurable Technol. Syst.* 12.3 (Aug. 2019). ISSN: 1936-7406. DOI: `10.1145/3328222`. URL: `https://doi.org/10.1145/3328222`.

[58] Tuan La et al. “Denial-of-Service on FPGA-based Cloud Infrastructure-Attack and Defense”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)* 2021.3 (2021), pp. 441–464.

[59] Vincent van der Leest et al. “Efficient Implementation of True Random Number Generator Based on SRAM PUFs”. In: *Cryptography and Security: From Theory to Applications*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 300–318. ISBN: 9783642283673.

[60] Charles R Lefurgy et al. “Active guardband management in Power7+ to save energy and maintain reliability”. In: *IEEE Micro* 33.4 (2013), pp. 35–45.

[61] 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*. ACSAC 2015. Los Angeles, CA, USA: ACM, 2015, pp. 461–470. ISBN: 978-1-4503-3682-6. DOI: `10.1145/2818000.2818030`. URL: `http://doi.acm.org/10.1145/2818000.2818030`.

[62] Linaro. *OP-TEE: Open Portable Trusted Execution Environment*. `https://github.com/OP-TEE`. 2022.

[63] Linaro Limited. *MbedTLS – Trusted Firmware*. https://www.trustedfirmware.org/projects/mbed-tls/. 2022.

[64] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[65] Moritz Lipp et al.
"Nethammer: Inducing Rowhammer Faults through Network Requests".
In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2020, pp. 710–719. DOI: `10.1109/EuroSPW51379.2020.00102`.

[66] Heiko Lohrke et al. "Key Extraction Using Thermal Laser Stimulation: A Case Study on Xilinx Ultrascale FPGAs". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (Aug. 2018), pp. 573–595.
DOI: `10.13154/tches.v2018.i3.573-595`.
URL: `https://tches.iacr.org/index.php/TCHES/article/view/7287`.

[67] Yukui Luo and Xiaolin Xu. "A Quantitative Defense Framework against Power Attacks on Multi-tenant FPGA".
In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2020, pp. 1–4.

[68] Dina Mahmoud and Mirjana Stojilović.
"Timing violation induced faults in multi-tenant FPGAs".
In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1745–1750.

[69] Dina G. Mahmoud et al. "FPGA-to-CPU Undervolting Attacks".
In: *2022 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2022, pp. 999–1004. DOI: `10.23919/DATE54114.2022.9774663`.

[70] Abhranil Maiti, Vikash Gunreddy, and Patrick Schaumont. *A Systematic Method to Evaluate and Compare the Performance of Physical Unclonable Functions*.
Cryptology ePrint Archive, Report 2011/657.
`https://eprint.iacr.org/2011/657`. 2011.

[71] K. Matas et al.
"Power-hammering through Glitch Amplification – Attacks and Mitigation".
In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020, pp. 65–69.
DOI: `10.1109/FCCM48280.2020.00018`.

[72] Dominik Merli et al. "Side-Channel Analysis of PUFs and Fuzzy Extractors".
In: *Trust and Trustworthy Computing*. Ed. by Jonathan M. McCune et al.
Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 33–47.
ISBN: 978-3-642-21599-5.

[73] Microsoft. *MS TPM 2.0 Reference Implementation*.
`https://github.com/microsoft/ms-tpm-20-ref`. 2022.

[74] Daniel Moghimi et al. "TPM-FAIL: TPM meets Timing and Lattice Attacks".
In: *29th USENIX Security Symposium (USENIX Security 20)*.
Boston, MA: USENIX Association, Aug. 2020. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi`.

[75] Gordon E. Moore. "Cramming more components onto integrated circuits".
In: *Electronics* 38.8 (Apr. 1965).

[76] Amir Moradi et al. "On the Vulnerability of FPGA Bitstream Encryption against Power Analysis Attacks: Extracting Keys from Xilinx Virtex-II FPGAs". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA: Association for Computing Machinery, 2011,

pp. 111–124. ISBN: 9781450309486. DOI: 10.1145/2046707.2046722.
URL: https://doi.org/10.1145/2046707.2046722.

[77] Kit Murdock et al.
"Plundervolt: How a Little Bit of Undervolting Can Create a Lot of Trouble".
In: *IEEE Security and Privacy* 18.5 (Sept. 2020), pp. 28–37. ISSN: 1540-7993.
DOI: 10.1109/MSEC.2020.2990495.
URL: https://doi.org/10.1109/MSEC.2020.2990495.

[78] Hassan Nassar et al. "LoopBreaker: Disabling Interconnects to Mitigate Voltage-Based Attacks in Multi-Tenant FPGAs".
In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*.
IEEE. 2021, pp. 1–9.

[79] Svetla Nikova, Christian Rechberger, and Vincent Rijmen.
"Threshold Implementations Against Side-Channel Attacks and Glitches".
In: *Information and Communications Security*.
Ed. by Peng Ning, Sihan Qing, and Ninghui Li.
Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 529–545.
ISBN: 978-3-540-49497-3.

[80] L. E. Olson et al. "Border control: Sandboxing accelerators". In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2015, pp. 470–481. DOI: 10.1145/2830772.2830819.

[81] Gilles Piret and Jean-Jacques Quisquater. "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad".
In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 77–88.
DOI: 10.1007/978-3-540-45238-6_7.

[82] George Provelengios, Daniel Holcomb, and Russell Tessier.
"Mitigating voltage attacks in multi-tenant FPGAs". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 14.2 (2021), pp. 1–24.

[83] Pengfei Qiu et al. "VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-Core Frequencies". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19.
London, United Kingdom: Association for Computing Machinery, 2019, pp. 195–209. ISBN: 9781450367479. DOI: 10.1145/3319535.3354201.
URL: https://doi.org/10.1145/3319535.3354201.

[84] Himanshu Raj et al. *fTPM: A Firmware-based TPM 2.0 Implementation*.
Tech. rep. MSR-TR-2015-84. Nov. 2015.
URL: https://www.microsoft.com/en-us/research/publication/ftpm-a-firmware-based-tpm-2-0-implementation/.

[85] Falk Schellenberg et al.
"Remote Inter-Chip Power Analysis Side-Channel Attacks at Board-Level".
In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.
2018, pp. 1–7. DOI: 10.1145/3240765.3240841.

[86] Geert-Jan Schrijen and Vincent van der Leest.
"Comparative Analysis of SRAM Memories Used as PUF Primitives". In:
DATE '12. Dresden, Germany: EDA Consortium, 2012, pp. 1319–1324.
ISBN: 9783981080186.

[88] William Stallings. *Data and Computer Communications*. 2007. URL: https://memberfiles.freewebs.com/00/88/103568800/documents/%5C%5CData.And.Computer.Communications.8e.WilliamStallings.pdf (visited on 05/28/2022).

[89] Takeshi Sugawara et al. "Oscillator without a combinatorial loop and its threat to FPGA in data centre". In: *Electronics Letters* 55.11 (2019), pp. 640–642.

[90] Haonan Sun et al. "eTPM: A trusted cloud platform enclave TPM scheme based on intel SGX technology". In: *Sensors* 18 (Nov. 2018), p. 3807. DOI: 10.3390/s18113807.

[91] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1057–1074. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[92] Shanquan Tian and Jakub Szefer. "Temporal Thermal Covert Channels in Cloud FPGAs". In: FPGA '19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 298–303. ISBN: 9781450361378. DOI: 10.1145/3289602.3293920. URL: https://doi.org/10.1145/3289602.3293920.

[93] Niek Timmers and Cristofaro Mune. "Escalating Privileges in Linux Using Voltage Fault Injection". In: *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 2017, pp. 1–8. DOI: 10.1109/FDTC.2017.16.

[94] J. Vliegen et al. "SACHa: Self-Attestation of Configurable Hardware". In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, pp. 746–751. DOI: 10.23919/DATE.2019.8714775.

[95] Jo Vliegen, Nele Mentens, and Ingrid Verbauwhede. "Secure, remote, dynamic reconfiguration of FPGAs". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7.4 (2014), pp. 1–19.

[96] Christian Werling and Robert Buhren. *Dissecting the AMD Platform Security Processor*. Chaos Computer Club e.V. https://doi.org/10.5446/43206 (Last accessed: 25 Oct 2022). 2019. DOI: 10.5446/43206. URL: https://doi.org/10.5446/43206.

[97] A. Wild and T. Güneysu. "Enabling SRAM-PUFs on Xilinx FPGAs". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–4.

[98] Xilinx. *Accelerating Cryptographic Performance on the Zynq UltraScale+MPSoC*. WP512 (v1.0). May 2019.

[99] Xilinx. *AR 72243 Zynq UltraScale+ MPSoC/RFSoC: UG1085 v1.9 - Incorrect statement on Encrypt Only Secure Boot*. https://www.xilinx.com/support/answers/72243.html. 2019.

[100] Xilinx. *AR 72654 Zynq UltraScale+ MPSoC/RFSoC: ACP Usage with XMPU / XPPU / TrustZone Isolation*. https://www.xilinx.com/support/answers/72654.html. 2019.

[101] Xilinx. *Developing Tamper-Resistant Designs with Zynq UltraScale+ Devices*. XAPP1323 (v1.1). Aug. 2018.

[102] Xilinx. *libdfx - Linux User Space Solution for FPGA Programming*. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/1959854475/libdfx+-+Linux+User+Space+Solution+for+FPGA+Programming. 2022.

[103] Xilinx. *Linux SHA Driver for Zynq Ultrascale+ MPSoC*. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841654/Linux+SHA+Driver+for+Zynq+Ultrascale+MPSoC. 2022.

[104] Xilinx. *Measured Boot of Zynq UltraScale+ devices*. XAPP1342 (v1.0). Apr. 2019.

[105] Xilinx. *Solution Zynq PL Programming With FPGA Manager*. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841645/Solution+Zynq+PL+Programming+With+FPGA+Manager. 2022.

[106] Xilinx. *Solution ZynqMP PL Programming*. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841847/Solution+ZynqMP+PL+Programming. 2020.

[107] Xilinx. *Zynq Ultrascale+ Device Technical Reference Manual*. v2.3. Sept. 2022.

[108] Xilinx. *Zynq-7000 SoC Technical Reference Manual*. v1.13. Apr. 2021.

[109] N. Zhang et al. "CaSE: Cache-Assisted Secure Execution on ARM Processors". In: *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 72–90. DOI: 10.1109/SP.2016.13.

[110] Ning Zhang et al. *TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices*. Cryptology ePrint Archive, Report 2016/980. https://eprint.iacr.org/2016/980. 2016.

[111] M. Zhao and G. E. Suh. "FPGA-Based Remote Power Side-Channel Attacks". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 229–244. DOI: 10.1109/SP.2018.00049.

[112] Shijun Zhao et al. "Providing Root of Trust for ARM TrustZone Using On-Chip SRAM". In: *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*. TrustED '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 25–36. ISBN: 9781450331494. DOI: 10.1145/2666141.2666145. URL: https://doi.org/10.1145/2666141.2666145.

[113] Loïc Zussa et al. "Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism". In: *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. 2013, pp. 110–115. DOI: 10.1109/IOLTS.2013.6604060.