How to Break Secure Boot on FPGA SoCs through Malicious Hardware

Nisha Jacob[1], Johann Heyszl[1],
Andreas Zankl[1], Carsten Rolfes[1], and Georg Sigl[1,2]

[1] Fraunhofer Research Institution AISEC, Munich, Germany,
firstname.lastname@aisec.fraunhofer.de
[2] Technische Universität München, EI SEC, Munich, Germany,
sigl@tum.de

Abstract. Embedded IoT devices are often built upon large system on chip computing platforms running a significant stack of software. For certain computation-intensive operations such as signal processing or encryption and authentication of large data, chips with integrated FPGAs, FPGA SoCs, which provide high performance through configurable hardware designs, are used. In this contribution, we demonstrate how an FPGA hardware design can compromise the important secure boot process of the main software system to boot from a malicious network source instead of an authentic signed kernel image. This significant and new threat arises from the fact that the CPU and FPGA are connected to the same memory bus, so that FPGA hardware designs can interfere with secure boot routines on FPGA SoCs that are without any interruption on regular SoCs. An enabling factor is that integrated hardware designs are likely bought from external partners and there is a realistic lack of security review at the system integrators. This facilitates flaws or even unwanted functionality in such hardware designs. We perform a proof of concept on a Xilinx Zynq-7000 FPGA SoC, and the threat can be generalized to other devices. We also present as effective mitigation, an easy-to-review and re-usable wrapper module which prevents any unauthorized memory access by included hardware designs.

Keywords: FPGA SoCs, secure boot, hardware design, outsourced, threat

1 Introduction

We are currently experiencing a rapid increase in the number of embedded devices being used in the context of the Internet-of-Things (IoT) and cyber physical systems. The application domains of such systems range from automotive, aviation, infrastructure, to industrial production and even home appliances. Across all domains, embedded systems are mostly build on powerful high-volume System on Chips (SoCs) running a mixture of open-source and closed source software, and include network communication interfaces. Such devices perform critical tasks, however, are at the same time physically accessible for attackers in many cases. Fortunately, several security mechanisms have been developed to counteract possible attacks based on physical access. The arguably most important and widely adopted countermeasure is a secure boot mechanism which ensures that only authentic and unmodified software can be run right from the start of the first code within the CPU. This prevents attackers from manipulating software images and restarting devices into a manipulated behaviour. As such, it can be seen as the foundation of all further software security measures. Some applications of embedded systems require high computational capabilities for signal processing or cryptographic operations. At the same time, devices often remain in the field for many years which means that updates of the functionality are likely required. For such cases, FPGA manufacturers such as Xilinx have produced SoC chips known as FPGA SoCs which include configurable hardware logic on the same chip as a conventional CPU architecture. Embedded systems built using such devices (e.g. [26]) are able to support significant computational capabilities through hardware acceleration while both, software as well as hardware can be updated in the field.

However, this additional configurable hardware may lead to severe security issues. Configurable hardware within FPGA SoCs is typically connected to high-bandwidth memory buses of the main CPU which means that hardware blocks may possibly access memory regions which are access-managed by the software system. This has severe consequences for the system, because this may easily corrupt the security of the entire system. We have in the past seen similar attack vectors in the PC world which were successful. For instance, external high-speed interfaces have been misused to directly access internal memory [12, 24]. Fortunately, countermeasures such as Input Output Memory Management Units (IOMMUs) have also been developed against such attacks [3] which handle the memory management of peripherals with direct access to memory and thereby prevent unauthorised memory accesses. In our opinion, however, such threats are now possible from a new direction i.e., through the integrated hardware. In order to understand the reasons for this, it is important to realize that hardware designs (e.g. cryptographic accelerators) will possibly come from 'external' sources. This is simply cheaper and provides a faster time-to-market. Open-source software is used for similar reasons. Even large ASIC SoCs for short-lived consumer-grade routers/modems are designed with outsourced hardware modules, which means that the following threat also applies to ASICs in such cases. For example, the Elastic Compute Cloud (EC2) from Amazon Web Services (AWS) now includes an instance with integrated FPGAs (EC2 F1 instance) where IP cores can be used from a dedicated IP market place [15]. While designers of embedded systems will likely have sufficient software expertise within their team, they will, in many cases, lack proper hardware engineering expertise along with the required extensive tooling for hardware development and verification. If hardware is sourced from elsewhere, it is hence questionable whether the embedded system designers/integrators will be able to properly review the hardware code to check for unwanted functionality or possible attacks. In many cases, hardware blocks will even be delivered as a synthesized netlist which more or less prohibits proper review. The problem is that malicious functionality could be part of such hardware modules.

Previous contributions have already highlighted some of the issues arising from this where unwanted additional functionality in hardware blocks leaks or corrupts sensitive information. E.g. Kutzner et al. [19] describe how an AES core could be maliciously modified such that the key of the last round is output instead of the cipher text. Other contributions have demonstrated how cryptographic keys could be leaked via intentional side-channels such as the power consumption [21] or over the wireless channel [17]. King et al. [18] and Yang et al. [29] show how a privilege escalation of applications can be achieved at run-time using malicious hardware blocks. For this, King et al. modify the data cache and MMU of the Leon processor, while Yang et al., modify the register that holds the privilege bit of the OpenRISC processor. Li et al. [20] describe how a hardware core, which is originally purposed for memory tracing in the context of software debugging, may include unwanted functionality to inject code into the running system. As a proof of concept, Li et al. demonstrate how a log-in password check of a Linux Operating System (OS) can successfully be circumvented by a hardware core scanning and manipulating the memory on the Xilinx Zynq-7000. Jacob et al. [23] show how public authentication keys can be overwritten by a malicious hardware core in FPGA SoCs so that the devices accepts malicious system updates.

In this work, we show that even the secure boot process, one of the most important and basic security features of embedded systems, can be compromised by malicious hardware blocks in the FPGA on FPGA SoCs. We describe a proof of concept on the Xilinx Zynq-7000 device and explain why even later models which include IOMMUs (and are fit to counteract attacks such as described by Li et al. [20] and Jacob et al. [23]) will likely be susceptible to such attacks. Our proof of concept system includes a conventional software stack along with an additional hardware block for the FPGA. The included unwanted hardware functionality overwrites parameters of the second stage boot loader, U-boot, during the secure boot so that an *unauthorized* kernel image is retrieved and booted over the network instead of the local *authorized* image even though all previous boot stages are properly verified prior to that. In our opinion, it will not be realistic for general design teams to acquire the necessary hardware expertise to prevent this. Hence, we developed an efficient countermeasure, which provides full re-usability, and is easy-to-review because of the small code size, that protects against all unauthorized memory accesses through hardware cores while raising an alarm at every attempt. It is a simple hardware wrapper for the AXI bus interface which is easy to wrap around all outsourced hardware cores with memory access and is configured through every access from software. It can be seen as a stripped-down IOMMU which instead works straight from configuration (without requiring the software to explicitly enable or configure it, which is usually done after boot in the OS) and has a smaller set of functionality, thus, trusted code base.

The paper is organised as follows. Section 2 reviews the security of embedded systems generally. Section 3 outlines the attack on the secure boot process. Section 4 describes the Xilinx Zynq device and boot sequence along with a generalization to other devices. Section 5 presents the proof of concept along with a discussion. Section 6 presents the countermeasure.

2 On the Security of Embedded Systems

To highlight the importance of a secure boot process we review popular security mechanisms for embedded systems in this section. They can be divided into three general kinds of security measures built on top of each other.

Hardware Security. Since many IoT embedded systems are in physical reach of potential attackers, security must be rooted in the hardware of respective devices. Hardware security mechanisms for instance include protection against physical tampering, which can be achieved by using tamper-proof casings with light sensors to detect break-ins. Chip-internal tamper detection sensors typically monitor clock and voltage inputs to prevent fault-injection attacks [4]. In case of a tamper event, critical data is e.g. cleared and the system is shut down. Many SoCs for embedded systems include dedicated secure memory for keys and/or certificates. This helps to refrain from storing such information on external memories. Another important aspect of hardware security is to protect debug interfaces using e.g. passwords so that read-out and/or corruption of data and software is prevented in the field [13].

Secure System Startup. One of the most important security mechanisms is a secure boot process. In this process, all executed code is verified for integrity and authenticity using cryptographic means before execution. This means that right after the system startup, the running software can be trusted, which is the required foundation for all later security mechanisms. For a secure boot, a chain-of-trust is established which starts from the very first code that is executed from within the CPUs internal hardwired ROM (also including respective keying material), commonly called the hardware root-of-trust [10]. Given that attackers have physical access, a trusted boot process using Trusted Platform Modules (TPMs) on the contrary, cannot provide this assurance, since the external TPM can always be manipulated by malicious software on the main CPU (comparable to a man-in-the-middle attack). Using similar functions as secure boot, secure update processes allow updates in-the-field from authentic sources by checking authenticity and integrity as well as decrypting confidential data using cryptographic methods. Both require a secure storage for the respective key material.

Runtime Security. After the device startup, all assets such as cryptographic keys, processed data and control functions must be protected against run-time attacks on the software. This only makes sense if the running software is trusted from the start (i.e. secure boot). Popular mechanisms include different kinds of software isolation and virtualization to prevent corrupted processes (e.g. after successful exploits) from accessing sensitive or higher-privilege information. Memory isolation can be achieved through the OS or a hypervisor. Both need hardware support in the form of a Memory Management Unit (MMU) and/or IOMMUs [3]. Trusted execution environments [25] provide an additional privilege level for processes where sensitive processes including their memory regions for e.g. keys, and even peripherals, can be put into a secure world to prevent access by corrupted processes from the normal world.

Through this work we would like to highlight that even if we have an FPGA SoC with a secure hardware root-of-trust and a secure boot process, as well as runtime protection (by isolation of software components), we still have a major risk that the system may be compromised. This risk comes from the reconfigurable hardware. We show that protection mechanisms such as a secure boot, MMU, or privilege levels do not help against such threats.

3 Attacking the Secure Boot on FPGA SoCs

In this section, we outline the general idea of an attack on the secure boot process in the context of FPGA SoCs. The main underlying observation for all secure boot processes is that the verification of the authenticity and integrity of a software image is either not done in-place, or, more importantly, the process which performs the verification and later hands off control to the subsequent stage is inherently not atomic in the sense that it could be interrupted by manipulations. This is an issue that we generally want to highlight and which is of particular relevance for embedded systems built upon FPGA SoCs. In most cases of conventional SoCs, there is no reason to believe that a manipulation of the memory is happening while the CPU is executing the secure boot code, since no-one besides the CPU is accessing the memory. However, in the case of FPGA SoCs, hardware cores on the FPGA have access to the shared memory bus. Hence, such cores, once loaded, present as immediate additional actors on those buses and may manipulate memory content while the CPU is not 'aware' of this. Specifically, such hardware cores, once loaded, are able to manipulate the boot process such that a malicious software image is executed instead of an authentic one. This can be achieved by the hardware secretly overwriting parts of a running bootloader. We present a proof of concept on the Xilinx Zynq-7000 FPGA SoC.

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

4 Relevant Properties of the Xilinx Zynq-7000

We chose the Xilinx Zynq-7000 device for our practical proof of concept since it is a popular choice for contemporary embedded system designs. Also, the insights are generalizable to later models as well as devices from different manufacturers.

The Xilinx Zynq-7000 is an FPGA SoC consisting of a dual core ARM Cortex A9 CPU and a Xilinx 7-series FPGA fabric on the same die. The processing system includes a MMU and two-level cache. A small on-chip RAM of 256KB is available for the storage of sensitive information or code. A larger external DDR memory can be accessed via the memory controllers. The main memory bus is an ARM AMBA AXI bus system. External communication is supported through CAN, I2C, Ethernet and USB interfaces. The device includes a hard-core AES and HMAC implementation which are used during the decryption and authenticity verification of images and configuration files to be run on the Zynq-7000. The ARM trusted execution environment known as TrustZone is available for runtime security of the software system. A one-bit hardware setting divides all processes and peripherals into either a secure, or a normal world.

4.1 Secure Boot Process on the Xilinx Zynq-7000

Figure 1 depicts the boot process on the Zynq-7000 FPGA SoC. It consists of five stages after power-up:

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

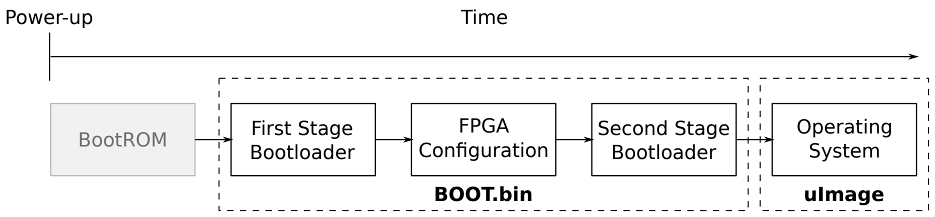

Fig. 1: Overview of boot process on Xilinx Zynq-7000

The FSBL, bitstream and second stage bootloader are packed into a single boot image i.e., BOOT.bin as separate partitions. Each partition within the boot image is separately encrypted and authenticated. Figure 2 depicts the structure of such a partition. It contains the payload as the main part. For AES/HMAC-based authentication and integrity checks, the HMAC key as well as the HMAC digest are appended to the payload before encryption. Xilinx uses the MAC-then-encrypt order of encryption and authentication i.e., the message digest of each partition is first computed followed by its encryption. The key for the AES encryption can either be stored in the battery-backed RAM or in eFuses of the chip. The selection of the AES key source can be enforced by setting the corresponding eFuse. If the optional RSA algorithm is used for authentication, an RSA signature verification is computed in software and the signature as well as the certificate are appended to the partition. The hash of the RSA public key, which is used to validate the certificate, is stored in an on-chip eFuse array.

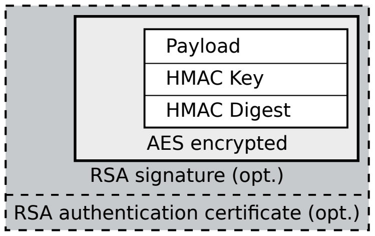

Fig. 2: Content of an authenticated and encrypted boot partition

After power-up, the CPU executes the hardwired instructions from the internal BootROM which is a small and inaccessible read-only memory of 128KB. It also initializes the clocks and configures the first ARM processor core along with the necessary peripherals to fetch the FSBL from Non-Volatile Memory (NVM) based on the boot mode stating the source where the FSBL can be fetched (i.e. SD card, QSPI flash, NAND flash or NOR flash). The boot mode is determined by the voltage levels on the chip's external pin[3]. The BootROM code then copies the FSBL from the NVM to the 192KB[4] on-chip memory (which is typically large enough). The FSBL code is decrypted and authenticated using the AES/HMAC core on the fly while copying it to the internal memory. Upon successful verification of the HMAC, control is handed off to the FSBL and it is executed from the same internal memory. The FSBL is a Xilinx-specific bootloader which initializes clocks, GPIOs, DDR controller and the FPGA fabric. Following the initializations, the FSBL loads subsequent partitions. Xilinx provides a template of the FSBL code, which can be customized. The FSBL controls the decryption and authentication of the bitstream and second stage bootloader. If a bitstream is part of the boot image, this is loaded next. (The bitstream may alternatively be loaded at a later stage of boot through the U-boot or the OS. This, however, is uncommon since it requires additional code to be inserted into the U-boot or later OS instead of using the Xilinx template. Also, hardware acceleration

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

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

would not be available for the verification of the OS image.). The bitstream is decrypted and authenticated using AES/HMAC while it is loaded into the FPGA configuration memory. If the verification fails, the FPGA containing an unauthentic configuration is not activated. As a next step, the second stage bootloader, e.g., U-boot is decrypted and authenticated. As the on-chip memory is not large enough for typical loaders, the decrypted U-boot is stored on the external DDR memory. If the verification is successful, control is then handed off to U-boot. After initialization of the platform (processor, clocks, memory) and reservation of memory, U-boot enters the main loop where it decrypts and authenticates the kernel image. U-boot then reads the kernel image header and jumps to the address of the kernel header, handing off control to the OS.

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

5 Proof of Concept: Breaking the Secure Boot on Xilinx Zynq-7000

In this section, we describe our proof of concept where we practically break the secure boot process of the Xilinx Zynq-7000 FPGA SoC using a hardware module. The investigation was carried on a Zedboard Rev. D development board with 512MB of external DDR memory. The FSBL v2015.4, u-boot-xlnx v2016.1 and linux-xlnx v2016.1 from Xilinx are used [27]. The Xilinx tool *bootgen* is used to encrypt and compute the message digests of each partition. Each partition of the boot image is decrypted and authenticated using the on-chip AES/HMAC. The AES encryption key is stored in the battery-backed RAM.

Hardware Module. As a likely scenario for the proof of concept, we chose a hardware module similar to a cryptographic accelerator which we connect to the AXI bus of the Xilinx Zynq-7000. In our case this module only XORs two input values, which can be seen as a placeholder for several meaningful cryptographic operations. We use an interface which is typical for high-speed hardware accelerators and consists of a low-speed slave interface for configuration and control (i.e. source address, destination address, length, and enable signal) as well as a high-speed master interface for data transfer [5, 6, 11]. The master interface is DMA-like and, hence, reduces the load on the processing system by not requiring the CPU for data input/output. The CPU only passes the configuration information after which the module starts to perform its core function whilst accessing data directly from the memory. Upon completion, a flag is set to alert the processor.

In addition to this, the module also contains unwanted functionality. Using the high-speed data interface, the module progressively scans the external DDR memory and maliciously alters its content. It specifically scans the U-boot binary in the DDR memory for the kernel boot parameters. After finding the target memory location, the boot parameters are modified to load an unauthorized kernel image from a remote server over the network instead of booting from the verified source. For this unwanted functionality, our example hardware module requires 117 LUTs and 46 Slices in addition to the original functionality. This hardware overhead will likely pass unnoticed since e.g. a high-speed AES-GCM core for the same family of FPGAs and including a similar interface requires 22.7 k LUTs and 6.9 k Slices [5] for example, which is larger by orders of magnitude. Another smaller example of a SHA-384/512 core (excluding the interface) for the same FPGA family still requires 2.5 k LUTs and 700 Slices [14], which is also significantly larger.

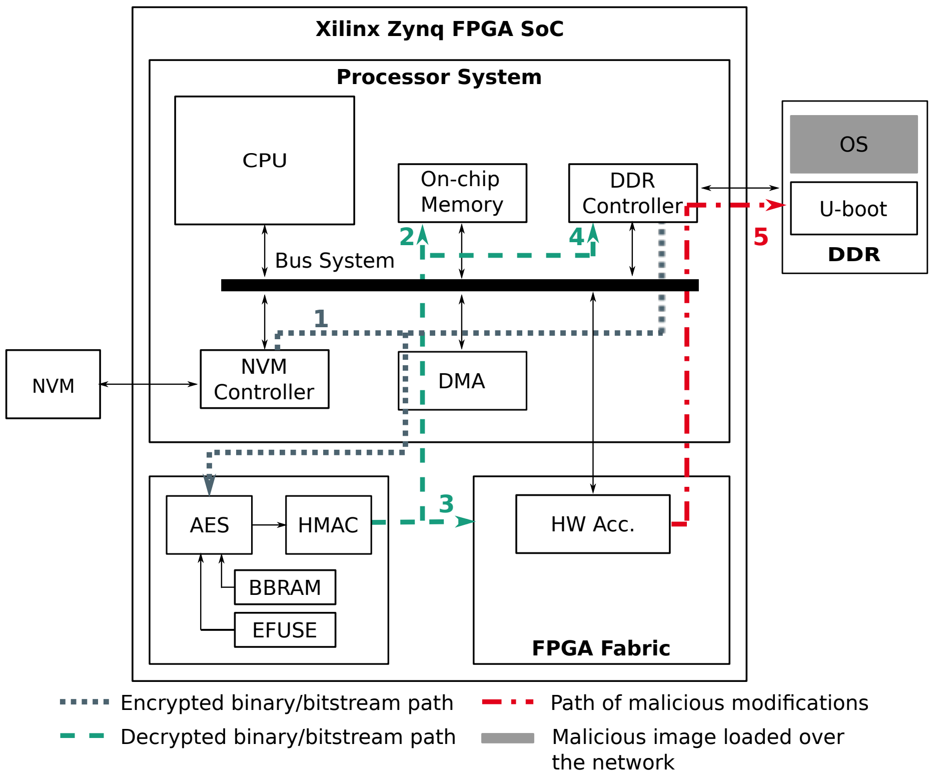

Fig. 3: Secure boot attack on Xilinx Zynq-7000 FPGA SoC

Sequence of Events During the Attack. Figure 3 depicts the Xilinx Zynq-7000 FPGA SoC and highlights the data flow of the encrypted and decrypted images during the secure boot process as it is described in Section 4. The numbers in the figure indicate the sequence of events during the attacked boot process. In step

1 (dotted grey path), the encrypted and authenticated partitions are read from NVM (our system boots from an SD-card) or DDR memory and piped through the AES/HMAC cores for verification[5]. Boot steps 2-4 (dashed green paths) depict the successive loading of the FSBL, bitstream and U-boot respectively, as described in Section 4.1. Following the successful verification and loading of the bitstream to the FPGA, U-boot is verified and loaded to the external DDR memory and control is handed-over. As soon as the hardware is activated after the successful verification of the HMAC, the malicious core starts to scan the external DDR memory for a particular U-boot setting which is to be modified. This is the actual attack and depicted as step 5 in Fig. 3.

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

(a) Authentic image

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

(b) Corrupted image

Fig. 4: Excerpt of the .rodata section of the U-boot image

Figure 4a shows an excerpt of the trusted U-boot image which contains strings representing the boot parameters in the .rodata section of the image. In step 5 explained above explained, the hardware module searches for the SD card boot parameters i.e., `fi.sdboot=` (see Figure 4a). Upon locating the string, the following original boot parameters from the authentic image, which are used to load and verify the kernel image on the SD card, are overwritten:

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

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

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

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

The `bootp` command downloads a file from the specified server IP address to the DDR memory[6]. Once the image is downloaded to the DDR memory from the remote server, the kernel is booted. For this, the malicious core also writes a regular U-boot boot command, which specifies the locations where the `bootp` command had placed the kernel image, uramdisk and devicetree in the DDR memory, as follows:

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

Figure 4b shows the same excerpt as Fig. 4a, with the described malicious modifications highlighted in grey. (In our set-up, the kernel image (`uI`), devicetree (`dt.dtb`) and uramdisk (`ur.gz`) are three separate images and, hence, successively loaded.) Note that the U-boot code after the overwritten section of code is corrupted. This is highlighted in red in Fig. 4b but does not make a difference as control is handed off to the OS after the `bootm` command.

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

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

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

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

5.1 Discussions and Generalizations

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

IOMMUs. In general, IOMMUs or System MMUs (as called by ARM), are designed to protect the system against threats such as the one demonstrated in this contribution where bus peripherals access shared memory resources without

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

proper authorization. IOMMUs are hardware cores which, when properly configured, control bus access rights of such peripherals. They are even available in newer (and partly more expensive) FPGA SoCs such as the Xilinx UltraScale+ [28] and Altera Stratix 10 [1]. However, IOMMUs are typically initialized by the OS, which is the last stage in the boot process. Hence, we conclude that the availability of IOMMUs on FPGA SoCs will not generally prevent the described attack because this would require an earlier configuration. We advise the use of the countermeasure presented in the next section instead.

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

Static Access Restrictions on Xilinx Devices. The Zynq-7000 allows to statically restrict the access of peripherals to the memory at design time. This means that under no operational circumstances, the hardware may access certain excluded memory regions. A similar feature is also offered by Microsemi for the SmartFusion2 FPGA SoCs [22]. However, this would pose a drastic limitation for designs since for most cases, the final use of a hardware module such as a cryptographic accelerator will be determined by software and it will be beneficial if all memory regions can be accessed. For example, a cryptographic core that is used for run-time integrity checking of software requires access to all memory regions.

On the Xilinx Zynq UltraScale+ series, a XMPU module can be used to restrict the access of masters on the bus. This access configuration can optionally be locked at boot time (recommended method by Xilinx [28]). This, however, means that the settings can only be changed after a power-on-reset using a modified and signed image. Alternatively, if this setting is not locked at boot time, the configuration can be changed at run-time. In our opinion, a static configuration will not likely be used for acceleration-type cores since it poses as an unfavorable restriction to designs. Instead the countermeasures presented in the next section is advised.

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

The order of the boot of FPGA and software system influences the vulnerability of the boot process. In case of FPGA SoCs from Altera, three cases of boot [2] are available: (i) the CPU boots and configures the FPGA during its boot sequence (similar to Zynq-7000), (ii) the FPGA is configured first and the CPU boot sequence is controlled by the FPGA, and (iii) the FPGA and CPU boot independently. The first case is the same with the same issue, the second is even worse as the FPGA is configured first. In the last, the FPGA and CPU are booted independently so the success will depend on whether the FPGA boots before the OS or during the CPU. In all cases, devices like, e.g. Stratix V and Arria 10 have no IOMMU and are vulnerable at run-time at least.

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

To summarize the above cases, whenever the FPGA is configured early during the boot sequence, and this is often the case, a secure boot process can be compromised by a malicious core in the FPGA and the use of the countermeasure described in the next section is advised.

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

Finding the Location of the Code to be Overwritten. One of the key factors for attack vectors as the one described in this contribution is to estimate the location of the code which is to be overwritten. The less precise this is, the more code needs to be searched which takes more time. Interestingly, the location of the respective U-boot image depends only on a few factors and can be determined using publicly accessible information. Within the U-boot image, the presented attack overwrites U-boot environment variables which are located in the `.rodata` section of the image in the form of strings. By being part of the U-boot image, the variables are authenticated. (Technically, there are cases where such variables are instead retrieved from other, unauthentic external memories, and are loaded onto the heap in RAM at run-time. However, this option is completely unreasonable in the context of secure embedded systems since attackers with physical access may easily modify them [16]. Hence, we assume that U-boot is compiled such that it does not read environment variables from external memory.)

U-boot is generally unaware where the previous bootloader, in our case the FSBL, was loaded. Hence, after the basic initialization, U-boot checks the current value of the program counter to determine the location. By default, the FSBL loads the U-boot binary to start at address `0x4000000`. There are regions in DDR memory, which need to be preserved to store the kernel image, devicetree and uramdisk. Hence, if U-boot detects that the previous loader has put it into those regions, it relocates itself to a predefined region before continuing execution. The relocation offset is usually at the end of the RAM so that one big continuous part of the memory remains for the OS. By analysing the U-boot code (which is open-source), the offset address can be retrieved[7]. In our practical investigation, we found that the initialization before the relocation takes approximately 37 *ms*. Afterwards, the remaining part of the initialization is done, which takes approximately 400 ms on our example setup. Finally the kernel is loaded as a last step. From this we see that an attack could target the initial location of the U-boot during the 37 *ms* until it is relocated, or the final location during the 400 *ms* of further initialization.

Timing and Durations. The available time to perform a search for the specific string to be overwritten depends on whether it is done before or after the relocation, as described above. Our presented hardware module is able to scan up to 2.1 MB of memory, and successfully modify the specific boot parameters during the shorter time of 37 *ms* before relocation. For comparison, the size of a standard U-boot is about 3 MB which already hints at the high likeliness of success. Our practical investigations have indeed shown, that for both cases, the attack could be performed successfully.

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

6 Wrapper Countermeasure

In this section, we propose a general countermeasure to protect systems against unauthorized memory access from hardware cores within the FPGA part of FPGA SoCs. For this purpose, we developed a flexible and lightweight security-enhanced hardware wrapper for cores with an AXI interface[8].

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

Typically, a software process using a hardware core writes configuration information (source address, destination address, length, enable) to the core via the core's slave interface. Based on this information the core performs its function

[7] The U-boot command `bdinfo` outputs the relocation offset on a running system.
[8] The source code for the wrapping module can be retrieved from `https://github.com/Fraunhofer-AISEC/axi-firewall`

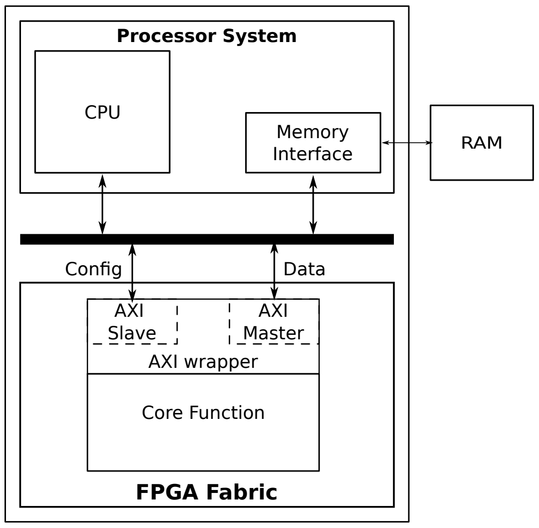

Fig. 5: Hardware core with security-enhanced wrapper

using the master interface for memory access. Figure 5 depicts a system architecture of an FPGA SoC including a core which is wrapped with our proposed design. Our wrapper uses the received configuration information to monitor the AXI-transactions of the master interface and only allows access to the memory range specified by the software process while leveraging the time during the AXI handshake to enforce the access commands. In a typical AXI transaction, the address channel is first set followed by the data channel. A transaction begins with the master sending the address of the memory location to be read/written along with an address valid signal. It then waits until the slave responds with an address ready signal. Next, the slave sends a data valid signal and the master responds with a data ready signal following which valid data can then be read /written to the memory. The wrappers checks the address issued by the master while it waits for the slaves' address ready response signal. If the wrapped underlying core attempts to access memory outside of the allowed range, an alarm signal is set. Thereby aborting a transaction before any valid data could be read/written to the memory while not affecting the performance of legitimate transactions. The wrapper also checks the length of data read/written as per the current configuration. If the core attempts to read or write more data, the remaining transactions are dropped and an alarm is raised. Furthermore, the wrapper also ensures that the core is only functional when a software process has explicitly set the enable signal, which prevents the core from performing accesses while technically in idle state. If the core tries to initiate any unauthorised transactions, an alarm is raised. Thus the core is not able to enable itself or modify its configuration settings.

Table 1: Wrapper error codes

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

The alarm signal can be connected to the interrupt controller or a separate tamper detection unit. Currently, all subsequent transactions are blocked after an alarm is raised. However, based on the criticality of the system, other actions can be taken such as e.g. putting the system into a secure lockdown mode. The wrapper also includes a 2-bit error output code to indicate the cause of the alarm which is listed in Table 1.

For cores that may require access to both the secure and normal world e.g., run-time integrity monitors, the TrustZone setting may not be fixed at boot (through the FSBL). In such cases, cores with a master interface may set the TrustZone security bit themselves and are, hence, free to access the secure world memory without explicit permission. Our wrapper, however, sets the security bit of the master interface to normal world by default unless explicitly reconfigured through software which prevents this.

Our proposed wrapper requires a hardware overhead of 133 LUTs and 55 Slices which is a low overhead compared to the given security gain in our opinion. There is no cycle count penalty during operation.

In a previous contribution, Brunel et al. [7] have developed a secure AXI bridge which is similar to a full IOMMU core for SoCs. The downside in our view is that it requires significantly more hardware resources and contains a significantly larger amount of source code to review. Coburn et al., [8] and Cotret et al. [9] present a post-boot run-time protection core similar to TrustZone. This is achieved by storing system wide or processor specific (for MPSoCs) security policies in large look-up-tables or BRAMS resulting in a significant overhead in area and latency. In contrast, the proposed wrapper protects devices against malicious hardware IP cores. Further, the wrapper reuses the configuration information passed to it from the firmware and leverages the time between the AXI handshaking for the enforcement. Hence minimizing the overhead in terms of area, performance, latency and maintenance of the security policies.

Xilinx provides a module known as the XMPU which can be used to dynamically restrict memory access from early boot stages onwards [28] but is only available for the Zynq Ultrascale+ devices. Unfortunately, the sources of the module are not publicly available. In contrast, our AXI-wrapper can be used for any IP cores with memory access and is not restricted to any manufacturer or device. Also, as the source code of the wrapper is small in size and public, it can be easily reviewed and re-used.

7 Conclusion

We successfully demonstrated the feasibility and practical impact of attacks on the secure boot process of FPGA SoCs through hardware on the FPGA. In a time where services such as the Amazon AWS EC2 F1 instances and embedded systems based on FPGA SoCs entice the broader use of hardware from IP vendors, the trust level of such outsourced hardware is difficult to determine. Hence, in our opinion, hardware cores including unwanted functionality such as the one described here, could become more common in hardware IP marketplaces. Hence, to prevent attacks against secure boot such as the one we presented and protect against similar attacks through unauthorized memory accesses from hardware cores generally, we propose to use our efficient wrapping module as a countermeasure. Alternatively, a strict restriction in the boot order (FPGA last) and early configuration of IOMMUs would be necessary.

References

1. Altera Corporation. Stratix 10 Secure Device Manager Provides Best-in-Class FPGA and SoC Security, 2015.
2. Altera Corporation. Arria 10 SoC Boot User Guide, 2016.
3. AMD. I/O Memory Management Unit. `http://developer.amd.com/wordpress/media/2012/10/48882.pdf`, 2011.
4. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. Cryptology ePrint Archive, Report 2004/100, 2004. `http://eprint.iacr.org/2004/100`.
5. BarcoSilex. BA415-AES-GCM 10 to 100 Gbps IP Core. `http://www.xilinx.com/products/intellectual-property/1-4sw1c9.html`, 2015.
6. BarcoSilex. BA413-SHA1, SHA2 and HMAC IP Core. `http://www.barco-silex.com/ip-cores/encryption-engine/BA413`, 2016.
7. J. Brunel, R. Pacalet, S. Ouaarab, and G. Duc. Secbus, a software/hardware architecture for securing external memories. In *2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, MobileCloud 2014, Oxford, United Kingdom, April 8-11, 2014*, pages 277–282, 2014.
8. J. Coburn, S. Ravi, A. Raghunathan, and S. Chakradhar. SECA: Security-enhanced Communication Architecture. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '05, pages 78–89, New York, NY, USA, 2005. ACM.
9. P. Cotret, F. Devic, G. Gogniat, B. Badrignans, and L. Torres. Security enhancements for fpga-based mpsocs: A boot-to-runtime protection flow for an embedded linux-based system. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), York, United Kingdom, July 9-11, 2012*, pages 1–8, 2012.
10. Dick Wilkins. UEFI Firmware Security Best Practices. *UEFI Plugfest*, 2014.
11. Ensilica. Ensilica eSi - SHA-256. `http://www.ensilica.com/wp-content/uploads/eSi-SHA-256.pdf`, 2013.
12. Gamma International. Tactical IT Intrusion Portfolio: FINFIREWIRE. `https://wikileaks.org/spyfiles/files/0/293_GAMMA-201110-FinFireWire.pdf`, 2011.

13. B. Gonzalvo, E. Bourbao, F. Majéric, and L. Bossue. JTAG Combined Attacks. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2016.
14. Helion. HTSHA-FAST64: Fast SHA-384/512 Hashing. `http://www.xilinx.com/products/intellectual-property/1-8dyf-612.html`, 2016.
15. Jeff Barr. Developer Preview – EC2 Instances (F1) with Programmable Hardware. *Amazon Web Services*, 2016.
16. Jeong Wook Oh. Reverse Engineering Flash Memory for Fun and Benefit. In *Blackhat US*, 2014.
17. Y. Jin and Y. Makris. Hardware Trojans in Wireless Cryptographic ICs. *IEEE Design & Test of Computers*, 27(1):26–35, 2010.
18. S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and Implementing Malicious Hardware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, LEET'08, pages 5:1–5:8, Berkeley, CA, USA, 2008. USENIX Association.
19. S. Kutzner, A. Y. Poschmann, and M. Stöttinger. Hardware Trojan Design and Detection: A Practical Evaluation. In *Proceedings of the Workshop on Embedded Systems Security*, WESS '13, pages 1:1–1:9, New York, NY, USA, 2013. ACM.
20. L. W. Li, G. Duc, and R. Pacalet. Hardware-assisted Memory Tracing on New SoCs Embedding FPGA Fabrics. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 461–470, New York, NY, USA, 2015. ACM.
21. L. Lin, M. Kasper, T. Güneysu, C. Paar, and W. Burleson. Trojan Side-Channels: Lightweight Hardware Trojans through Side-Channel Engineering. In *CHES*, pages 382–395, 2009.
22. Microsemi Corporation. SmartFusion2 and IGLOO2 FPGA Security and Reliability, 2015.
23. Nisha Jacob, Carsten Rolfes, Andreas Zankl, Johann Heyszl, and Georg Sigl. Compromising FPGA SoCs using Malicious Hardware Blocks. In *Design Automation and Test in Europe, DATE 2017*, Lausanne, Switzerland, March 2017.
24. Russ Sevinsky. Funderbolt Adventures in Thunderbolt DMA Attacks. In *BlackHat US*, 2013.
25. Steven J. Murdoch. Introduction to Trusted Execution Environments (TEE). *University of Cambridge*, 2014.
26. Xilinx Inc. The Roads Must Roll: Zynq SoC will be used to build Intelligent Transport System in Singapore. `https://forums.xilinx.com/t5/Xcell-Daily-Blog/The-Roads-Must-Roll-Zynq-SoC-will-be-used-to-build-Intelligent/ba-p/600630`, 2015.
27. Xilinx Inc. Xilinx Github. `https://github.com/Xilinx`, 2016.
28. Xilinx Inc. *UG 1085: Zynq UltraScale+ MPSoC: Technical Reference Manual*, February 2017.
29. K. Yang, M. Hicks, Q. Dong, T. M. Austin, and D. Sylvester. A2: Analog Malicious Hardware. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 18–37, 2016.