BYOTEE: Towards Building Your Own Trusted Execution Environments Using FPGA

Md Armanuzzaman and Ziming Zhao
CactiLab, University at Buffalo
{mdarmanu, zimingzh}@buffalo.edu

Abstract—In recent years, we have witnessed unprecedented growth in using hardware-assisted Trusted Execution Environments (TEE) or enclaves to protect sensitive code and data on commodity devices thanks to new hardware security features, such as Intel SGX and Arm TrustZone. Even though the proprietary TEEs bring many benefits, they have been criticized for lack of transparency, vulnerabilities, and various restrictions. For example, existing TEEs only provide a static and fixed hardware Trusted Computing Base (TCB), which cannot be customized for different applications. Existing TEEs time-share a processor core with the Rich Execution Environment (REE), making execution less efficient and vulnerable to cache side-channel attacks. Moreover, TrustZone lacks hardware support for multiple TEEs, remote attestation, and memory encryption.

In this paper, we present BYOTEE (Build Your Own Trusted Execution Environments), which is an easy-to-use infrastructure for building multiple equally secure enclaves by utilizing commodity Field Programmable Gate Arrays (FPGA) devices. BYOTEE creates enclaves with customized hardware TCBs, which include softcore CPUs, block RAMs, and peripheral connections, in FPGA on demand. Additionally, BYOTEE provides mechanisms to attest the integrity of the customized enclaves' hardware and software stacks, including bitstream, firmware, and the Security-Sensitive Applications (SSA) along with their inputs and outputs to remote verifiers. We implement a BYOTEE system for the Xilinx System-on-Chip (SoC) FPGA. The evaluations on the low-end Zynq-7000 system for four SSAs and 12 benchmark applications demonstrate the usage, security, effectiveness, and performance of the BYOTEE framework.

I. INTRODUCTION

Existing hardware-assisted Trusted Execution Environments (TEE) on commodity computing devices make use of hardware security primitives offered by the CPU, such as Intel SGX [31] and Arm TrustZone [64], to guarantee code and data loaded inside to be protected, with respect to confidentiality and integrity, from the Rich Execution Environment (REE). In recent years, we have witnessed unprecedented growth in using SGX and TrustZone in real-world products and academic projects, which include real-time kernel protections [19], [36], securing containers and runtime libraries [18], [70], [79], shielding applications from attacks [22], [55], [72], etc.

Limitations of Existing TEEs. The hardware and software of existing TEEs nevertheless have the following issues, making them either ineffective, inefficient, or untrustworthy: i) existing TEE architectures, such as Intel SGX and Arm Trust-Zone, are proprietary and only work for specific computing architectures, which users have to trust blindly. It is impossible for users to verify the correctness of the TEE designs or attest the hardware states at run-time; ii) they only provide a static

and fixed hardware Trusted Computing Base (TCB), which cannot be customized for different applications. For TrustZone, it means the TEE has the highest privilege to control the REE and communicate with all peripherals. The design violates the principle of least privilege by including unnecessary peripherals and buggy peripheral drivers in the software TCB [27] and exposing the TEE to malicious peripheral inputs [38]. For SGX, it means applications in enclaves have to go through and trust the REE OS to communicate with peripherals [71], bloating the size of software TCB by including a usually monolithic REE OS kernel, e.g., 27.8M Source Lines of Code (SLOC) of the Linux kernel [23]; iii) the TEE shares a processor core with the REE in a time-sliced fashion, not only costing many CPU cycles [75], [98] for the expensive context switches between the TEE and REE but also making it vulnerable to cache side-channel attacks [25], [30], [39], [97], [99], which directly undermine TEE's security promises; iv) the software TCBs in TEEs are large, creating big attack surfaces for runtime attacks that hijack the control or data flow [17], [101]. For example, Haven [21] places a whole Windows 8 library OS inside the enclave. OP-TEE, a Cortex-A secure world OS, has 277K SLOC [10], and TF-M [54], a trusted firmware for Cortex-M TrustZone, also over 117K SLOC. Partly due to these reasons, in 2022 Intel has removed support for SGX in some future CPUs [78]. In addition, TrustZone has many other limitations: i) TrustZone only provides *one* isolated execution environment; ii) TrustZone does not encrypt the contents of Dynamic RAM (DRAM). Therefore, it is subject to code and data disclosure from cold-boot attacks [40]; and iii) TrustZone does not have native hardware supports for remote attestations of code and data integrity [16], [61] due to the lack of platform registers to accumulate measurements.

While many software-based projects attempted to address one or more of the aforementioned issues by building software layers on SGX, TrustZone, or RISC-V, some issues, e.g., non-verifiable design, non-customizable hardware TCB, time-sharing of the processor, vulnerable to cold-boot attacks due to the slow DRAM decay, etc., are rooted in the design of their respective hardware system, business model and ecosystem, etc.; therefore they cannot be addressed by software alone (refer to Table VI for comparisons). For example, SANCTUARY [24] uses the memory access controller to provide multi-domain isolation for sensitive applications on Cortex-A TrustZone, and CURE [20] enables the exclusive assignment of system resources, e.g., peripherals, CPU cores, or cache resources, to

Vendor	Intel	Xilinx	
	Agilex F	Zynq-7000	UltraScale+ EG
SoC FPGA	R25A [47]	XC7Z007S [11]	ZU19EG [83]
LE/LC	2,692,760	3,600	1,143,000
ALMs/LUT	912,800	14,400	523,000
PLLs	28	2	12
DSP	8,528	66	1,968
BRAM (MB)	259	0.225	80.4
I/O	624	49	347
Hardcore CPU	Cortex-A53 (Q)	Cortex-A9 (S)	Cortex-A53 (Q)
DRAM (GB)	16	0.512	8
Price in 2022	≈$10K	≈$130	≈$4.3K

TABLE I: Example SoC FPGA products. LE: Logic Elements (Intel); LC: Logic Cells (Xilinx); ALM: Adaptive Logic Modules (Intel: Multiple LUTs, registers, adders, and multiplexers make up an ALM); LUT: Look Up Table (Xilinx); DSP: Digital Signal Processing blocks; PLLs: Fabric and I/O Phase-Locked Loops; I/O: Maximum User I/O Pin; S: Single-core; Q: Quad-core. We use a Zynq-7000 system in our evaluations.

each enclave on RISC-V. Even though they address some of the aforementioned issues, both solutions still suffer from cold-boot attacks and a large software TCB. Other hardware-based solutions, such as HECTOR-V [60], Graviton [80], etc., do not address all the issues either, and they cannot be deployed on commodity devices because hardware changes are required.

Benefits of FPGA. Field-Programmable Gate Array (FPGA) is designed to be configured by users using a hardware description language after manufacturing. Besides reconfigurability, it has advantages of high performance, fast development round, etc. As shown in Table I, a variety of FPGA products ranging from embedded systems, i.e., Xilinx Zynq-7000 with only 3,600 logic elements (≈$130), to data center devices, i.e., Agilex F R25A with 2.6 millions of logic elements (≈$10k), are available in the market. Various FPGA-based application-specific accelerators, such as for deep convolutional neural networks [76], [96], recurrent neural networks [53], classic and post-quantum cryptographic algorithms [28], [33], Memcached [51], etc., have been proposed and deployed on such devices [44], [65].

More interestingly, FPGA can also be used to build general-purpose computing platforms, in which users can design and implement their own softcore CPUs or customize existing open-sourced [1], [2], [5]–[8], [13], [15] or proprietary ones [49], [56]. The available softcore CPUs range from the partially configurable (i.e., cache size, pipeline depth, memory management unit, etc.) and proprietary low-end 32-bit MicroBlaze [56], to the fully customizable and open-sourced mid-end 32-bit RISC-V [15], [57] and high-end 64-bit A2I POWER processor [1]. While the low-end softcore CPUs are comparable to micro-controllers, their mid-end and high-end counterparts have performances comparable to hardcore micro-processors [41]. Because it is possible to formally verify the FPGA implementations of system-on-chip resources [26], [74], including the softcore CPUs, users do not have to blindly trust a whole proprietary CPU but just the FPGA configuration modules and verifiers.

Customizing Enclaves Using FPGA. We present a hardware and software co-design framework to Build Your Own Trusted Execution Environments (BYOTEE) on commodity FPGA devices. Using the BYOTEE toolchain, users can easily build multiple equally secure and customized enclaves on-demand to execute their Security-Sensitive Applications (SSA). An enclave can include only the hardware and software functionality necessary for the SSA and exclude other hardware and software components on the system, minimizing the sizes of hardware and software TCB.

At the hardware layer, BYOTEE utilizes commodity System-on-Chip (SoC) FPGA devices, which integrate both hardcore CPU system, e.g., Cortex-A, RISC-V, etc., and FPGA programmable logic architectures. The nature of FPGA enables on-demand configurations and reconfigurations of enclaves' hardware TCBs, which may include softcore CPUs, Block RAM based (BRAM; same as Static RAM on known SoC FPGA devices) main memory, and peripherals. For example, an enclave may have a softcore A2I POWER CPU and a hardware GPU as its dedicated peripheral for acceleration. Each enclave has its own isolated physical address space, which maps its own main memory, system configuration registers, and peripherals. The software on the hardcore CPU and other enclaves cannot see an enclave's address space unless it is explicitly specified in the design. By assigning the hardware resources to co-resident enclaves, BYOTEE builds a multiprogramming environment, isolates software faults, and provides memory protection on FPGAs. Because enclaves do not time-share any processor with each other and the hardcore system, the cache side-channel attack vector is removed. Cold-boot attacks on these enclaves are also difficult due to the characteristics of BRAM.

FPGA alone, however, does not address all the issues of existing TEEs, and the SSAs need external libraries to execute and drivers to control the peripherals. On the software front, the configurable FIRMWARE component of BYOTEE provides necessary software libraries, e.g., libc, and a Hardware Abstraction Layer (HAL), for a minimum software TCB. Even though formally verifying the correctness and security of hardware designs, e.g., softcore CPU, is beyond our scope, BYOTEE offers mechanisms to bootstrap the trust and attest the integrity of the hardware implementations. In particular, to bootstrap trust in enclaves statically and dynamically, BYOTEE offers example paths building on the secure boot or a run-time trusted module, respectively. Additionally, the FIRMWARE offers two attestation mechanisms, namely pre-execution and post-execution attestations. The former extends the traditional remote attestation of code and input to include hardware integrity, whereas the latter extends output data integrity attestation [16].

We implement a proof-of-concept BYOTEE infrastructure and toolchain for the Xilinx SoC FPGA. The toolchain includes HARDWAREBUILDER, which takes developers' hardware resource need as input and automatically generates hardware modules and interconnections. HARDWAREBUILDER enables developers to focus on SSA development, increases the

usability of BYOTEE, and decreases the chances of developer-induced misconfigurations. The system and toolchain also include FIRMWARE, which provides a run-time environment, and SSAPACKER, which encrypts and signs an SSA binary. We demonstrate the usage of BYOTEE with four example SSAs. We emphasize none of the existing TEE solutions can meet all the security needs of the SSA-3, which is a secure music player with digital rights management. We evaluate the security and performance of BYOTEE with 12 benchmark applications and the four SSAs. The evaluation results on the low-end Zynq-7000 system show BYOTEE is effective in defeating attacks and efficient in executing SSAs. The contributions of this paper are summarized as follows:

- We present BYOTEE, a framework for building multiple general-purpose enclaves with configurable hardware and software TCBs utilizing commodity FPGA devices. The idea of BYOTEE can be implemented on FPGA systems from any vendor;
- We present example paths to bootstrap trust in an enclave's hardware and software statically or dynamically. We design and implement two attestation mechanisms that capture the identifies of boot loaders, bitstream (hardware implementation), FIRMWARE, and SSA;
- We implement the BYOTEE system and toolchain for the Xilinx SoC FPGA, which include the HARDWARE-BUILDER, FIRMWARE, SSAPACKER, and other helper tools, e.g., linker scripts, to facilitate developers to design, develop, and debug their SSAs. We open-source the BYOTEE system and toolchain[1];
- We choose the very low-end MicroBlaze softcore CPU and Zynq-7000 system to demonstrate BYOTEE's usage, security, effectiveness, and performance with the Embench-IoT benchmark suite and four SSAs.

II. BACKGROUND: SOC FPGA

In this section, we discuss the hardware components of SoC FPGA, followed by its design, development, and run-time workflow.

A. Hardware Components

A SoC FPGA comprises: i) hardcore CPU system, which is formed around hard processors, such as the Cortex-A processor, to run the traditional operating systems and applications; ii) FPGA, which can implement arbitrary systems, including softcore CPUs, high-speed logic, arithmetic, and data flow subsystems. In addition to the general fabric, the FPGA has Block RAMs (BRAM) to store data. Note that BRAM is made of Static RAM (SRAM) on existing SoC FPGA platforms. Compared to DRAM whose cells are made of capacitors and is vulnerable to cold-boot attacks due to the slow decay [40], SRAM decays faster [66]. FPGA is configured with a *bitstream*, which is programmed in hardware design description languages, such as Verilog, VHDL, etc.; iii) other integrated on-chip memory and high-speed communications interfaces.

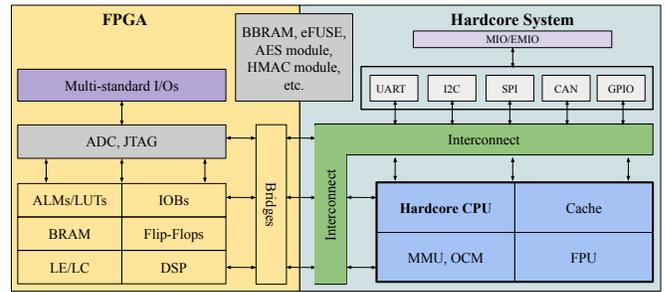[1]https://github.com/CactiLab/BYOTee-Build-Your-Own-TEEs

Fig. 1: An overview of SoC FPGA hardware

Figure 1 shows the architecture of SoC FPGAs. The main building blocks of the hardcore CPU system include the hardcore CPU, its Memory Management Unit (MMU), On-Chip Memory (OCM), caches, etc. Additionally, the hardcore system connects external I/O interfaces, such as SPI, I2C, UART, etc. Bridges and interconnects connect the hardcore system and FPGA interfaces. The FPGA side is mainly composed of configurable logic blocks, logic cells, Adaptive Logic Modules (ALMs), Lookup Tables (LUTs), flip-flops, switch matrix, carry logic, BRAM, and Input/Output Blocks (IOB) for interfacing. The FPGA also includes modules for Analogue to Digital Conversion (ADC) and a set of Joint Test Action Group (JTAG) ports for configuration and debugging. Usually several security-relevant modules, including non-volatile Battery-backed RAM (BBRAM), one-time programmable electronic fuse (eFUSE), and cryptographic accelerators, are connected to both the hardcore system and FPGA.

B. Development and Run-time Workflow

A typical design and development flow of general-purpose computing platforms on SoC FPGA involves: i) the development of the hardware system on the FPGA, including designing the peripheral blocks and creating the connections between these blocks and the hardcore CPU system. In this step, a developer can use and customize open-sourced and proprietary hardware IPs; ii) the development of the software system on the hardcore and softcore CPUs. To be compatible with SoCs without FPGAs, when a SoC FPGA device is powered on, the hardcore system boots first before going on to configure the FPGA.

To enable secure and trusted boot, unique AES and RSA keys can be generated off the device and programmed to the persistent secure storage, e.g., eFUSE array, on the device. The keys are used to encrypt and sign the bitstream and the firmware (e.g., in ELF format) that runs on the softcore CPU in the development stage. The device first starts with the hard-wired boot ROM, which verifies, decrypts, and loads the First Stage Boot Loader (FSBL) from supported interfaces, such as SD card, JTAG, etc. The FSBL, in turn verifies and sets up the FPGA with bitstream using the device configuration (DevC) [94] interface and firmware, after which the firmware on FPGA starts execution. The FSBL also verifies the Second Stage Boot Loader (SSBL), such as U-Boot, and gives control of the hardcore system to it. The SSBL verifies and boots the operating system on hardcore CPU. In addition to boot-

time configuration, some FPGAs can also be programmed at run-time. For instance, software running on the hardcore of some Xilinx Zynq devices can use the DevC and Processor Configuration Access Port (PCAP) interfaces to reconfigure the whole or part of the FPGA.

III. SYSTEM, THREAT AND DEPLOYMENT MODEL

System Model. We assume the DRAM can be configured to connect to both of the hardcore system and FPGA, and peripherals can be connected to the FPGA without routing through the hardcore system. The former enables the hardcore system and FPGA modules to communicate efficiently via shared memory, and the latter makes sure the software on the hardcore system cannot eavesdrop or tamper the data between the FPGA and peripherals. We assume the hardcore system and Direct Memory Access (DMA) masters cannot dump the content in the FPGA. All of the assumptions are realistic in that they are the standard configurations on most commercial systems [63]. Even though recent research [34] discovered dumping content from the FPGA is possible due to some implementation bugs, it was not intended to be a feature. We assume secure or trusted boot, which loads the software images onto the hardcore system and the bitstream and FIRMWARE onto the FPGA after integrity checks at boot-time. We assume the cryptographic algorithms are secure.

Threat Model. We assume the adversary can compromise and take full control of the software system of the hardcore system at run-time, which means the user developed Untrusted Applications (UA), the kernel, and even privileged software in the secure world of a TrustZone-based system can be malicious. The compromised software on the hardcore system can send arbitrary data to the FIRMWARE and SSAs via shared DRAM regions and also to the enclave hardware pins, such as interrupts. Therefore, the FIRMWARE and SSA in an enclave can be compromised at run-time. Attackers can also perform cold-boot attacks to dump the content in DRAM.

Deployment and Key Management Models. BYOTEE does not aim to address any specific deployment and key management model, as it is related to the user and application's policies and needs. Instead, BYOTEE provides the mechanisms required to build a secure system.

We nevertheless discuss two examples for local and cloud deployment and key management. At the cryptographic layer, we assume each device has a device key, e.g., AES (k_d), RSA (sk_d, pk_d), which are used to encrypt/decrypt and sign/verify software images, bitstream, and the FIRMWARE. The device keys are unique to each device and are programmed in the eFUSE or BBRAM using hardware interfaces with physical access. We assume a developer can be identified by a developer key, e.g., RSA (sk_u, pk_u), which the developer uses to encrypt and sign the SSAs, and the FIRMWARE uses it to decrypt and verify the SSAs. Developer public keys are either embedded into the FIRMWARE at the development stage or loaded into the FIRMWARE at run-time.

An example solution for local deployment is that the device owner programs the device keys into the one-time programmable eFUSE, e.g., through the JTAG interface on a development board [93]. The FIRMWARE developer embeds the allowed the developers' public keys inside the FIRMWARE at the development stage. If SSA developers need to register or update a key, they will send the key public keys to the FIRMWARE developers.

An example solution for cloud deployment is that the manufacturer programs the device keys into the one-time programmable eFUSE same as Intel creates root keys for any SGX-enabled CPUs. The FIRMWARE developers generate FIRMWARE and bitstream, and they also encrypt them using the public part of the device key and embed developers' public keys in the FIRMWARE. The protected FIRMWARE and bitstream along with the UA are sent to the untrusted cloud provider. The developers' keys update process is similar to the process for local deployment.

IV. BYOTEE ARCHITECTURE

In this section, we first present the security and functional design goals of BYOTEE followed by an overview of the BYOTEE architecture and workflow. We then illustrate hardware TCB customization, bootstrapping trust in enclaves, secure execution of SSA, and remote attestation mechanisms.

A. Design Goals

BYOTEE provides physically isolated execution environments on-demand, which even hardware debuggers and DMA-enabled devices cannot access. With the help of BYOTEE, users can use the exact and even formally verified hardware and software needed for their applications. BYOTEE has the following security and functional design goals:

G1. General-purpose execution environments. BYOTEE should provide general-purpose execution environments similar to SGX and TrustZone, not application-specific accelerators. SSAs can be implemented in any language as long as they can be linked against the FIRMWARE.

G2. Multiple isolated execution environments. BYOTEE should provide multiple execution environments at the same time and guarantee the secrecy and integrity of the SSA running inside each.

G3. Physical execution isolation. BYOTEE should provide dedicated CPUs for execution environments, and all hardware resources for enclaves are physically isolated from the REE and each other. With physical isolation, BYOTEE mitigates the side-channel attacks that are prevalent in CPU-sharing TEEs, such as SGX and TrustZone.

G4. Customizable hardware TCB. The hardware TCB of each enclave should be customizable, allowing for a minimal TCB that only includes the hardware, e.g., peripherals, necessary for the SSA running inside and excludes other hardware on the system. Note that formally verifying the customized hardware [26], [74] is beyond the scope of BYOTEE.

G5. Isolated path between SSA and peripherals. An enclave should isolate the communication path between the SSA inside and peripherals from the hardcore system and other enclaves, preventing software-based eavesdropping and tampering.
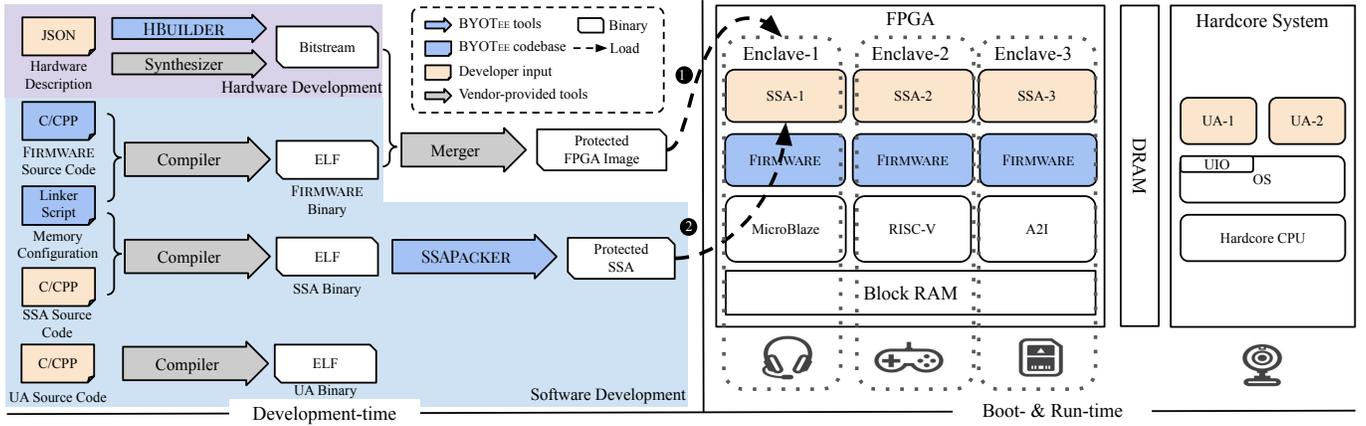
Fig. 2: The architecture and workflow of the BYOTEE framework at development-, boot- and run-time. During development, BYOTEE and vendor-provided tools are used to generate the protected FPGA image and protected SSAs, which are loaded onto the FPGA (❶) and enclaves (❷), respectively. In this run-time architecture example, three enclaves with different hardware configurations, including softcore CPUs and peripherals, are presented. Untrusted Applications (UA) access the shared DRAM region through a userspace I/O interface (UIO).

G6. Enclave-to-Hardcore System and Inter-enclave communication. The SSA in an enclave should be able to communicate with software modules on the hardcore system and other enclaves. The inter-enclave communication should be isolated from the hardcore system and other non-participating enclaves.

G7. Minimal software TCB. The firmware serving an SSA should only include the housekeeping libraries and drivers that are necessary for the SSA execution and exclude other software.

G8. Remote attestation mechanisms. BYOTEE should provide mechanisms to support sophisticated protocols to attest the integrity of an enclave's hardware and software stacks, including boot loaders, bitstream, FIRMWARE, SSAs and their inputs and outputs to remote verifiers.

G9. Easy to use. BYOTEE should be easy to use, especially for the software developers who do not have hardware programming experience. As a rule of thumb, developing a BYOTEE SSA should not take significantly more time and effort than developing a Linux application with the same functionality.

B. BYOTEE Overview

Figure 2 presents an overview of the architecture and workflow of the BYOTEE framework. The BYOTEE tools and codebase mainly include the HARDWAREBUILDER, FIRMWARE, and SSAPACKER. During the development stage, the HARDWAREBUILDER generates synthesizer commands, e.g., Tcl, based on the SSA's needs specified in the developer's hardware description JSON input. Then, the SoC FPGA vendor-provided synthesizer, e.g., Xilinx Vivado [84], Intel Quartus Prime [48], generates the bitstream file using the synthesizer commands. The developer can customize the FIRMWARE by only including the needed source code and writing the SSA source codes, which are compiled with the corresponding compiler for the softcore CPU, e.g., `mb-gcc`

for MicroBlaze. The bitstream and FIRMWARE binary are encrypted, signed, and packed by the vendor-provided merger, e.g., `UpdateMEM` from Xilinx, into a protected FPGA image. The SSA binary is encrypted, signed, and packed by the SSAPACKER into a protected SSA. At boot-time, the bitstream is loaded onto the FPGA. As a result, multiple enclaves are created, and the corresponding FIRMWARE starts running. Then, an untrusted application can trigger the loading of a protected SSA into an enclave.

BYOTEE meets *G1*, *G2*, *G3*, *G4*, and *G5* by configuring the FPGA to build enclaves. BYOTEE constructs enclaves with softcore CPUs, which provide general-purpose computing environments (*G1*). The FIRMWARE includes the standard C libraries and connected peripheral HAL libraries for SSAs to invoke at run-time. Each enclave has its own set of hardware (*G2*), including softcore CPU, e.g., MicroBlaze, UltraSPARC, etc., Block RAM, and peripherals. With FPGA routing, these hardware resources within an enclave are connected together but isolated from the hardcore system and other enclaves (*G3*). The softcore CPU in an enclave is not time-shared with the hardcore system and other enclaves, mitigating the cache side-channel attacks (*G3*). No additional hardware modules, such as debuggers, can be connected to an enclave unless it is explicitly specified by the developer (*G4*). The connections among these resources are also physically isolated from the hardcore system and other enclaves, preventing eavesdropping and tampering (*G5*).

Enclaves use interrupts on softcore CPUs and shared physical memory regions on the DRAM to communicate with the hardcore system, whereas enclaves use interrupts and shared regions on the BRAM to communicate with each other. Since a shared BRAM region is only mapped in the address spaces of the participating enclaves, it is isolated from the hardcore system and other enclaves (*G6*). BYOTEE includes FIRMWARE, which can be customized and only consists of
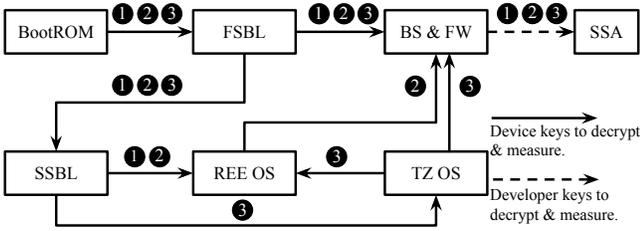
Fig. 3: Example static and dynamic trust bootstrap paths on Zynq SoC. ❶ static trust bootstrap. ❷ dynamic trust bootstrap without TrustZone. ❸ dynamic trust bootstrap with TrustZone.

libraries, a HAL for the needed peripherals, and a loader for the SSA (*G7*). The FIRMWARE also provides softcore CPU interrupt-based remote attestation mechanisms for proving the integrity of enclaves (*G8*). BYOTEE provides an easy-to-use toolchain for developers to focus on SSA development, increasing the usability of BYOTEE and decreasing the chances of developer-induced misconfigurations (*G9*).

C. Customizing Hardware TCB for Enclaves

To customize the hardware TCB, the developer designs one or more enclaves using a hardware description language, e.g., Verilog or VHDL. The output is a bitstream file that configures the FPGA. To facilitate this step, BYOTEE has a component, named HARDWAREBUILDER, which takes developer-specified hardware description in JSON format as input (See an example in §V-B), allocates hardware resources, and outputs a script, e.g., in Tcl format, that can be processed by a synthesis tool to generate the bitstream. Each enclave's hardware description includes but is not limited to: i) a softcore CPU and its configurations, e.g., clock frequency, cache size, etc; ii) a corresponding debug IP to enable software debugging on the softcore CPU; iii) its main BRAM memory address and size; iv) the address and size of the shared DRAM with the hardcore system; v) the address and size of the shared BRAM with other enclaves; and vi) connected peripherals. The HARDWAREBUILDER assigns a contiguous address space of the BRAM to each enclave and connects the hardware components automatically.

D. Bootstrapping Trust in Enclaves

Because the details of trust bootstrap are related to the specific deployment model and policies, BYOTEE provides the mechanisms to build trust bootstrap paths. BYOTEE supports FPGAs that can be configured at boot time and/or reconfigured at run-time by providing trust bootstrap in enclaves both statically and dynamically. The static trust bootstrap builds on secure or trusted boot, whereas the dynamic trust bootstrap relies on some dynamic root-of-trust modules at run-time.

Without loss of generality, Figure 3 shows three example paths for static and dynamic trust bootstrap on the Xilinx Zynq SoC with Arm hardcore CPUs. In the static bootstrap case ❶, BYOTEE relies on secure or trusted boot to launch enclaves. BootROM, which is the root of trust for measurement, first

decrypts, measures, i.e., $m_1 = H(FSBL)$, and executes the FSBL using the device keys. The FSBL initializes the hardcore system by decrypting, measuring, and executing the SSBL using the device keys. A measurement $m_2 = H(m_1 || SSBL)$ is generated in this step. The FSBL also initializes the FPGA by decrypting, measuring, and configuring the FPGA using the device keys, after which bitstream is programmed on the FPGA, and the FIRMWARE starts execution. A measurement $m_3 = H(m_2 || BS || FW)$ is generated by the FSBL and placed on the shared DRAM region with the enclave for the future use of attestation report generation, which we discuss in §IV-F. The FIRMWARE uses the developer keys, e.g., k_u and/or pk_u, to decrypt and measure the SSA. Allowed developer keys are embedded in the FIRMWARE at the development stage. Because the FIRMWARE is encrypted at rest and only decrypted on the BRAM, the developer keys are secure.

In the dynamic case, software running on the hardcore system of a Xilinx Zynq device can use the device configuration (DevC) or Processor Configuration Access Port (PCAP) interfaces to reconfigure the FPGA. Therefore, the software module that reconfigures the FPGA needs to access the device key and must be trusted. In the case that TrustZone is not available or used ❷, the trusted software module can be part of the REE kernel, which uses the device keys to decrypt and measure the bitstream and FIRMWARE. If TrustZone is available ❸, the trust software module can be a kernel module inside the TrustZone secure world operating system.

E. Executing SSAs in Enclaves

To create an SSA, the developer links the source code against the FIRMWARE. After the launching of an enclave, the FIRMWARE initializes the softcore CPU and other components, then it waits for requests from the hardcore system. Both the FIRMWARE and SSA use a shared DRAM region in the SSA Execution Block (SEB) format as shown in Figure 4, to have two-way data transmissions with UA on the hardcore system. To initiate the transmission from the hardcore system to an enclave, UA on the hardcore system raises interrupts on the enclave's softcore CPU, which are handled by the FIRMWARE. BYOTEE defines three service primitives through the LdExec* interrupts: i) load and execute an SSA (LdExec); ii) load and execute an SSA with pre-execution attestation (LdExecPreAtt); iii) load and execute an SSA with post-execution attestation (LdExecPostAtt). The softcore CPU interrupts can be implemented as GPIO interrupts in the enclave and memory-mapped to a DRAM address for the UA to access. In this subsection, we focus on LdExec, and the other two primitives are discussed in §IV-F.

To execute an SSA on an enclave, the untrusted application first fills data into the SEB and raises a LdExec* interrupt. As shown in Figure 4, a SEB has regions for the encrypted and signed SSA (SSA*), input data for the SSA, output data from the SSA, a challenge $Chal$ from a remote verifier, a pre-execution ($PreExecAtt$), a post-execution attestation measurement ($PostExecAtt$) and other data. When the FIRMWARE receives a LdExec* interrupt, it
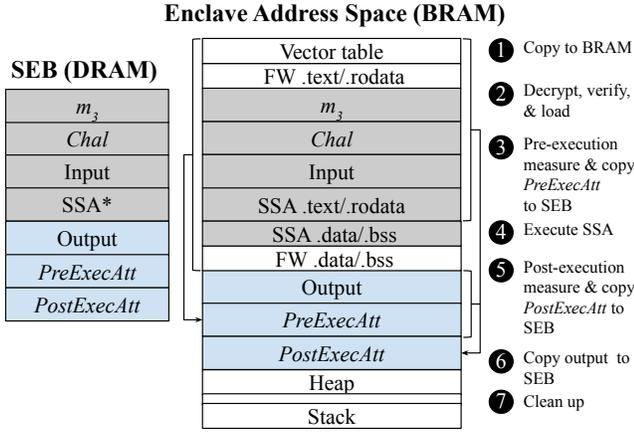
Fig. 4: SSA Execution Block (SEB) layout, simplified enclave address space layout, and steps in executing an SSA.

copies SSA* and optionally $Chal$ and input data in the SEB from DRAM to its own BRAM ❶. The FIRMWARE can also disable LdExec* interrupts after the data is copied. Note that it is critical for the FIRMWARE to perform measurement on the BRAM since the DRAM can be changed asynchronously by the hardcore system. The FIRMWARE then decrypts and verifies the encrypted SSA* using the corresponding developer's keys ❷. Upon the successful verification of the SSA's integrity, the FIRMWARE loads sections of the decrypted SSA to the right locations and gives the control to the SSA ❺. If there is an output, the SSA writes it in the output region on the BRAM, and yields the control of the softcore CPU back to the FIRMWARE. The FIRMWARE copies the output from the BRAM to the DRAM ❻. Finally, the FIRMWARE cleans up all the input, output, and SSA-related regions on the BRAM and awaits new requests from the hardcore system ❼. While SSAs can execute concurrently on their own enclaves respectively, the FIRMWARE also supports executing multiple SSAs sequentially or the same SSA multiple times on the same enclave without reconfiguring the FPGA but just re-initializing the enclave, e.g., flush the cache, clean up the BRAM ❼, etc.

F. Pre-execution and Post-execution Attestation Mechanisms

BYOTEE provides two attestation mechanisms, namely pre-execution and post-execution attestations, as shown in Figure 4. The former extends the traditional remote attestation of code and input integrity with bitstream, whereas the latter extends the output data integrity attestation [16]. Note that BYOTEE only provides the mechanism for attestation, which can support sophisticated attestation protocols. With the help of trust bootstrapping discussed in §IV-D, the measurement mechanism not only captures the identity of the loaded SSA but also the bitstream, FIRMWARE and programs, e.g., FSBL and SSBL, before it.

In pre-execution attestation, a verifier sends a cryptographic nonce as $Chal$, which is copied to the BRAM by the FIRMWARE ❶. After loading the SSA sections to the right addresses, the FIRMWARE computes a measurement

$PreExecAtt$ on the vector table, FIRMWARE code and data, m_3, $Chal$, input data and SSA sections ❸, and copies the measurement to the DRAM. Depending on scenarios and attestation protocol details, the FIRMWARE can use a device key, developer key, or other shared keys to compute the measurement. In post-execution attestation, the $Chal$ from Vrf is also copied to the BRAM by the FIRMWARE ❶. After the SSA finishes execution ❹, the FIRMWARE computes a measurement $PostExecAtt$ on the vector table, FIRMWARE code and read-only data, m_3, $Chal$, input data, output data generated by the SSA, SSA's read-only sections and $PreExecAtt$, and copies the measurement to the DRAM ❺.

G. Multiple Inputs to SSA

In the case that the UA on the hardcore system needs to continuously send input data to the SSA, e.g., not all input data is available at the beginning, the size of input in SEB is not big enough, etc., the UA writes the newly available input data in the input region inside the SEB, and it can use two mechanisms to notify the FIRMWARE and SSA that new data is available. The first mechanism works for softcore CPUs that support priority interrupts. On such systems, BYOTEE defines a NewData interrupt, which UA can raise. The NewData interrupt has a low priority so that it cannot interrupt the execution of the SSA. Only after the SSA finishes execution and yields the control back to the FIRMWARE, the FIRMWARE can copy and measure the input data from the DRAM to the BRAM, and gives the control to the SSA again. On softcore CPUs without priority interrupts, the FIRMWARE uses global variables to indicate whether new data is available in the input region to synchronize with the SSAs on the enclaves.

H. Optional Multiple Protected SSA Sessions

As an option, an enclave can also interleave the execution of multiple eligible SSAs with proper hardware re-initialization, e.g., flush cache, reset all memory, etc. To this end, BYOTEE defines two interrupt-based service primitives: i) suspend and export the SSA state (SusExp); ii) restore and execute a saved and encrypted SSA state (ReExec). When a SusExp interrupt is raised, the FIRMWARE copies the SSA context, e.g., general and system registers, onto the BRAM. Then, the FIRMWARE uses the developer key to encrypt and sign the saved SSA context and all of the SSA's writable memory regions, e.g., stack, .data, .bss, etc. The encrypted blob is placed in the SEB output region for the UA to retrieve, after which the FIRMWARE cleans up BRAM and awaits new requests. When a ReExec interrupt is raised, the FIRMWARE retrieves an encrypted blob from the SEB input region. Upon a successful signature verification, the FIRMWARE loads the decrypted memory contents to the right locations, restores the registers, and resumes the SSA execution.

V. BYOTEE APPLICATIONS AND DEVELOPER'S PERSPECTIVE

In this section, we first use multiple SSA examples to demonstrate how BYOTEE can secure real-world applications

in several classes. Then, we discuss how developers can easily develop and deploy enclave hardware, SSAs, and UAs using the BYOTEE toolchain.

A. BYOTEE *Applications*

Computational Applications. Computational applications take input from the hardcore system or other enclaves, perform the intended computational operations, and send the outputs back. They represent computational tasks, such as encryption, decryption, machine learning-based classification, etc., that do not need peripherals. BYOTEE protects such applications from code and data disclosure, memory corruption, and cache side-channel attacks from the hardcore system and other enclaves at run-time. We implement an AES accelerator SSA (SSA-1) as an example for computational applications. To use SSA-1, a UA places the plaintext or ciphertext in the SEB and notifies the SSA. When the encryption or decryption is finished, SSA-1 places the outputs on the SEB.

Peripheral Interacting Applications. Peripheral interacting applications use some peripherals, but they do not communicate with other SSAs or the hardcore system. An example usage is cyber-physical applications that read from sensors, make local decisions, and control an actuator. Besides the attacks BYOTEE protects the computational applications from, BYOTEE also protects the paths between the SSA and peripherals from attacks for these applications. We develop an LED toggler SSA (SSA-2) that uses a button and an LED. When the button is pressed, SSA-2 toggles the color of the LED. Both the button and LED are connected to the enclave of SSA-2 only and cannot be accessed by the hardcore system or other enclaves.

Peripheral and Hardcore System Interacting Applications. Peripheral and hardcore system interacting applications do not only control some peripherals but also continuously interacts with a UA. For demonstration, we develop a music player system with digital rights management that guarantees the confidentiality, integrity, and authenticity of songs for artists and recording studios. This means i) songs cannot be digitally disclosed, ii) songs cannot be modified, and iii) only songs that were protected can be played.

To this end, the music player system has three components: i) a trusted song protector (in Python with 160 SLOC), which is an offline component to encrypt and sign a song file (WAV format); ii) a UA (in C with 695 SLOC) running on the hardcore system, which provides a user interface to play, pause, resume, and stop a protected song. The UA awaits for the user's commands, reads protected songs from storage, e.g., SD card, and sends them to the song playing SSA. Because the protected song file is big (e.g., an original 77 seconds, 48KHz, and a single channel WAV file is around 33MB. The protected song file is several hundred bytes bigger.), the UA needs to continuously read the protected song file data from the storage and send it to the SSA; iii) a song playing SSA (SSA-3) that authenticates, decrypts, and plays a song by sending the plaintext data of it to a hardware audio module. The hardware audio module is only connected to the enclave running SSA-3, so the hardcore system and other enclaves cannot access it.

We emphasize that any software solution that solely trusts SGX or TrustZone cannot meet the security requirements of this music player system because: i) there is no isolated or trusted I/O path between an SGX enclave and the hardware audio module; hence a malicious REE OS can breach the confidentiality, integrity, and authenticity of a song. Some solutions, such as SGXIO [81], attempt to address this issue but they need to add additional hardware, e.g., the hypervisor, into the TCB; ii) a TrustZone application must decrypt the song in DRAM before sending it to play; hence, vulnerable to cold-boot attacks.

Distributed Applications. A distributed application consists of multiple inter-communicating SSAs running on different enclaves at the same time. The SSAs communicate through a shared BRAM region. BYOTEE not only protects each of the SSAs but also their communications from the hardcore system and other enclaves. For demonstration, we develop an application that processes data in sequence with two SSAs. SSA-1 first receives data from a UA, decrypts the data, outputs to the shared BRAM instead of DRAM, and the second SSA (SSA-4) takes the output of SSA-1 and performs a SHA512-HMAC signature verification.

B. *Developer's Perspective*

Creating Enclave Hardware. A developer can use the HARDWAREBUILDER to design and create the hardware comprising of one or multiple enclaves. Listing 1 shows an example hardware description in JSON format of three enclaves. Enclave-1 has a 32-bit MicroBlaze softcore CPU [89]. Enclave-2 has a 32-bit VexRisc softcore CPU [15], Enclave-3 uses a 64-bit A2I softcore CPU [1]. The softcore CPUs of Enclave-2 and Enclave-3 have FPUs, instruction, and data caches. A2I softcore of Enclave-3 has MMU enabled with a page size of 4KB. Debugging is enabled on the softcore CPU of Enclave-1 only, for which HARDWAREBUILDER inserts a debugging module to help the developer debug and trace the SSA on this enclave. A DRAM region is reserved for the SEB of each enclave, respectively. A UART peripheral is only connected to the Enclave-1 and cannot be accessed by the hardcore system or the other enclave. Additionally, a GPIO peripheral is connected to both the hardcore system and Enclave-2 but cannot be accessed by Enclave-1 or Enclave-3. Each enclave shares a DRAM region with the hardcore system for 2-way enclave-to-hardcore System communication. Enclave-1 and Enclave-3 can also use the shared BRAM region to communicate.

The developer uses HARDWAREBUILDER to generate hardware configurations, which outputs the Tcl scripts containing the synthesizer commands. The -d parameter specifies the JSON configuration file, and -o defines the output Tcl path:

```
hardwareBuilder.py -d <CONFIG_JSON> -o <TCL>
```

Then, the developer invokes the synthesizer tool with the Tcl script as input. The -n parameter specifies the name of

```
1  {"Enclaves": [
2      {"Name": "Enclave-1",
3       "Processor": {"Type": "MicroBlaze 32bit",
4        "Debugging": "Enabled"},
5       "Memory Size": "512KB",
6       "Shared DRAM SEB": {
7        "Base": "0x20000000", "Size": "2MB"}},
8      {"Name": "Enclave-2",
9       "Processor": {"Type": "VexRisc 32-bit",
10       "Data Cache": "16KB",
11       "Instruction Cache": "16KB", "FPU": "F32",
12       "Debugging": "Disabled"},
13      "Memory Size": "32MB",
14      "Shared DRAM SEB": {
15       "Base": "0x20000800", "Size": "128MB"}},
16     {"Name": "Enclave-3",
17      "Processor": {"Type": "A2I 64bit",
18       "Data Cache": "64KB",
19       "Instruction Cache": "64KB",
20       "MMU": "Enabled", "MMU Page Size": "4KB",
21       "FPU": "AXU", "Debugging": "Disabled"},
22      "Memory Size": "64MB",
23      "Shared DRAM SEB": {
24       "Base": "0x20020800", "Size": "256MB"}}],
25  "Peripherals": [
26      {"Type": "AXI Gpio",
27       "Board Interface": "Btns 2bits",
28       "Access": ["Hardcore system", "Enclave-2"]},
29      {"Type": "Uart Lite 8bit",
30       "Baud Rate": "115200",
31       "Access": ["Enclave-1"]},
32      {"Type": "Dual Port BRAM Generator",
33       "Base Address": "0x1F0000", "Size": "2MB",
34       "Access": ["Enclave-1", "Enclave-3"]}]}
```

Listing 1: An example hardware description defining three enclaves in JSON format. Each enclave has its own customized softcore CPU (processor model, cache, FPU, etc.), peripheral connections, memory, etc.

the hardware project, and the `bf` parameter specifies the mode of operation, which includes generating bitstream, combining bitstream with FIRMWARE, etc. The final output of the HARD-WAREBUILDER is a bitstream file specified by `-o`:

```
createFPGAImage -d <TCL> -n <PROJ_NAME> -bf <
    BUILD_FLAG> -o <FPGA_IMAGE>
```

Creating Boot Images. After the HARDWAREBUILDER, the developer uses the boot loader creation tool with the developer-defined boot image format, e.g., `.bif`, and the protected FPGA image to create a deployable binary file.

```
createBootImage <SYSTEM_BIF> <FPGA_IMAGE> -o <
    BYOTee_BIN>
```

Creating SSAs and UAs. BYOTEE FIRMWARE is primarily developed in C, but software modules developed in any language that can be linked against the FIRMWARE can be included in an SSA. The SSAs, which have their own `main` functions with a declaration of `int main() __attribute__ ((section (".text.ssa_entry")))`, are developed as separate applications from the FIRMWARE. In our prototype implementation, SSAs are statically linked against the FIRMWARE with the linker script BYOTEE provides,

which reserves physical memory regions for the SSAs' code and data sections.

Not all libc functions are available for the SSAs to use. For example, `printf` may is not available since `stdout` is not defined in the FIRMWARE for a specific FPGA system. To move data among DRAM, BRAM, and peripheral memories, system-specific underlying mechanisms will be used. The FIRMWARE provides a HAL with interfaces like `BYOT_MemCpy` to replace the libc `memcpy`. The UAs execute as unprivileged applications on the operating system, e.g., PetaLinux for Xilinx devices, of the hardcore system. The UAs use a UIO interface to communicate with the FIRMWARE and SSA running on the FPGA.

The developer uses the SSAPACKER to generate protected SSA binaries:

```
SSAPACKER -d <SSA_BIN> -o <PROTECTED_SSA>
```

BYOTEE also includes a tool to put the boot loader, PetaLinux image, UA, protected FPGA image, and protected SSA binaries on the SD card for the deployment and debugging on Xilinx SoC FPGA systems:

```
deploySoC <SD_DEVICE> <BYOTee_BIN> <FPGA_IMAGE> <
    PROTECTED_SSA> <UA> <IMAGE.ub>
```

VI. SECURITY ANALYSIS AND LIMITATIONS

In this section, we conduct an informal security analysis of BYOTEE, in which we discuss the attacks BYOTEE can and cannot defend from.

A. Malicious and Compromised Hardcore System Software

Read and Write to Enclaves. The hardcore system software, including the operating system and UA, is not a part of the TCB in BYOTEE. Even if the hardcore system software is compromised at run-time, the attacker cannot access the data on/from enclave hardware resources, including BRAM and peripherals, because they are in the isolated address space of the target enclave. The attacker cannot breach the confidentiality of SSA code and data as well, because they are encrypted at build time. The enclave-hardcore system 2-way communication is based on interrupts and the shared DRAM. Malicious hardcore system software can raise the interrupt to the enclave to carry out a DoS attack. Utilizing priority interrupts in sophisticated softcore processors, BYOTEE can prevent these attacks from the hardcore system side.

Reconfiguring FPGA. The secure or trusted boot, i.e., static root of trust, process of BYOTEE ensures the secure loading of the bitstream to configure the FPGA at boot time. An alternative dynamic root of trust approach ensures a compromised hardcore system software cannot load an attacker-controlled bitstream to reconfigure the FPGA if the hardware supports run-time reconfiguration.

B. Compromised FIRMWARE and SSAs

If the software layer of an enclave, i.e., FIRMWARE and SSA, is compromised at run-time by untrusted input sent by the hardcore system software, it can disclose information in

the compromised enclave address space, including data on the BRAM and data from the connected peripherals. But it cannot read data from the BRAM or peripherals of other enclaves, since they are in different address spaces and cannot be seen by the compromised software. Therefore, the attack is confined within the compromised enclave.

C. Hardware IPs and Malicious Peripherals

Malicious hardware IPs cannot be loaded since a bitstream is signed by a trusted developer and verified before loading. Even if peripherals are malicious and send out rogue DMA requests to access sensitive memory regions, they are confined in the enclave they are assigned to. Therefore, a malicious peripheral can only cause limited damages.

D. Cold-boot Attack

Cold-boot attacks rely on the observation that the contents in memory are not immediately erased after power is lost. While cold-boot attacks on DRAM even at room temperature are proven very effective [40], attacks on SRAM without external power sources are less feasible [66]. Most data BYOTEE stores on the DRAM is either encrypted or does not need to be protected. For instance, even if the SEB is located on the DRAM and subject to cold-boot attacks, the SSA*, which includes developer keys, is encrypted. Obviously, *Chal*, *PreExecAtt*, *PostExecAtt* do not need to be protected. It is, however, possible to dump the input and output fields of the SEB using cold-boot attacks on DRAM. Other sensitive data, such as developer keys, plaintext SSA, program states, are placed on an enclave's BRAM. Cold-boot attacks on BRAM are difficult because: i) the BRAM cells are hardware initialized during FPGA configuration in many SoC FPGA systems [90]; ii) even without initialization, the contents in BRAM decays faster [66]; iii) BRAM is embedded on-chip and cannot be physically taken out, so attackers have to bypass software protections to dump its content.

E. Side-channel Attacks

Cache side-channel. Because the CPU is time-shared between the REE and TEE in SGX and TrustZone, cache side-channel attacks are effective [25], [30], [97], [99]. In BYOTEE, the REE on the hardcore system side and enclaves do not time-share any CPU resources; hence, there is no cache side-channel between the REE and TEEs.

Power side-channel. In FPGA-based remote power side-channel attacks, the attacker builds an on-chip ring oscillators-based power monitor to conduct power analysis attacks on other modules on the same FPGA or a CPU on the same SoC [100]. BYOTEE cannot directly mitigate these attacks but can them by only loading authenticated and trusted enclave bitstreams that do not have a power monitor.

Other side-channels. When multiple enclaves reside on the same SoC FPGA, they share FPGA hardware resources. Therefore, it is possible to conduct other sharing-based side-channel attacks, such as FPGA long wire-based attacks [37], [67]. Similar to power side-channel attacks, BYOTEE cannot prevent these attacks directly.

VII. IMPLEMENTATION AND EVALUATION

In this section, we present an implementation of the BYOTEE framework for the Xilinx SoC FPGA and evaluate it on a low-end Digilent Cora Z7-07S development board ($\approx$\$130) with a Zynq-7000 FPGA.

A. Experiment Environment

The Cora Z7-07S board has a single-core 667MHz Arm Cortex-A9 processor with 512MB DDR3 memory, 32KB L1 cache, 512KB L2 cache and a Xilinx Zynq-7000 FPGA. As shown in Table I, the Zynq-7000 FPGA has 3,600 logic cells, 14,400 LUTs, 6,000 LUTRAM, 28,800 flip-flops, a 225KB BRAM, 66 Digital Signal Processing (DSP) slices, and 100 IOBs. The development board also has an SPI header, two push-buttons, two RGB LEDs, a microSD card slot, two Pmod connectors, etc. We connect a Pmod I2S2 stereo audio input and output device [12] to the board for SSA-3 evaluation. Figure 7 (in Appendix) shows the top and bottom view of the board with the connected audio device. We create two partitions, namely `boot` and `root`, on an SD card and use the Xilinx `bootgen` tool to generate device boot images [95]. `Bootgen` stitches a stock FSBL for the hardcore system and the protected FPGA image together to create a binary boot file. The boot file, U-Boot as the SSBL for the hardcore system, and a PetaLinux image for the hardcore system are stored in the `boot` partition, whereas the protected SSAs, UAs, and other application files are stored in the `root` partition.

B. BYOTEE Implementation

We implemented the BYOTEE infrastructure and toolchain, which include HARDWAREBUILDER, SSAPACKER, and FIRMWARE, for the Xilinx SoC FPGA. The HARDWARE-BUILDER was developed in Python (2.5K SLOC). The SS-APACKER include Python (63 SLOC) and C code (420 SLOC). The FIRMWARE was developed in C and has an SSA loader and cleaner (1.1K SLOC), an attestation module (333 SLOC), an interrupt initialization and handling module (101 SLOC), and a linker script (212 lines). The FIRMWARE is linked against the vendor-provided HAL (7.9K SLOC) and libraries, e.g., libc (1.2MB), etc. The FIRMWARE, especially the HAL, can be customized to reflect an SSA's needs. Our implementation of the SSA loader uses AES-256 for SSA encryption and SHA512-HMAC to protect the integrity and authenticity of SSAs. We use the BLAKE2 [9] hash algorithm to implement the pre-execution- and post-execution-attestations of SSA applications in the attestation module. On the hardcore system side, a userspace I/O interface is used for the UAs to access the shared DRAM regions between the hardcore system and FPGA.

C. Enclaves for the Example SSAs

We specified the hardware description for each example SSA in §V-A and used the HARDWAREBUILDER and synthesizer to generate its enclave bitstream, e.g., Enclave-1 for SSA-1. All the enclaves are configured with a 32-bit Microblaze CPU (version 10.0, 100MHz, no instruction/data cache, no

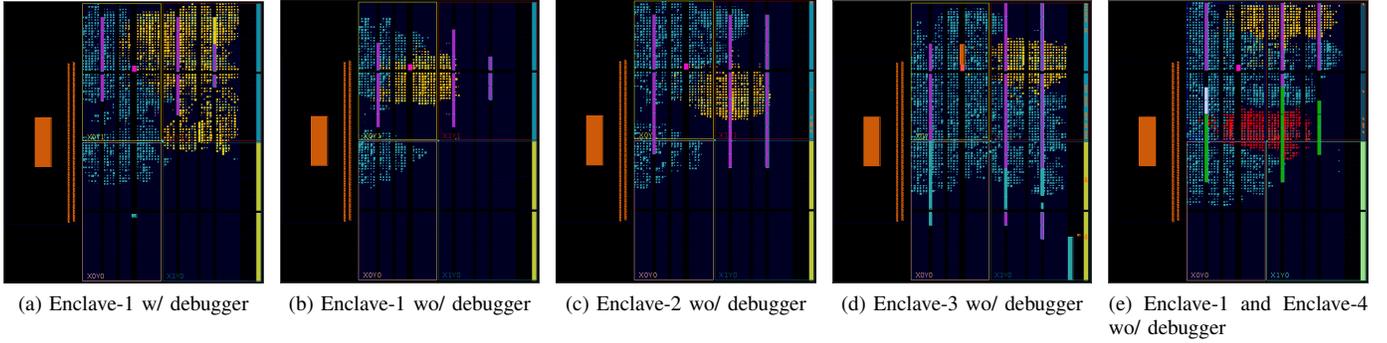| (a) Enclave-1 w/ debugger | (b) Enclave-1 wo/ debugger | (c) Enclave-2 wo/ debugger | (d) Enclave-3 wo/ debugger | (e) Enclave-1 and Enclave-4 wo/ debugger |

Fig. 5: Resource footprints of the enclaves with MicroBlaze softcore CPUs for the example SSAs on the Cora Z7-07S. The yellow and red portions represent the MicroBlaze softcore CPU cells. The purple portion represents the cells of BRAMs. The pink rectangle represents the I/O ports. In (d), the rectangle on top of the I/O ports represents an analog to digital conversion module. The blue portions represent all other IPs, such as different peripherals, debugging modules, interconnects.

TABLE II: Hardware TCB size and resource utilization of enclaves for the example SSAs on Cora Z7-07S

| | Enclave-1 | | Enclave-2 | | Enclave-3 | | Enclave-4 | |
Resource	w/ debugger	w/o debugger	w/ debugger	w/o debugger	w/ debugger	w/o debugger	w/ debugger	w/o debugger
LUT	5,255 (36.5%)	2,232 (15.5%)	6,385 (44.3%)	3,291 (22.9%)	10,781 (74.9%)	6,778 (47.1%)	5,302 (36.8%)	3,130 (21.7%)
LUTRAM	419 (7.0%)	211 (3.5%)	507 (8.5%)	282 (4.7%)	725 (12.1%)	427 (7.1%)	319 (5.3%)	145 (2.4%)
Flip-flop	5,245 (18.2%)	2,259 (7.8%)	6,759 (23.5%)	37,64 (13.1%)	11,363 (39.5%)	7,721 (26.8%)	5,497 (19.1%)	3,014 (10.5%)
BRAM	18 (36.0%)	16 (32.0%)	34 (68.0%)	32 (64.0%)	48 (95.0%)	45.50 (91.0%)	28 (56.0%)	26 (52.0%)
DSP	3 (4.5%)	0 (0.0%)	3 (4.5%)	0 (0.0%)	3 (4.5%)	0 (0.0%)	3 (4.5%)	0 (0.0%)
IOB	0 (0.0%)	0 (0.0%)	6 (6.0%)	6 (6.0%)	28 (28.0%)	28 (28.0%)	0 (0.0%)	0 (0.0%)

TABLE III: Size of the example SSAs' software TCB

| | SSA | | Corresponding FIRMWARE | | | | |
	SLOC	Bytes	SLOC	.text	.data	.bss	Total
SSA-1	717	12,892	3,143	27,296	3,236	448	30,532
SSA-2	346	2,868	3,532	30,748	2,800	440	33,988
SSA-3	1,029	20,380	9,698	57,142	4,308	635	62,085
SSA-4	622	31,088	3,235	28,377	3,608	528	35,748

FPU). The Enclave-1, Enclave-2, Enclave-3 have a 128KB BRAM, whereas Enclave-4 has a 32KB BRAM as their main memory. The peripherals that belong to an enclave are connected through a dedicated AXI Interconnect IP. Figure 5 shows the footprints of the four hardware designs on the Z7-07S device. These figures demonstrate the configurable nature of the BYOTEE hardware TCB and the physical isolation of the enclaves from each other and from the hardcore processor. Figure 5(a) shows the hardware design with a debugger, which developers can use to debug the SSA and FIRMWARE on the softcore, whereas all other designs do not have a debugger for the minimum hardware TCB. The orange portion of the footprint represents cells of the ZYNQ7 processing system, whereas yellow represents softcore cells. The purple straight lines represent the cells of BRAMs. The pink rectangle represents the I/O ports, and in Figure 5(d) the rectangle on top of the I/O ports represents an analog to digital conversion module. Figure 5(e) represents hardware implementation of two enclaves, namely Enclave-1 (yellow) and Enclave-4 (red), to run SSA-1 and SSA-4, respectively. There are three types of BRAM in this design. The purple straight lines represent

the contiguous memory of Enclave-1 (128KB), and the green straight lines represent the contiguous memory of Enclave-4 (32KB). The ash straight lines represent the shared BRAM (8KB) between Enclave-1 and Enclave-4. The blue portions in Figure 5 represent the rest of the IPs used in the hardware design. These include interconnects, interrupt controllers, etc.

Table II presents each enclave's hardware TCB and resource utilization on the Cora Z7-07S board with and without a debugger IP. As the table shows, the debugger IP significantly increases the resource utilization of an enclave as it uses three DSP slices, two BRAMs, and many other resources. Since SSA-1 and SSA-4 do not use peripherals, Enclave-1 and Enclave-4 do not have any IOB. Figure 8 (in Appendix) shows the block diagrams of the enclaves generated by the HARDWAREBUILDER for the four example SSAs.

D. Security Evaluation

To demonstrate the security of BYOTEE, we report the software TCB size of the four example SSAs and evaluate the cold-boot attacks on DRAM and BRAM of the same board.

1) Software TCB Size: Table III presents the size of the software TCB for the four example SSAs and their corresponding FIRMWARE. As the table shows, the size of FIRMWARE increases as the SSA gets more complicated and needs more services. Nevertheless, the run-time software TCB (SSA and FIRMWARE combined) of SSA-3, which is a functional digital right management music player, has only 10,727 SLOC, representing a significant software TCB reduction from its counterpart implemented as a TrustZone, e.g., TF-M [54] has

TABLE IV: FIRMWARE performance evaluation on the low-end softcore MicroBlaze CPU (Time in milliseconds; size in bytes). The experiments show SSA-3 can verify, decrypt, and play 48KHz WAV music smoothly on the very low-end softcore CPU.

	Binary size	Input size	Output size	Loading	Decryption	Integrity and authenticity verification	pre-execution attestation	post-execution attestation	Cleaning up	Suspend and export	Restore and execute
SSA-1	12,892	64	64	1.39	2784.56	118.54	153.94	154.79	0.54	3694.55	3609.65
SSA-2	2,596	N/A	N/A	0.30	579.11	29.15	30.90	30.90	0.11	741.71	729.69
SSA-3	20,152	128	132	2.17	4414.32	185.30	258.30	260.50	0.90	5787.97	5638.76

TABLE V: Embench-IoT performance evaluation in millisecond on softcore MicroBlaze CPU (Zynq-7000 FPGA implemented, version 10.0, 100MHz, no cache, no FPU) and hardcore Cortex-M4 CPU (16MHz, no cache, no FPU, officially reported performance from [3]).

Application	Description	M4 [3]	MicroBlaze
aha-mont64	Modulo generator	4,004	501
crc32	32 bit error detector	4,010	193
huffbench	Data compressor	4,120	111
minver	Floating point matrix inversion	3,998	327
nettle-aes	Low level AES library	4,026	245
nsichneu	Computes permutation	4,001	237
primecount	Prime counter	n/a	193
sglib-combined	Sort, search, and query on array, list, and tree	3,981	189
slre	Regex matching	4,010	113
statemate	Car window lift control	4,001	139
tarfind	Archive file finder	n/a	163
ud	Matrix factorization	3,999	343
Geometric Mean		4,015	208

(a) 0 s   (b) 5 s   (c) 15 s   (d) 20 s   (e) 30 s

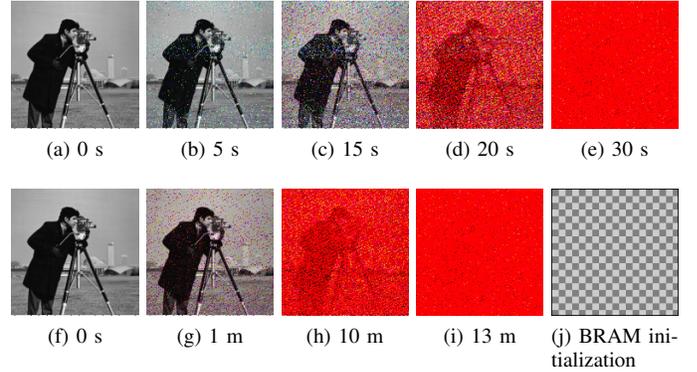(f) 0 s   (g) 1 m   (h) 10 m   (i) 13 m   (j) BRAM initialization

Fig. 6: Visualizing DRAM decay after power down and BRAM hardware initialization after power up on the same Z7-07S board. We loaded a bitmap image (150×150 pixel; 90.1kB) on the DRAM and BRAM. We then measure 1) DRAM decay at room temperature (20°C/68°F) after (a) power reset (0 second) and (b) - (e) losing power for different intervals; 2) DRAM decay at -18°C/0°F after (f) power reset and (g) - (i) losing power for different intervals; and 3) (j) BRAM hardware initialization after power up. The reconstructed image from the fully decayed DRAM is red because half of the cells are measured as 1s and the other half as 0s. The reconstructed image from the BRAM is transparent because all the bits are initialized to 0s.

over 117K SLOC, or SGX application, e.g., the Gramine library OS [4] has 83K SLOC.

2) Cold-boot Attacks on DRAM and BRAM: We evaluated the feasibility of cold-boot attacks on DRAM and BRAM on the same Cora Z7-07S board. In these experiments, we loaded a bitmap image onto the DRAM and BRAM and measured the DRAM decay at room temperature (20°C/68°F) and -18°C/0°F after power reset (0 seconds) and losing power for different intervals. We dumped the content of BRAM, for which the Xilinx Zynq-7000 FPGA has a non-bypassable hardware initialization mechanism after power up to clear all the bits to 0s. As we discussed in VI-D, even if the BRAM is not initialized, cold-boot attacks on it are much more difficult than on DRAM. Figure 6 visualizes the cold-boot attack results, which clearly show cold-boot attacks on DRAM are feasible but not on BRAM.

E. Performance on the Low-end MicroBlaze Softcore CPU

We evaluate the performance of the low-end MicroBlaze-powered enclaves, which provides a lower-bound performance estimation of available softcore CPUs. We first evaluate the performance using 12 Embench-IoT benchmark applications [14]. We then evaluate the BYOTEE software runtime performance by measuring the time cost of different FIRMWARE operations.

1) Benchmark Performance Evaluation: To show the performance of the MicroBlaze softcore compared to the Cortex-M4 hardcore, we use 12 applications from the Embench-IoT

benchmarks [14]. As Table V shows, the applications run comparatively faster on the low-end MicroBlaze softcore CPU than the hardcore Cortex-M4. For better performance, users can choose more advanced softcore CPUs.

2) FIRMWARE Performance: We evaluate the time FIRMWARE spends on the dynamic loading, decrypting, integrity and authenticity verification, attestation, cleaning up, suspending, and restoring operations of three SSAs, As Table IV shows, the time spent by the FIRMWARE on these operations is linear to the size of the SSA and its data combined. To copy the protected SSA and its input from DRAM to BRAM, FIRMWARE running on the Z7-07S spends around 1.07 ms for every 10,000 bytes. To decrypt the protected SSA and its data using 256-bit CBC mode AES, FIRMWARE spends around 2182 ms for every 10,000 bytes. The integrity and authenticity verification costs around 93.43 ms for every 10,000 bytes. The BLAKE-based pre-execution and post-execution cost around 124.77 ms for every 10,000 bytes. Cleaning up the BRAM takes around 0.45 ms for every 10,000 bytes. The SHA512-HMAC and AES 256-bit with CBC mode based suspending and restoring cost roughly 2834 ms for every 10,000 bytes.

Projects	Benefits												
	Underlying Hardware Primitive	Multiple TEEs	TEE with dedicated hardware	Customizable CPU and Memory	Customizable Peripheral Connections	Minimum Software TCB	Trust Bootstrap	Remote Attestation	Post-Execution Attestation	Concurrent TEE and REE Execution	Cache Side-channel Attack Resistant	Cold-boot Attack Resistant	Deployable on Commodity Devices
Flickr [59]	S/T					✓	✓	✓	✓				✓
TrustVisor [58]	S/T					✓	✓	✓	✓				✓
Haven [21]	X	✓						✓	✓				✓
SGXIO [81]	X+H	✓			✓				✓				✓
SGX-FPGA [82]	X+F	✓			✓				✓				✓
KeyStone [52]	R	✓				✓			✓				✓
Sanctum [32]	R	✓				✓			✓	✓	✓		
CURE [20]	R	✓				✓					✓		✓
Composite Encl. [71]	R	✓				✓		✓	✓				✓
SANCTUARY [24]	A	✓			✓				✓				✓
TrustICE [77]	A	✓							✓	✓			✓
vTZ [43]	A+H	✓											✓
Ambassy [45]	A+F	—	—	✓	—				✓	✓	✓	✓	✓
uTango [62]	M	✓							✓				✓
Graviton [80]	G	✓						✓	✓		✓	✓	
HECTOR-V [60]	N	✓	✓		✓						✓	✓	
TEEOD [73]	F	✓	✓	✓		✓					✓	✓	✓
BYOTEE	F	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

S: AMD Secure Virtual Machine extension, T: Intel Trusted eXecution Technology, H: Hypervisor, A: Arm Cortex-A TrustZone, M: Arm Cortex-M TrustZone, X: Intel SGX, F: FPGA, R: RISC-V, G: GPU, N: New hardware design. —: not applicable.

TABLE VI: Comparison with other TEE projects

VIII. RELATED WORK

Many software or hardware solutions have been proposed to address one or more limitations of existing TEEs, but none tackle all the issues at the same time as BYOTEE does. Among them, TEEOD [73], which Pereira et al. independently developed at roughly the same time, is most related to BYOTEE. Different from BYOTEE, TEEOD implemented SSA loading, inter-enclave communications, etc., as hardware IPs. BYOTEE offers additional security features, such as trust bootstrap, attestation, etc. Table VI lists the benefits BYOTEE offers and compares related work with BYOTEE.

To address the single TEE issue, vTZ [43] provides each guest virtual machine with a virtualized guest TEE by running a monitor within the secure world, which virtualizes memory mapping and world switching for the guest TEEs on Cortex-A TrustZone. SANCTUARY [24] addresses the same drawback by utilizing the memory access controller to provide multi-domain isolation for sensitive applications. TrustICE [77] creates multiple isolated computing environments in the normal domain and runs a monitor in the secure world. uTango [62] use the secure attribution unit of Cortex-M to create multiple secure execution environments within the non-secure state. The uTango kernel runs in the secure state privileged level, while other applications, services, OSes are isolated in their own non-secure state domains. On RISC-V, KeyStone [52] utilizes the Physical Memory Protection (PMP) feature to create multiple enclaves. The TEE and REE in these solutions time-share the CPU and other hardware resources, resulting in side-channel attacks.

To enable isolated or trusted I/O paths between TEEs and peripherals, CURE [20] enables the exclusive assignment of system resources, e.g., peripherals, CPU cores, or cache resources to single enclaves. Composite Enclaves [71] builds on top of KeyStone and extends the TEE to several hardware components. HECTOR-V [60] uses a dedicated processor as a TEE with configurable peripheral access permissions for secure communication. CURE, Composite Enclaves, and HECTOR-V were implemented on RISC-V. SGXIO [81] presents a hypervisor-based trusted path architecture for Intel SGX. SGX-FPGA [82] builds a secure hardware isolation path between CPU and FPGA. To eliminate side-channel attacks, Sanctum [32] adopts isolation that combines minimally invasive hardware modifications with a trusted software security monitor on RISC-V.

There have also been attempts to utilize other hardware computing modules to build TEEs. Graviton [80], which requires some hardware change, enables applications to offload security and performance-sensitive kernels and data to a GPU, and execute them in isolation. Ambassy [45] uses FPGA logic fabric to construct a second TEE and the secure world of TrustZone to control the secure loading of bitstream to the FPGA. Different from BYOTEE, Ambassy does not include softcore CPUs for developers to execute arbitrary code. Other off-SoC dedicated processor solutions, such as Google Titan [50], Samsung eSE [69], and Apple SEP [46] uses external connections for communication between the REE and TEE, making them vulnerable to physical probing attacks. BYOTEE is also inspired by other isolated execution environment solutions, including Flickr [59], TrustVisor [58], and Haven [21], and other hypervisor-based protections, such as Overshadow [29], InkTag [42], AppSec [68], and Terra [35].

IX. CONCLUSION

Even though existing hardware-assisted TEEs on commodity computing devices have been widely adopted in commercial systems, they have many issues that make them either ineffective, inefficient, or untrustworthy. In this paper, we present BYOTEE, which is a framework for building multiple equally secure TEEs on-demand with configurable hardware and software TCBs utilizing commodity SoC FPGA devices. In BYOTEE, enclaves, which include softcore CPUs, memory, and peripherals, are created on the FPGA, and the BYOTEEFIRMWARE provides necessary software libraries for the SSAs to use. Additionally, the FIRMWARE offers two attestation mechanisms, namely pre-execution and post-execution attestations, to verify the hardware and software stacks. We implemented a proof-of-concept BYOTEE for Xilinx SoC FPGA. The evaluation results on the low-end Zynq-7000 system for benchmark applications and example SSAs show the effectiveness and performance of BYOTEE.

X. ACKNOWLEDGMENT

REFERENCES

[1] A2I Source code. https://github.com/openpower-cores/a2i, -. [Online; accessed 09-Feb-2022].

[2] A2O Source Code. https://github.com/openpower-cores/a2o, -. [Online; accessed 09-Feb-2022].

[3] Embench IoT benchmark Cortex-M4 data. https://gitlab.inria.fr/mesc oute/embench-iot/-/tree/76e887fac691d3d3f42cd32636b347bf262603 6b/doc, -. [Online; accessed 09-Feb-2022].

[4] Gramine Source Code. https://github.com/gramineproject/gramine, -. [Online; accessed 8-March-2022].

[5] libreSOC documentation, Source code. https://libre-soc.org/#, -. [Online; accessed 09-Feb-2022].

[6] Microwatt Source code. https://github.com/antonblanchard/microwatt, -. [Online; accessed 09-Feb-2022].

[7] Open Cores. https://opencores.org/, -. [Online; accessed 06-March-2022].

[8] OpenSPARC T1 Softcore Processor. https://www.oracle.com/servers /technologies/opensparc-t1-page.html, -. [Online; accessed 09-Feb-2022].

[9] BLAKE2—fast secure hashing. https://www.blake2.net/, 2015. [Online; accessed 20-June-2021].

[10] OP-TEE. https://github.com/OP-TEE/optee_os, 2015. [Online; accessed 20-Nov-2020].

[11] Cora Z7-07S Board description. https://store.digilentinc.com/cora-z7-zynq-7000-single-core-and-dual-core-options-for-arm-fpga-soc-devel opment/, 2018. [Online; accessed 09-August-2021].

[12] Pmod I2S2: Stereo Audio Input and Output. https://store.digilentinc.co m/pmod-i2s2-stereo-audio-input-and-output/, 2018. [Online; accessed 09-August-2021].

[13] Neo430 soft core source code. https://github.com/stnolting/neo430, 2020. [Online; accessed 02-Dec-2020].

[14] Embench IoT github. https://github.com/embench/embench-iot, 2021. [Online; accessed 06-October-2021].

[15] VexRiscv soft core source code. https://github.com/SpinalHDL/VexRi scv, 2022. [Online; accessed 22-Jan-2022].

[16] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A.-R. Sadeghi, and M. Schunter. DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[17] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer. μRAI: Securing embedded systems with return address integrity. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[18] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell, et al. SCONE: Secure linux containers with intel SGX. In *USENIX symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[19] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[20] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *USENIX Security Symposium*, 2021.

[21] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[22] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 2015.

[23] S. Bhartiya. Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd. https://www.linux.com/news/linux-in-2020-2 7-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/.

[24] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[25] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. *arXiv preprint arXiv:1702.07521*, 2017.

[26] P. T. Breuer, C. K. Delgado, A. L. Marin, N. Martinez Madrid, and L. Sanchez Fernandez. A refinement calculus for the synthesis of verified hardware descriptions in vhdl. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):586–616, 1997.

[27] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *IEEE symposium on Security and Privacy (S&P), San Francisco, CA, USA*, 2020.

[28] W. N. Chelton and M. Benaissa. Fast elliptic curve cryptography on fpga. *IEEE transactions on very large scale integration (VLSI) systems*, 16(2):198–205, 2008.

[29] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review (OSR)*, 2008.

[30] H. Cho, P. Zhang, D. Kim, J. Park, C.-H. Lee, Z. Zhao, A. Doupé, and G.-J. Ahn. Prime+Count: Novel cross-world covert channels on arm trustzone. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[31] V. Costan and S. Devadas. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016.

[32] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[33] R. Elkhatib, R. Azarderakhsh, and M. Mozaffari-Kermani. High-performance fpga accelerator for sike. *IEEE Transactions on Computers*, 2021.

[34] M. Ender, A. Moradi, and C. Paar. The unpatchable silicon: A full break of the bitstream encryption of xilinx 7-series fpgas. In *USENIX Security Symposium*, 2020.

[35] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[36] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747*, 2014.

[37] I. Giechaskiel, K. B. Rasmussen, and K. Eguro. Leaky wires: Information leakage and covert communication between FPGA long wires. In *Asia Conference on Computer and Communications Security (AsiaCCS)*, 2018.

[38] M. Gross, N. Jacob, A. Zankl, and G. Sigl. Breaking trustzone memory isolation through malicious hardware on a modern fpga-soc. In *ACM Workshop on Attacks and Solutions in Hardware Security Workshop (ASHES)*, 2019.

[39] M. Gutierrez, Z. Zhao, A. Doupé, Y. Shoshitaishvili, and G.-J. Ahn. Cachelight: Defeating the cachekit attack. In *Workshop on Attacks and Solutions in Hardware Security*, 2018.

[40] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM (CACM)*, 2009.

[41] C. Heinz, Y. Lavan, J. Hofmann, and A. Koch. A catalog and in-hardware evaluation of open-source drop-in compatible risc-v softcore processors. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2019.

[42] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: Secure applications on an untrusted operating system. In
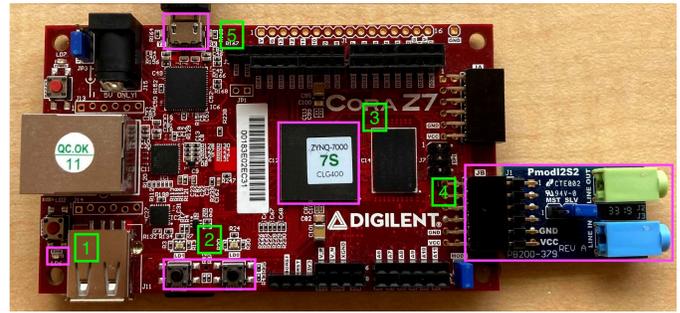
International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2013.

[43] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vtz: Virtualizing ARM trustzone. In *USENIX Security Symposium*, 2017.

[44] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong. Programming and runtime support to blaze fpga accelerator deployment at datacenter scale. In *ACM Symposium on Cloud Computing*, pages 456–469, 2016.

[45] D. Hwang, S. Yeleuov, J. Seo, M. Chung, H. Moon, and Y. Paek. Ambassy: A Runtime Framework to Delegate Trusted Applications in an ARM/FPGA Hybrid System. *IEEE Transactions on Mobile Computing (TMC)*, 2021.

[46] A. Inc. Security enclave processor for a system on a chip. US8832465B2. https://patents.google.com/patent/US8832465, 2020. [Online; accessed 02-Jan-2021].

[47] Intel. Agilex F-series 27 R25A. https://www.intel.com/content/www/us/en/products/sku/208599/intel-agilex-fseries-027-fpga-r25a/specifications.html, -. [Online; accessed 09-Feb-2022].

[48] Intel. Intel® Quartus® Prime Pro Edition Design Software. https://www.intel.com/content/www/us/en/software-kit/706104/intel-quartus-prime-pro-edition-design-software-version-21-4-for-linux.html?, -. [Online; accessed 09-Feb-2022].

[49] Intel. Intel NIOS soft core. https://www.intel.com/content/www/us/en/products/details/fpga/nios-processor/v.html#:~:text=Nios%C2%AE%20V%20processor%20is,Software%20starting%20with%20version%202021.3., 2020. [Online; accessed 02-Dec-2020].

[50] S. Johnson, D. Rizzo, P. Ranganathan, J. McCune, and R. Ho. Titan: enabling a transparent silicon root of trust for Cloud. In *Hot Chips: A Symposium on High Performance Chips*, volume 194, 2018.

[51] M. Lavasani, H. Angepat, and D. Chiou. An fpga-based in-line accelerator for memcached. *IEEE Computer Architecture Letters*, 13(2):57–60, 2013.

[52] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *European Conference on Computer Systems (EuroSys)*, 2020.

[53] Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Liu, Q. Qiu, W. Xu, X. Lin, X. Qian, et al. E-rnn: Design optimization for efficient recurrent neural networks in fpgas. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 69–80. IEEE, 2019.

[54] Linaro. Trusted Firmware M (TFM) v1.3.0 sourcec code. https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tag/?h=TF-Mv1.3.0. Online; accessed 15 Apr 2021.

[55] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, et al. Glamdring: Automatic application partitioning for intel SGX. In *USENIX Annual Technical Conference (ATC)*, 2017.

[56] R. Lysecky and F. Vahid. Design and implementation of a microblaze-based warp processor. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(3):1–22, 2009.

[57] E. Matthews and L. Shannon. Taiga: A new risc-v soft-processor framework enabling high performance cpu architectural features. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[58] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE symposium on Security and Privacy (S&P)*, 2010.

[59] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *European Conference on Computer Systems (EuroSys)*, 2008.

[60] P. Nasahl, R. Schilling, M. Werner, and S. Mangard. Hector-v: A heterogeneous cpu architecture for a secure risc-v execution environment. *arXiv preprint arXiv:2009.05262*, 2020.

[61] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise. In *USENIX Security Symposium*, 2020.

[62] D. Oliveira, T. Gomes, and S. Pinto. uTango: an open-source TEE for the Internet of Things. *arXiv preprint arXiv:2102.03625*, 2021.

[63] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping trust in modern computers*. Springer Science & Business Media, 2011.

[64] S. Pinto and N. Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 2019.

[65] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35, 2016.

[66] A. Rahmati, M. Salajegheh, D. Holcomb, J. Sorber, W. P. Burleson, and K. Fu. TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks. In *USENIX Security Symposium*, 2012.

[67] C. Ramesh, S. B. Patil, S. N. Dhanuskodi, G. Provelengios, S. Pillement, D. Holcomb, and R. Tessier. FPGA side channel attacks without physical access. In *Annual international symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

[68] J. Ren, Y. Qi, Y. Dai, X. Wang, and Y. Shi. Appsec: A safe execution environment for security sensitive applications. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2015.

[69] Samsung. eSE Safeguard against digital attacks. https://www.samsung.com/semiconductor/security/ese/, 2020. [Online; accessed 20-Jan-2021].

[70] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[71] M. Schneider, A. Dhar, I. Puddu, K. Kostiainen, and S. Capkun. Composite Enclaves: Towards Disaggregated Trusted Execution. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022.

[72] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE symposium on Security and Privacy (S&P)*, 2015.

[73] C. R. Sergio Pereira, David Cerdeira and S. Pinto. Towards a Trusted Execution Environment via Reconfigurable FPGA. *arXiv preprint arXiv:2107.03781*, 2021.

[74] V. Sieh, O. Tschache, and F. Balbach. Verify: Evaluation of reliability using vhdl-models with embedded fault descriptions. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 32–36. IEEE, 1997.

[75] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[76] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 16–25, 2016.

[77] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015.

[78] B. Toulas. New Intel chips won't play Blu-ray disks due to SGX deprecation. https://www.bleepingcomputer.com/news/security/new-intel-chips-wont-play-blu-ray-disks-due-to-sgx-deprecation/#:~:text=Intel%20has%20removed%20support%20for,ray%20disks%20in%204K%20resolution., -. [Online; accessed 09-Feb-2022].

[79] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference (ATC)*, 2017.

[80] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted execution environments on gpus. In *USENIX symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[81] S. Weiser and M. Werner. Sgxio: Generic trusted i/o path for intel sgx. In *ACM on Conference on Data and Application Security and Privacy (CODASPY)*, 2017.

[82] K. Xia, Y. Luo, X. Xu, and S. Wei. Sgx-fpga: Trusted execution environment for cpu-fpga heterogeneous architecture. In *IEEE Design Automation Conference (DAC)*, 2021.

[83] Xilinx. UltraScale+ EV ZU9CG. https://www.xilinx.com/support/documentation/selection-guides/zynq-ultrascale-plus-product-selection-guide.pdf, -. [Online; accessed 09-Feb-2022].

[84] Xilinx. Xilinx Vivado Toolkit. https://www.xilinx.com/products/design-tools/vivado.html, -. [Online; accessed 09-Feb-2022].

[85] Xilinx. AXI GPIO v2.0. https://www.xilinx.com/support/documen tation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf, 2016. [Online; accessed 01-June-2021].

[86] Xilinx. AXI4-Stream FIFO v4.1. https://www.xilinx.com/support/d ocumentation/ip_documentation/axi_fifo_mm_s/v4_1/pg080-axi-fifo- mm-s.pdf, 2016. [Online; accessed 01-January-2021].

[87] Xilinx. LogiCORE IP Product Guide. https://www.xilinx.com/suppo rt/documentation/ip_documentation/axi_timer/v2_0/pg079-axi-timer.p df, 2016. [Online; accessed 14-May-2021].

[88] Xilinx. XADC Wizard v3.3. https://china.xilinx.com/support/docum entation/ip_documentation/xadc_wiz/v3_3/pg091-xadc-wiz.pdf, 2016. [Online; accessed 01-March-2021].

[89] Xilinx. MicroBlaze soft core. https://www.xilinx.com/products/design -tools/microblaze.html, 2018. [Online; accessed 19-Jul-2020].

[90] Xilinx. 7 Series FPGAs Memory Resources. https://www.xilinx.com /support/documentation/user_guides/ug473_7Series_Memory_Resourc es.pdf, 2019. [Online; accessed 08-Dec-2021].

[91] Xilinx. AXI DMA v7.1. https://www.xilinx.com/support/documen tation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf, 2019. [Online; accessed 29-Feb-2021].

[92] Xilinx. MicroBlaze Debug Modulev3.2. https://www.xilinx.com/suppo rt/documentation/ip_documentation/mdm/v3_2/pg115-mdm.pdf, 2021. [Online; accessed 09-March-2021].

[93] Xilinx. Using Encryption to Secure FPGA Bitstream. https://www.xi linx.com/support/documentation/application_notes/xapp1239-fpga-bit stream-encryption.pdf, 2021. [Online; accessed 08-Dec-2021].

[94] Xilinx. Zynq-7000 SoC Technical Reference Manual. https://www.xi linx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM. pdf, 2021. [Online; accessed 30-June-2021].

[95] Xlinix. Bootgen User Guide. https://www.xilinx.com/support/docum entation/sw_manuals/xilinx2018_2/ug1283-bootgen-user-guide.pdf, 2018. [Online; accessed 09-August-2021].

[96] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 161–170, 2015.

[97] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptology ePrint Archive*, 2016.

[98] P. Zhang, H. Cho, Z. Zhao, A. Doupé, and G.-J. Ahn. iCore: continuous and proactive extrospection on multi-core IoT devices. In *ACM Symposium on Applied Computing (SAC)*, 2019.

[99] X. Zhang, Y. Xiao, and Y. Zhang. Return-oriented flush-reload side channels on arm and their implications for android devices. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[100] M. Zhao and G. E. Suh. FPGA-based remote power side-channel attacks. In *IEEE symposium on Security and Privacy (S&P)*, 2018.

[101] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls. Silhouette: Efficient protected shadow stacks for embedded systems. In *USENIX Security Symposium*, 2020.
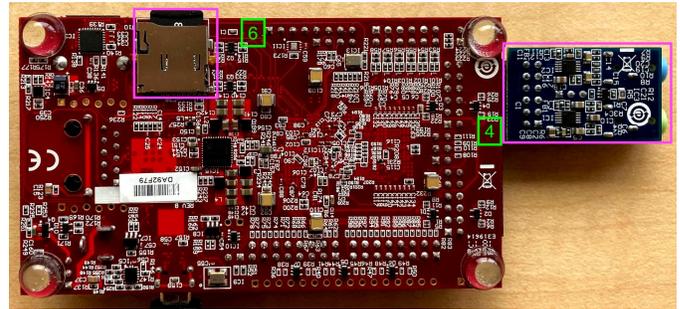
APPENDIX

Figure 7 shows the top and bottom view of the Cora Z7-07S development board with a single-core 667MHz Arm Cortex-A9 processor, a Xilinx Zynq-7000 FPGA, and the connected Pmod I2S2 audio device.

Figure 8(a) shows the hardware for SSA-1 with a MicroBlaze Debugging Module (MDM) [92] to help the developers debug SSAs. A master interface of MDM is also connected to the ps7_axi_periph to enable debugging from the hardcore system side. Figure 8(b) shows the same hardware configuration without the debugging module. As shown in Figure 8(c), the hardware design of SSA-2 includes two peripherals that are only connected to the FPGA. A pulse width modulation IP [87] connects the RGB_LED ports, and an AXI GPIO IP [85] connects the button ports to the mb_axi_interconnect_0 interconnect. As shown in Figure 8(d), the hardware for SSA-3 includes several more IPs for different functionality. An I2S

(a) Top view

(b) Bottom view

Fig. 7: The Z7-07S board with a Pmod audio module for experiments and evaluation: (1) an LED as an Enclave-2 peripheral used by SSA-2, (2) a button as an Enclave-2 peripheral used by SSA-2, (3) the Zynq-7000 SoC with a hardcore CPU and FPGA, (4) a Pmod port and the I2S2 stereo audio input and output device as an Enclave-3 peripheral used by SSA-3, (5) the on-board USB JTAG/UART for debugging and terminal output, (6) the SD card slot.

transmitter RTL (SPI) is added to interact with the Pmod I2S2 module Codec module. Additionally, an AXI direct memory access IP [91] with BRAM and an AXI stream data FIFO IP [86] to offload audio data streaming computation from MicroBlaze are inserted by the HARDWAREBUILDER. An ADC module for reading voltages, which is used for performance monitoring [88], is also connected. Figure 8(e) represents the hardware of SSA-4 with two enclaves. Both the enclaves have their own BRAM, 128KB, and 32KB respectively. Enclave-1 have a shared DRAM with Zynq processor as SEB for communication, while Enclave-2 does not have any SEB with Zynq. Instead of inter enclave communication, one 8KB BRAM is shared between Enclave-1 and Enclave-2 without the Zynq processing system access.

(a) Hardware for SSA-1 with a Debugging Module

(b) Hardware for SSA-1 without a Debugging Module

(c) Hardware for SSA-2 without a Debugging Module

(d) Hardware for SSA-3 without a Debugging Module

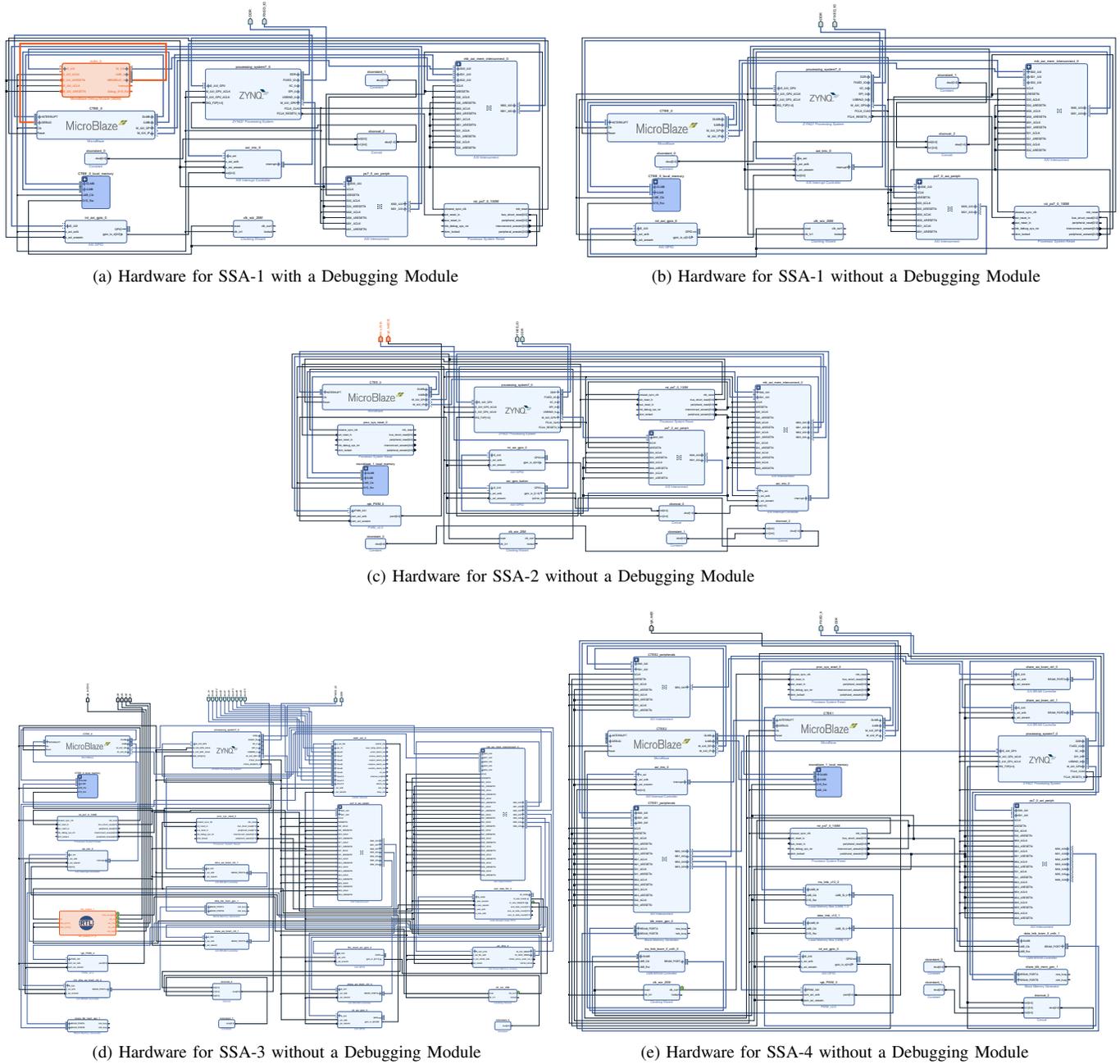(e) Hardware for SSA-4 without a Debugging Module

Fig. 8: Block diagrams of the hardware designs of the four enclaves for the four example SSAs. The ZYNQ processing system represents the hardcore Cortex-A processor. The Enclave-1, Enclave-2, and Enclave-3 have one MicroBlaze softcore processor, whereas Enclave-4 has two MicroBlaze softcore processors. The MicroBlaze interrupt interface is connected to an AXI interrupt controller and configured with an AXI GPIO. All output GPIO to the softcore is connected to the hardcore system for triggering interrupts. Two AXI Interconnects, mb_axi_mem_interconnect_0 for Microblaze and ps7_axi_periph for Cortex-A processor are used to connect external IPs. The ZYNQ, Microblaze, and other IPs get primary clock input from the clk_in1 port. The reset port is connected to the reset interfaces of the IPs.