Journal of Cryptographic Engineering (2022) 12:181–196
https://doi.org/10.1007/s13389-021-00273-8

REGULAR PAPER

Breaking TrustZone memory isolation and secure boot through malicious hardware on a modern FPGA-SoC

Mathieu Gross[1] · Nisha Jacob[2] · Andreas Zankl[2] · Georg Sigl[1,2]

Received: 3 December 2020 / Accepted: 30 August 2021 / Published online: 15 September 2021

Abstract
FPGA-SoCs are heterogeneous embedded computing platforms consisting of reconfigurable hardware and high-performance processing units. This combination offers flexibility and good performance for the design of embedded systems. However, allowing the sharing of resources between an FPGA and an embedded CPU enables possible attacks from one system on the other. This work demonstrates that a malicious hardware block contained inside the reconfigurable logic can manipulate the memory and peripherals of the CPU. Previous works have already considered direct memory access attacks from malicious logic on platforms containing no memory isolation mechanism. In this work, such attacks are investigated on a modern platform which contains state-of-the-art memory and peripherals isolation mechanisms. We demonstrate two attacks capable of compromising a Trusted Execution Environment based on ARM TrustZone and show a new attack capable of bypassing the secure boot configuration set by a device owner via the manipulation of Battery-Backed RAM and eFuses from malicious logic.

Keywords FPGA-SoCs · Memory and peripherals isolation · Hardware trojan · DMA attack · Trusted execution environment · Secure boot

1 Introduction

FPGAs are popular platforms used for the acceleration of computations. Due to their good computational power together with a low power consumption, these platforms are widely used in the Cloud as an alternative to GPU acceleration especially in machine learning applications. The FPGA computing platform is also popular in the embedded world, where system-on-chips (SoCs) with high-performance processing units are integrated together with an FPGA (FPGA-SoCs). Besides enhancing performance, the sharing of CPU resources together with an FPGA can also lead to security threats. Academia has demonstrated powerful attacks from the FPGA to the CPU in the FPGA-Cloud [8,18,23,29] and the FPGA-SoC [5,15,19] computation paradigms. For these kind of attacks, it is generally assumed that a third-party intellectual property (IP) contained inside the reconfigurable logic has a malicious hidden functionality or Hardware Trojan (HT). Similarly, in this work, we consider the presence of a HT inside a third-party IP used inside an FPGA-SoC.

Previous works which considered the FPGA-SoC scenario have shown that a HT can compromise the software running on the embedded CPU of an FPGA-SoC via DDR memory manipulation [5,15,19]. This type of attack is similar to a direct memory access (DMA) attack in which an external I/O interface [3,4,14,22] alters the software running on a host PC. DMA attacks can be prevented via the use of an input–output memory management unit (IOMMU). This component is responsible for the memory management of peripherals and is used to prevent unauthorized memory access from a peripheral. Modern FPGA-SoC architectures such as the Xilinx Zynq UltraScale+ (ZU+) or the Intel Stratix 10 integrate a

✉ Mathieu Gross
mathieu.gross@tum.de

Nisha Jacob
nisha.jacob@aisec.fraunhofer.de

Andreas Zankl
andreas.zankl@aisec.fraunhofer.de

Georg Sigl
sigl@tum.de

[1] Department of Electrical and Computer Engineering, Chair of Security in Information Technology, Technical University of Munich, Theresienstr. 90, 80333 Munich, Germany

[2] Fraunhofer Institute for Applied and Integrated Security AISEC, Lichtenbergstr. 11, Garching, 85748 Munich, Germany

system memory management unit (SMMU), which is the equivalent of an IOMMU in ARM terminology. The SMMU makes the attacks described in [5,15,19] more difficult. Furthermore, both architectures offer additional mechanisms for isolation such as a memory protection unit (MPU), a peripheral protection unit (PPU) and ARM TrustZone technology.

Despite the presence of these isolation mechanisms, we have shown in our previous work [10] that DMA attacks from a HT are possible on the ZU+. We found out that a hardware accelerator connected to the accelerator coherency port (ACP) is not affected by the SMMU and that the Xilinx memory protection units (XMPUs) fail in isolating the memory of the CPU from the ACP. This isolation issue enables a HT hidden inside a third party IP to compromise the software running on the embedded CPU of an FPGA-SoC. As a concrete example, we demonstrated two ways of bypassing security guarantees of a trusted execution environment (TEE) via memory manipulation attacks.

This work extends our research performed in [10]. We found out that the problem observed with the XMPUs also affects the Xilinx peripheral protection unit (XPPU). The isolation issue enables a HT contained inside a master using the ACP to access CPU peripherals which are protected by the XPPU. As a concrete example, we demonstrate a scenario in which a HT programs an AES key in the Battery-Backed RAM (BBRAM) and an RSA public key hash in the eFuses of a device. This enables an attacker to bypass the secure boot configuration set by the device owner and to start her own authenticated image on the attacked device.

1.1 Our contribution

Similar to the works of [5,15,19], we exploit a security vulnerability of FPGA-SoC architectures, which allows a HT to perform DMA attacks on the CPU subsystem. This work, however, considers the ZU+ architecture, which contains more protection mechanisms than the previous Zynq-7000 architecture.

In our original work [10], we show the feasibility of performing powerful DMA attacks on ARM TrustZone, despite the protection provided by this technology against DMA attacks. This work reveals that the memory isolation issue described in our original work is extended to the peripherals. We demonstrate 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 [15]. This work considers a similar attack on the ZU+ platform and uses a different approach as the one proposed in [15].

1.2 Structure of this work

The remainder of this work is organized as follows: Section 2 provides the background related to the ZU+ FPGA-SoC and ARM TrustZone. Section 3 describes the accelerator coherency port (ACP) and explains a security vulnerability in the mechanism used to isolate CPU private memory/peripherals from a tightly coupled ACP master. Section 4 demonstrates two concrete attack examples on a TrustZone based TEE. Section 5 demonstrates an attack which compromises of the hardware root of trust secure boot mode of the ZU+. Section 6 discusses possible mitigations against the attacks presented in this work and their portability to other FPGA-SoCs. Section 7 contains the conclusion of this work.

2 Background

This section introduces the necessary background required for understanding the attack methodology and the proofs of concept presented inside this work. The first part introduces the ZU+ architecture, some of its protection mechanisms and its secure boot. Subsequently, ARM TrustZone technology and the concept of trusted execution environment (TEE) are described.

2.1 Xilinx ZU+ architecture

This work uses an FPGA-SoC based on the Xilinx ZU+ EG MPSoC. This architecture consists of an quad-core ARM Cortex-A53 as application processor unit (APU), an ARM Mali-400 Graphic Processor Unit (GPU), a dual-core ARM Cortex-R5 real-time processing unit (RPU) and an FPGA. The MPSoC contains 256 kB on-chip memory (OCM) that can be used for storing sensitive data or code and 2 GB external DDR memory. The memory system is accessible through the ARM AMBA AXI4 bus system. The MPSoC contains state-of-the-art peripherals for external communication.

The interaction between the FPGA fabric and the processing system (PS) is implemented via interrupts, GPIO signals and AXI slave and master interfaces. This work particularly relies on the use of the ACP, a port typically used for the connection of a tightly coupled I/O coherent hardware accelerator to the Cortex-A53 memory subsystem. Further details about the ACP are provided in Sect. 3.1.

ZU+ contains a set of security mechanisms that enable secure boot, run-time protection, and secure key storage/generation. Since this work focuses on memory and peripheral manipulation via malicious logic, a description of the mechanisms used for isolation of those assets is provided in Sect. 2.2.

2.2 Memory and peripherals protection schemes in ZU+

This section presents the primitives that can be used to achieve memory and peripheral isolation inside the ZU+ MPSoC. Those consist of a system memory management unit (SMMU), Xilinx memory protection units (XMPUs) and a Xilinx peripheral protection unit (XPPU).

System Memory Management Unit (SMMU):

The SMMU is complementary to a traditional MMU. It provides a two-stage address translation for I/O devices. The first stage is relevant for systems running multiple OSs and is managed by a hypervisor. In this stage, virtual addresses are translated into intermediate physical addresses. The second stage tackles address translation for the applications running inside the OS. This is done by turning intermediate physical addresses into physical addresses.

Another important service provided by an SMMU is memory isolation. This property is achieved by restricting the reachable address space for I/O devices and hence protects the OS against DMA attacks.

Xilinx Memory Protection Units (XMPUs): Eight XMPUs work in collaboration with the SMMU to offer memory protection (DDR, OCM) via isolation. These units check explicitly if a master is allowed to access a given address via the definition of memory regions. A memory region consists of an address range and a list of masters which are allowed to access this region. Additionally, TrustZone support enables the placement of memory regions in the Secure World or Normal World such that only allowed secure masters can access a memory region tagged as secure. In case of an access violation, the XMPUs can notify the master via an interrupt or the AXI response signals (RRESP/BRESP).

Xilinx Peripheral Protection Unit (XPPU): XPPU is another important asset aiming at protecting peripherals and configuration registers. The usage of the XPPU is similar to the XMPUs, except that the concept of memory regions are replaced by apertures. More precisely, an aperture is a set of register addresses and the aperture permission list identifies the masters that can read/write to those addresses. Like the XMPUs, the XPPU supports TrustZone technology and thus enables a partitioning of the registers between the Normal World and the Secure World. An access violation is detected when a master attempts to access a register that it is not allowed to, or if a secure register is accessed via a non-secure request.

2.3 Secure boot on the ZU+

Secure boot is a crucial security feature which guarantees the integrity and authenticity of the software loaded during the boot process. On the ZU+, two secure boot modes are available: hardware root of trust and encrypt only. Recently, F-Secure [7] has shown that the encrypt only secure boot is vulnerable to boot header manipulation attacks. For the rest of this work, we consider the hardware root of trust secure boot. This boot scheme relies on an RSA authentication of the boot image and a comparison of the public authentication parameters with a value stored inside the eFuses. The description of the secure boot process used in this work is shown in Fig. 1.

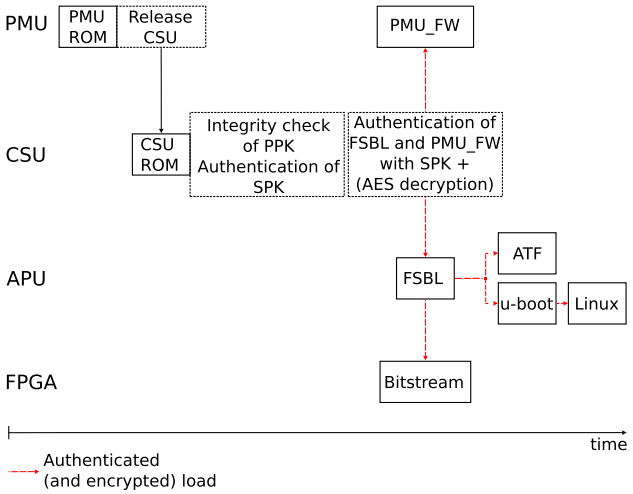

Fig. 1 Hardware root of trust secure boot on the ZU+

Upon startup of the ZU+, a hardware state machine performs some verification tests and compares the SHA3/384 digest of the PMU ROM with a value stored inside the device. If the two values match, the PMU ROM gets executed. This code is responsible for performing early device initialization and comparing a SHA3/384 digest of the CSU ROM with a golden copy stored inside the device. If the two values are equal, the PMU releases control to the CSU.

The CSU is then loading the RSA Primary Public Key (PPK) into OCM and compares its hash value with a value programmed inside the eFuses. Two PPK hash values can be programmed inside the eFuses, the boot image header specifies which value should be used. If the computed PPK hash value corresponds to the value programmed inside the eFuses, the PPK stored in OCM can be used for authenticating the Secondary Public Key (SPK). The SPK, SPK_{ID} and SPK signature are contained inside the boot image. The next step consists of checking if the SPK_{ID} matches the value programmed inside the eFuses and if the computed SPK signature corresponds to the value contained inside the boot image. If that is the case, the SPK can be used for authenticating the first-stage boot loader (FSBL) and the platform management unit firmware (PMU_FW). Optionally, both components can additionally be encrypted with AES-GCM 256. The AES key used for decryption is the device key; this

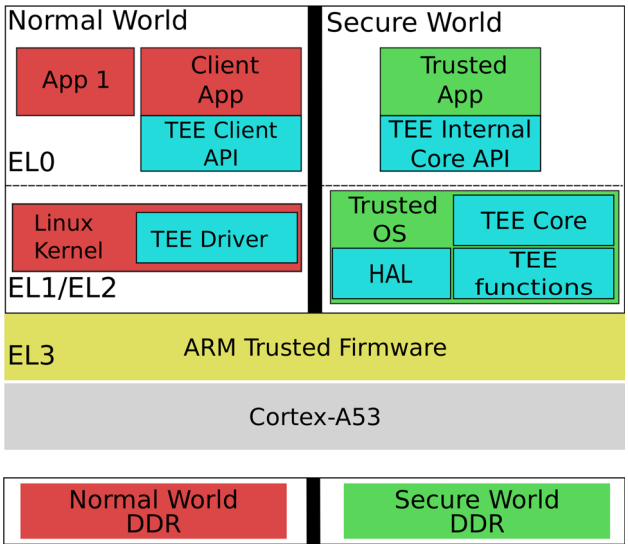

Fig. 2 TEE platform architecture

key is stored in the PUF, eFuses or BBRAM depending on what is specified inside the boot header.

After the release of the APU, the FSBL uses the SPK to authenticate the subsequent partitions contained inside the boot image, namely the bitstream, ARM trusted firmware (ATF) and u-boot. As before, these partitions can also be optionally encrypted with the AES device key.

Once the FSBL is finished loading the partitions, the control is released to u-boot. U-boot finishes the boot chain by authenticating the operating system with the SPK and optionally decrypts it with the device key.

2.4 ARM TrustZone-based trusted execution environment

ARM TrustZone [2] is a technology designed to provide hardware isolation for trusted software execution. It consists of a set of security extensions added to many ARMv7-A and ARMv8-A Cortex-A processors. Recently, TrustZone support has also been added for ARMv8-M processors; however, this variant is out of the scope of this work.

A Cortex-A processor supporting ARM TrustZone has its private resources (registers, caches, memory) partitioned between the Normal World (NoW) and the Secure World (SeW). The security configuration register (SCR) indicates in which world the CPU is currently running. This is also reflected on the ARM AMBA AXI bus via the ARPROT[0]/AWPROT[0] bit.

ARM trusted firmware (ATF) is the reference implementation of the SeW software and is executing at Exception Level 3 (EL3). ATF contains a secure monitor which handles the context switching between the two worlds upon receiving a secure monitor call (SMC).

A further system-wide isolation is achieved by defining SeW and NoW interrupts sources (FIQ for the SeW and IRQ for the NoW). Interrupts triggered by a FIQ source can only be handled in the SeW and similarly IRQ interrupts are handled in the NoW. Therefore, a world switch may be necessary before handling an interrupt. Finally, ARM TrustZone also enables the mapping (static or dynamic) of I/O devices such as the DMA peripheral to one world.

TrustZone support inside the FPGA fabric: TrustZone technology is extensible to the FPGA fabric through the use of the ARPROT[0]/AWPROT[0] bit and the AXI interconnect. A master inside the FPGA fabric can dynamically configure the security of a read/write transaction via the ARPROT[0]/AWPROT[0] bit. A secure transaction is indicated with the value 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 master generating secure transactions. A non-secure slave on the other hand is accessible by both secure or non-secure masters.

Trusted Execution Environment (TEE): ARM TrustZone technology enables the deployment of a TEE [9] inside the FPGA-SoC. The software architecture of such a system is shown in Fig. 2. The TEE is running in parallel to the rich execution environment (REE). The TEE enables the execution of security critical software in an isolated execution environment, which is not directly accessible to the Rich OS. In contrary to the Rich OS, only authenticated and unaltered binaries run inside the TEE. The goal is to guarantee a secure execution of critical software even if the Rich OS is compromised. To interact with the TEE, the REE kernel is enhanced with a TEE Driver. NoW client applications (CAs) use the global platform TEE client 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. Trustlets use the global platform TEE internal core API to access to the EL1 trusted operating systems (Trusted OS) functions such as cryptography and secure storage.

3 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 pre-

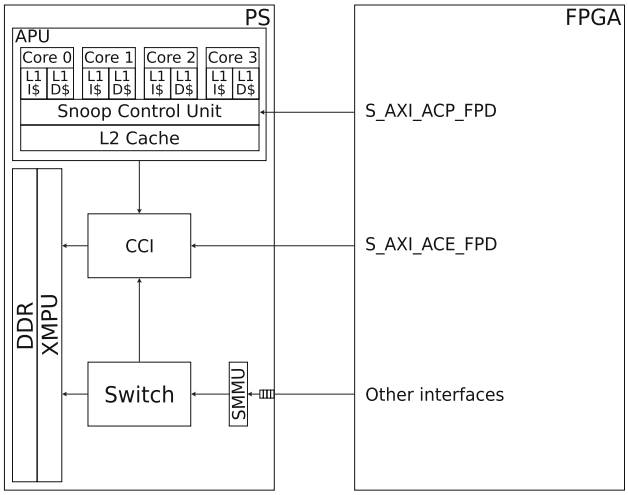

Fig. 3 Accelerator coherency port memory interfacing

vent an ACP master to access processor private memory and peripherals are discussed.

3.1 ACP slave interface on the Cortex-A53

Among all the available FPGA fabric memory interfaces, the ACP is recommended for applications where a hardware accelerator is tightly coupled with the application processor unit (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 Fig. 3). 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 cache coherent interconnect (CCI) enables a master in the FPGA fabric to have a faster 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 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.

3.2 Processor and ACP master memory isolation

The ACP is typically used to connect a tightly coupled 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 shown in Fig. 3, the ACP is not connected to the SMMU. Alternatively, the XMPUs should be suitable 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 Sect. 2.2, 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).
- The security property (secure/non-secure) of the region (R_SECURE).
- The region master ID value (R_MID_V) and the region master ID mask (R_MID_MASK).

An incoming read or write request on an AXI port is checked against the conditions listed in Eq. 1 for each memory region (R_i) defined in the XMPUs' configuration registers.

$$\begin{cases} R_i\_START \leq AXI\_ADDR \leq R_i\_ENDAXI\_MID\_V \\ AXI\_MID\_MASK == R_i\_MID\_V \& R_i\_MID\_MASK \\ AXI\_ARPROT[0]/AWPROT[0] == R_i\_SECURE \end{cases} \tag{1}$$

Only AXI transactions satisfying Eq. 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 [26] provides the necessary information regarding APU master ID. APU transactions have their master ID defined according to Eq. 2.

$$APU\_MID[9:0] = 0010||AXI\_MID[5:0] \tag{2}$$

Xilinx does not provide further information regarding the ACP master ID. An inspection of the ARM Cortex-A53 Technical Reference Manual [1] reveals that the six lowest bits of the AXI read/write transaction ID can differentiate APU and ACP transactions. The encoding for the six lowest bits of the read/write ID is given in Table 1

From Eq. 2 and Table 1, it is clear that APU transactions coming from one of the four CPU cores and ACP transactions can be distinguished by their ID.

Table 1 Read/write transactions ID encoding for the ARM Cortex-A53

AXI_MID	Issuing capability per ID	Transaction type
0b0000nn[1]	4	Core nn exclusiveread/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

To find out if this is indeed the case, we used Vivado 2018.2 to generate a design targeting the Xilinx ZU+ MPSoC ZCU102 Evaluation Kit. This design contains a hardware accelerator accessing memory via the ACP and the PS. Configuring the XMPUs manually by writing specific values inside registers can be quite a difficult and error prone task. This requires the adaptation of the bootcode (psu_init.c) where the memory regions for the XMPUs are defined. A manual configuration of the isolation might also lead to a too strong isolation, which prevents the system from working correctly. For those reasons, Xilinx recommends the use of the Vivado isolation configuration (VIC) [27], a graphical tool that can be used to achieve peripherals and memory isolation inside the ZU+. We used this tool to restrict the memory access of the hardware accelerator. The generated Hardware Description File (hdf) and bitstream are used to generate the bootcode of the Cortex-A53 and thereby configure the XMPUs. Once configured, the XMPUs' registers are locked, such that malicious software cannot alter the isolation configuration.

Despite a correct definition of the memory region inside the VIC, we discovered that the ACP master is able to access APU memory addresses which it is not supposed to. The unauthorized access is possible whether the corresponding data is present in the APU caches or not. An inspection of Fig. 3 reveals that the XMPUs are not located at the L2 cache controller but at the DDR controller instead. Therefore, the access to cached private data could be expected if the user carefully inspects the ZU+ architecture. The access to uncached APU private memory is however something which should be filtered by the XMPUs. A closer look at the generated bootcode reveals that the master ID, start and end address of the APU private regions are properly written in the XMPUs' registers. However, as explained earlier in this section, the XMPUs' transactions are also filtering master IDs with masks values. The bootcode reveals the IDs and masks values generated after using the VIC. The ID 128 and the mask 960 are used for defining memory regions for the Cortex-A53 core 0. An incoming ACP transaction on the other hand can have the IDs (144, 148, 152 or 160). Injecting these possible ACP ID values with the mask value 960 in Eq. 1 lead to the result 128. Therefore, the tuple (ID, mask) defined in software for APU core 0 does not allow the XMPUs to distinguish APU and ACP transactions, and thus enables illegal ACP memory access to APU private memory whether is located inside the L2 cache or not.

The mask configuration issue leads to severe memory isolation problems. A HT contained inside a third-party IP interfacing memory via the ACP can access the whole processor memory with physical addresses. This opens the door to DMA attacks performed from a HT contained in a hardware accelerator. To demonstrate the impact of this threat, we describe the implementation of two DMA attacks on a TrustZone-based TEE in Sect. 4.

3.3 Processor and ACP master peripheral isolation

As mentioned in Sect. 3.1, ACP is an interface which 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 Sect. 3.2, 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 Sect. 2.2, the XPPU is operating in a similar way as the XMPUs, except that the notion of memory regions, are replaced with apertures. An aperture is a range of registers 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 Eq. 3. The first step consists of identifying the aperture corresponding to an incoming AXI transaction ($\text{APPER}_{\text{inc}}$). Once

it is found that the XPPU performs a master ID filtering operation similar to the one of the XMPUs (see Sect. 3.2). 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 of verifying that the security of the transaction matches the one of the aperture.

If any of these three checks fail, the peripheral access is denied, which results in a rejection of the transaction.

$$\begin{cases} (AXI\_MID\_V \& AXI\_MID\_MASK) \\ \quad \in APPER_{inc}\_AUTHORIZED\_MASTERS \\ AXI\_ARPROT[0]/AWPROT[0] == APPER_{inc}\_SECURE \end{cases} \tag{3}$$

Similar to the observations made in Sect. 3.2, 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 Sect. 3.2 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 and mask 960. 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 Eq. 2, Table 1 and the discussion of Sect. 3.2).

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. To illustrate the consequences of this problem, we have implemented an attack in which an attacker can break secure boot and take control of a ZU+ device by interfering with eFuses and BBRAM in Sect. 5.

4 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 PoC demonstrates how the HT can affect the signature verification of trustlets before their execution inside OP-TEE [20]. 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.

4.1 System description

Architectural description: This work uses a Xilinx ZU+ MPSoC ZCU102 Evaluation Kit. The system considered in this work is presented in Fig. 4. It consists of the processing system (PS) and a third-party IP contained in the reconfigurable logic (IP 1). An embedded Linux solution (Rich OS) is running on the Cortex-A53. Furthermore, an ARM TrustZone-based TEE is executing in parallel to the Rich OS. The TEE consists of ARM trusted firmware (ATF) running inside the OCM and OP-TEE running inside the DDR memory. OCM and DDR are partitioned in the Normal World (NoW) and the Secure World (SeW). Since IP 1 is obtained from a third party, it cannot be fully trusted. Unfortunately, a hidden malicious functionality (Hardware Trojan) 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 Fig. 4). 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 [27]. The partitioning of the DDR memory and the OCM is shown in Table 2. As shown in Fig. 4, 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 in a wide range of applications such as video processing, machine learning, or cryptography.

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 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.

4.2 PoC1: Compromising the signature verification of trustlets done by OP-TEE

Trustlets are binaries running inside the SeW at Exception Level (EL 0). 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 RootFS (see Fig. 5). The private key used for signing the

Table 2 XMPUs configuration

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

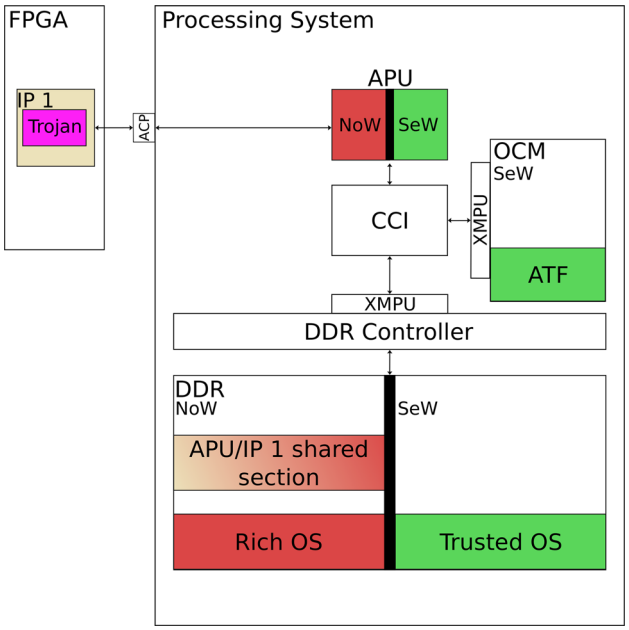

Fig. 4 System block design

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 DMA attack (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 Sect. 4.1 and the XMPUs configuration in Table 2, a DMA attack is possible because of the memory isolation issue described in Sect. 3.2. To extend the isolation issue to the TrustZone, the HT must access SeW APU memory. As explained in Sect. 3.2, XMPUs' registers are locked once configured. 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.

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 DMA attack on the signature verification function 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 signatures verification 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

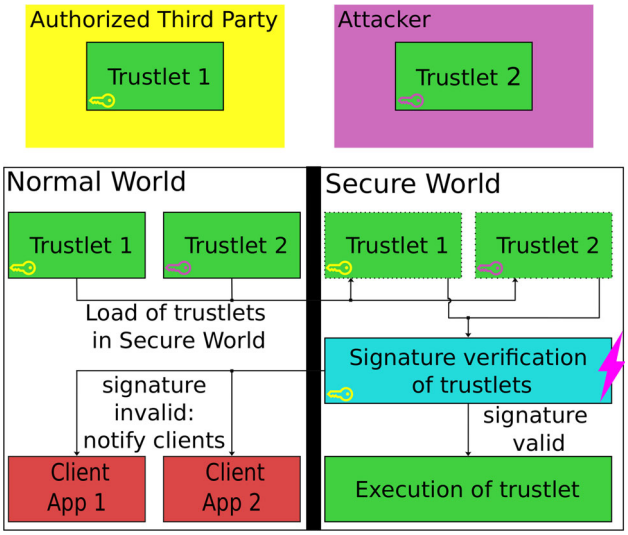

Fig. 5 Attack on the trustlets signature verification

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 its own private key.

4.3 PoC2: 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 of 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-128-GCM. 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.

In addition to a trustlet specific secure key storage, OP-TEE provides isolated AES-128-GCM 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

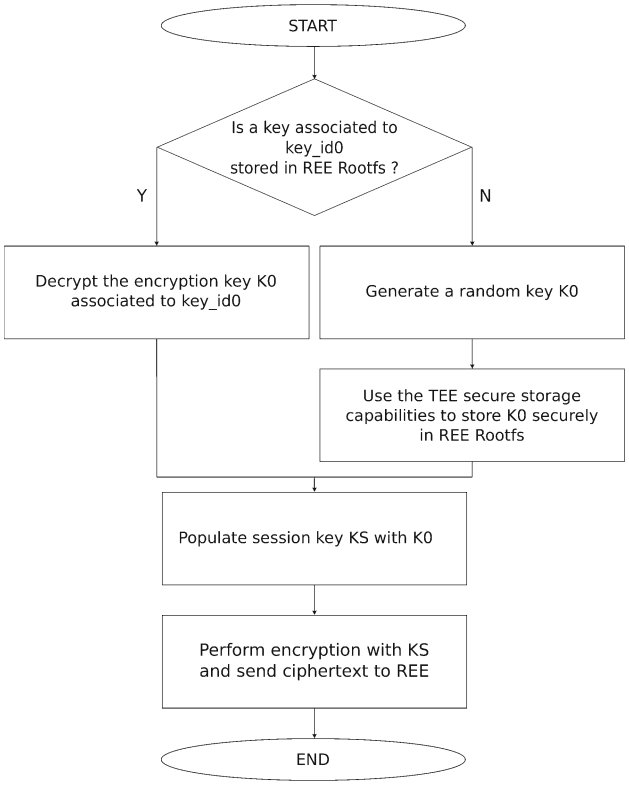

Fig. 6 *Trustlet_0* description

the AES key schedule and stores it in a contiguous memory buffer 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 Fig. 6. If a key (*key0*) associated with key_id0 exists in an encrypted form inside the Rich OS RootFS (see Appendix A), it is loaded from the Rich OS RootFS and decrypted inside *trustlet_0*. Once decrypted, *K0* is further 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.

DMA attack 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 of finding the AES key necessary for decrypting the file with the help of a SeW memory dump. The system setup described in Sect. 4.1 and the XMPUs' configuration described in Table 2 is assumed to be run on the FPGA-SoC. The first step of the attack consists of dumping the whole

```
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 fffffb0 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
```

Fig. 7 Little endian representation of the AES key schedule found in a Secure World memory dump

SeW memory (8 MB) via the HT contained inside IP 1 (see Fig. 4). This is done by generating secure read transactions (ARPROT=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 [11], 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.

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 ← 0
6:     while key_found ≠ 1 OR word_iterator ≠ endOfFile
   do
7:         key_cand ← 16 Bytes following word_iterator
8:         key_schedule_cand = KeySchedule(key_cand)
9:         if key_schedule_cand ⊂ memory_dump then
10:            key_found ← 1
11:        end if
12:        word_iterator++
13:    end while
14: end procedure
```

We verified the success of our approach for different AES keys. Figure 7 corresponds to the portion of the memory dump containing the key schedule (in little endian representation) associated with 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 [16]. We assume that this parameter is known to the attacker.

5 Compromising secure boot and secure device updates on the ZU+ via the ACP

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 (Battery-Backed RAM) BBRAM and eFuses via the ACP. After programming an RSA 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+.

5.1 System description

The system architecture is shown in Fig. 8. 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 Fig. 8. It consists of the processing system (PS) and a third-party IP located inside the FPGA fabric (IP 1) which is connected to the PS via the ACP. Similarly to Sect. 4.1, 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 first-stage boot loader (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. 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 5.2 and 5.3 explains 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 Sect. 5.4.

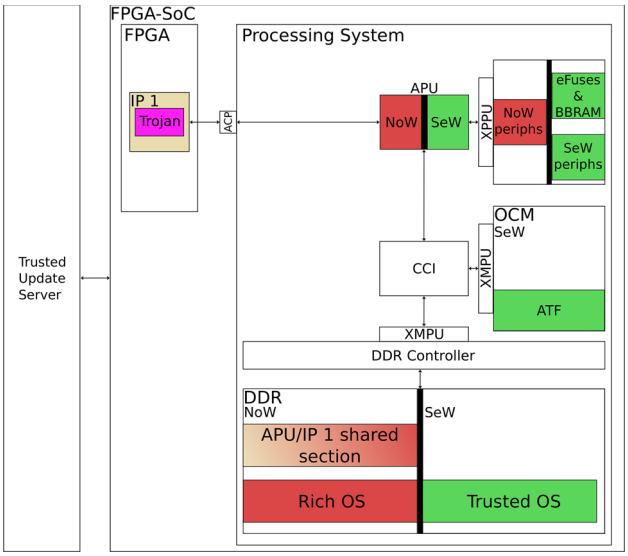

Fig. 8 Secure boot with an image obtained from a third party

5.2 Programming of an RSA public key hash into the eFuses from the ACP

As explained in Sect. 2.3, 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, this 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 Sect. 3.3, 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 Sect. 5.4, 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.

5.3 Programming of an AES key into BBRAM from 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

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 eFuses 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: $efuse\_pgm\_addr \leftarrow (row(RSA\_PPK\_HASH[i]), column(RSA\_PPK\_HASH[i]))$
8: **end if**
9: **end for**
10: $efuse\_cfg\_pgm\_en \leftarrow 0$ ▷ Disable programming mode
11: $efuse\_wr\_lock \leftarrow 0$ ▷ Lock the eFuses controller
12: **end procedure**

snippets which enables programming of BBRAM from a processor (APU or RPU). The BBRAM registers are accessible from the ACP. To verify the possibility of programming a BBRAM key from the ACP, we performed the steps mentioned in Algorithm 3.

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 <– 0x757BDF0D ▷ Put BBRAM in programming mode
3: $bbram\_\{0..8\}\_reg \leftarrow AES\_KEY$
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**

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 the BBRAM without preventing the device from booting. In Sect. 5.4, 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.

5.4 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 Sect. 5.2) and optionally to program an AES key into BBRAM (see Sect. 5.3). 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 via the HT contained inside IP 1 (see Fig. 8). 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 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 the 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 the BBRAM. Again, the device owner is expecting the device key to be stored in the AES eFuses; however, the compromised boot image has specified the BBRAM as device key source.

6 Mitigations and portability of the attacks on other FPGA-SoCs platforms

In this section, we discuss possible countermeasures against the attack vectors described in Sects. 3.2 and 3.3 and the PoCs described in Sects. 4 and 5. We also evaluate if the attacks presented in this work might be applicable on other FPGA-SoC platforms.

6.1 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 ✓DMA/× secure boot indicates that a preventive technique effectively mitigates the DMA attacks described in Sect. 4 but not the attack against the hardware root of trust secure boot described in Sect. 5.

Manual modification of the XMPUs'/XPPU's configuration (⋆DMA/✓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 1, the 5th 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. However, we observed that changing the mask value in the XMPUs registers prevents the system from booting. This means that there is at least one incoming APU transaction for which the second condition in Eq. 1 is not met. Our hypothesis is that the "Core nn read/write" transaction ID is implemented with the 5th 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 of 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 1), we chose to modify the XMPU configuration according to Table 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 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 shown in Fig. 3, the XMPU cannot prevent an accelerator from accessing data located inside the L2 cache. Therefore, in order to protect the TEE from the memory manipulation attacks 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 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.

Use of another FPGA fabric to PS memory interface (✓DMA/✓$secureboot$): 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 (✓DMA/✓$secureboot$): In contrast to most of the PS slaves ports (see Fig. 3), ACP transactions are not filtered by an SMMU. As an alternative, Olson et al. [21] propose Border Control, a mechanism that can substitute an SMMU by sandboxing accelerators and protecting the memory from a

Table 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 0 (160, 1008)
Core 2 (197, 1023)	Core 1 (80, 1022)
Core 3 (98, 1023)	Core 2 (197, 1023)
	Core 3 (98, 1023)

Table 4 XPPU configuration (ID, mask) for the APU profiles

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)

malicious or misbehaving accelerator. Similarly, a special AXI wrapper as used in [15] 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 ($\star$DMA/$\star secureboot$): A hardware accelerator can arbitrarily configure the security of a transaction via the ARPROT[0]/AWPROT[0] bit. A firewall associated with 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 ($\times$DMA/$\times$ $secureboot$): 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 [17] and Cold Boot [11] 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 Sect. 3.2.

Execute sensitive code in caches instead of DDR ($\times$DMA/$\times$ $secureboot$): 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 DMA attacks, the Sentry [6] and CaSE frameworks [28] enable the execution of a critical application inside the cache only via the ARM lockdown features. The protection against a DMA 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 ACP DMA adversary 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 ($\star$DMA/$\times$ $secureboot$): This work uses the Rich OS RootFS secure storage features provided by OP-TEE (see Appendix A). Alternative possibilities on a ZU+ FPGA-SoC are the BBRAM or eFuses. However, none of these features can help to protect against the PoC described in Sect. 4.3 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 and a software solution might be preferred for a faster deployment.

Program both set of eFuses used for storing the RSA PPKs hash ($\times$DMA/$\checkmark$ $secureboot$): The attack presented in Sect. 5.4 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 [26]. 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 ($\times$DMA/$\star secureboot$): 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 [24]. 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 [7].

6.2 Attack portability on other platforms

The PoC described in Sect. 4 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 [13] 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 DMA attacks from malicious logic consists of 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 DMA attack scenarios during the boot of the processor. Concerning authentication, Stratix 10 relies on ECDSA with a root public key (equivalent of the PPK on the ZU+) hash stored in the eFuses [12]. Only one root public key can be programmed into a Stratix 10 device and root key cancellation is not possible. Therefore, the attack performed in Section 5.4 seems not to be applicable on this architecture.

7 Conclusion

This work shows two approaches for compromising an FPGA-SoC via malicious hardware. The first one consists of manipulating memory in order to bypass some security mechanisms of a TEE. In contrast to previous works [5,15,19], 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 isolation mechanisms, we show that malicious hardware can still compromise memory via the accelerator coherency port (ACP). This interface is usually considered for scenarios where a hardware accelerator requires fast and cache coherent memory access.

The second approach consists of 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 [21], or to use a wrapper as done in [15].

8 Responsible disclosure

Xilinx has been informed about the XMPU vulnerability we discovered in July 2019 and responded via the Answer Record 72654 [25]. 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 [27]. 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 parallel to this submission, 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 26^{th} 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.

Acknowledgements This project was supported by the Bavarian Research Foundation as a part of the Bavarian project FutureIOT grant number AZ-1301-17.

Funding Open Access funding enabled and organized by Projekt DEAL.

Software-assisted secure data storage in OP-TEE

The software-assisted secure data storage functionality implemented inside OP-TEE follows the recommendations specified in the TEE Internal Core API specification. This feature allows to store key material and general-purpose data with a confidentiality and integrity guarantee.

The Rich OS RootFS secure storage feature relies on the use of several encryption keys. The secure storage key (SSK) 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 Eq. 4.

$$SSK = HMAC_{SHA256}(HUK, ChipID||''staticstring'') \tag{4}$$

The trusted application storage keys (TSKs) are 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 Eq. 5.

$$TSK = HMAC_{SHA256}(SSK, TA\_UUID) \tag{5}$$

Each generation of a TEE file inside a TA comes with the generation of a new FEK. This key is generated by a pseudo-random-number generator and is further used to encrypt the meta data of the file and the data blocks composing it. Metadata encryption results in the creation of the MetaData Field as explained in Eq. 6:

$$\begin{cases} FEK_{crypt} = AES-ECB(FEK, TSK) \\ (MetaData_{crypt},\ TAG) = AES-GCM(MetaData,\ IV,\ FEK_{crypt}) \\ MetaData\ Field = (FEK_{crypt}\ ||\ IV\ ||\ TAG\ ||\ MetaData_{crypt}) \end{cases} \tag{6}$$

Similarly, the encryption of a data block results in the creation of a Data Block Field as defined in Eq. 7:

$$\begin{cases} (DataBlock_{crypt}\ ||\ TAG) = AES-GCM(DataBlock, IV, FEK) \\ DataBlock\ Field = ((DataBlock_{crypt}\ ||\ TAG\ ||\ IV) \end{cases} \tag{7}$$

As explained in Eqs. 6 and 7, 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.

The encrypted file is stored inside a hash tree, where the Hash Tree Header contains the MetaData Field from Eq. 6 and where each Node contains the TAG and initialization vector (IV) of a Data Block Field. A secure data file consists of the encrypted data blocks and the generated Hash Tree. This file (and its backup) is stored in the Rich OS RootFS under /data/tee.

References

1. ARM: ARM Cortex-A53 MPCore Processor Technical Reference Manual. Revision r0p2. (2014)
2. ARM: ARM Security Technology - Build a Secure System using TrustZone Technology. Issue D.c. (2016)
3. Aumaitre, D., Devine C.: Subverting windows 7 x64 kernel with dma attacks. In: HITBSecConf Amsterdam (2010)
4. Becher. M., Dornseif, M., Klein, C.: Firewire: all your memory are belong to us. (2005) https://cansecwest.com/core05/2005-firewire-cansecwest.pdf
5. Chaudhuri, S.: A security vulnerability analysis of socfpga architectures. In: Proceedings of the 55th Annual Design Automation Conference, ACM, New York, NY, USA, DAC '18, pp 139:1–139:6, https://doi.org/10.1145/3195970.3195979 (2018)
6. Colp, P., Zhang, J., Gleeson, J., Suneja, S., de Lara, E., Raj, H., Saroiu, S., Wolman, A.: 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, ACM, New York, NY, USA, ASPLOS '15, pp 177–189, (2015) https://doi.org/10.1145/2694344.2694380
7. F-Secure.: Security advisory: Xilinx zu+ encrypt only secure boot bypass. (2019) https://github.com/f-secure-foundry/advisories/blob/master/Security_Advisory-Ref_FSC-HWSEC-VR2019-0001-Xilinx_ZU+-Encrypt_Only_Secure_Boot_bypass.txt
8. Glamočanin, O., Coulon, L., Regazzoni, F., Stojilović, M.: Are cloud fpgas really vulnerable to power analysis attacks? In: 2020 Design, Automation Test in Europe Conference Exhibition (DATE), pp 1007–1010, https://doi.org/10.23919/DATE48585.2020.9116481 (2020)
9. Global Platform.: Introduction to trusted execution environments (2018)
10. Gross, M., Jacob, N., Zankl, A., Sigl, G.: 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, Association for Computing Machinery, New York, NY, USA, ASHES'19, p 3–12, (2019) https://doi.org/10.1145/3338508.3359568
11. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest we remember: cold-boot attacks on encryption keys. Commun. ACM **52**(5), 91–98 (2009). https://doi.org/10.1145/1506409.1506429
12. Intel.: Intel Stratix 10 Device Security User Guide. V20.3 (2020a)
13. Intel.: Intel Stratix 10 Hard Processor System Technical Reference Manual. V20.2 (2020b)
14. Ionescu, A.: Getting physical with usb type-c. In: Recon Brussels (2017)
15. Jacob, N., Heyszl, J., Zankl, A., Rolfes, C., Sigl, G.: How to break secure boot on fpga socs through malicious hardware. In: Cryptographic Hardware and Embedded Systems – CHES 2017, Springer, Lecture Notes in Computer Science, vol 10529, pp 425–442, (2017) 10.1007/978-3-319-66787-4_21
16. Joux, A.: Authentication failures in nist version of gcm (2006)
17. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: an experimental study of dram disturbance errors. SIGARCH

Comput. Archit. News **42**(3), 361–372 (2014). https://doi.org/10.1145/2678373.2665726
18. Krautter, J., Gnad, D.R.E., Tahoori, M.: Fpgahammer: remote voltage fault attacks on shared fpgas, suitable for dfa on aes. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**, 44–68 (2018)
19. Li, LW, Duc, G., Pacalet, R.: Hardware-assisted memory tracing on new socs embedding fpga fabrics. In: Proceedings of the 31st Annual Computer Security Applications Conference, ACM, New York, NY, USA, ACSAC 2015, pp 461–470, (2015) https://doi.org/10.1145/2818000.2818030
20. Linaro.: Op-tee: Open portable trusted execution environment.https://github.com/OP-TEE (2019)
21. Olson, L. E., Power, J., Hill, M. D., Wood, D. A.: Border control: sandboxing accelerators. In: 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp 470–481, (2015) https://doi.org/10.1145/2830772.2830819
22. Snare, Rzn.: Thunderbolts and lightning - very, very frightening. In: Proceedings of SyScan Singapore (2014)
23. Weissman, Z., Tiemann, T., Moghimi, D., Custodio, E., Eisenbarth, T., Sunar, B.: Jackhammer, : Efficient rowhammer on heterogeneous fpga-cpu platforms. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(3), 169–195 (2020). https://doi.org/10.13154/tches.v2020.i3.169-195
24. Xilinx.: Ar # 72243 zynq ultrascale+ mpsoc/rfsoc: Ug1085 v1.9 - incorrect statement on encrypt only secure boot. (2019a) https://www.xilinx.com/support/answers/72243.html
25. Xilinx.: Ar # 72654 zynq ultrascale+ mpsoc/rfsoc: Acp usage with xmpu / xppu / trustzone isolation. https://www.xilinx.com/support/answers/72654.html (2019b)
26. Xilinx.: Zynq Ultrascale+ Device Technical Reference Manual. V1.9 (2019c)
27. (Xilinx) LS, : Isolation Methods in Zynq Ultrascale+ MPSoCs. **XAPP1320 (v1.0)**,(2017)
28. Zhang, N., Sun, K., Lou, W., Hou, Y.T.: Fpga-based remote power side-channel attacks. In: 2018 IEEE Symposium on Security and Privacy (SP), pp 229–244, https://doi.org/10.1109/SP.2018.00049 (2016)
29. Zhao, M., Suh, G.E.: Fpga-based remote power side-channel attacks. In: 2018 IEEE Symposium on Security and Privacy (SP), pp 229–244, https://doi.org/10.1109/SP.2018.00049 (2018)