Building Your Own Trusted Execution Environments Using FPGA

Md Armanuzzaman
CactiLab, University at Buffalo
Buffalo, USA
mdarmanu@buffalo.edu

Ahmad-Reza Sadeghi
Technische Universität Darmstadt
Darmstadt, Germany
ahmad.sadeghi@trust.informatik.tu-darmstadt.de

Ziming Zhao
CactiLab, University at Buffalo
Buffalo, USA
zimingzh@buffalo.edu

ABSTRACT

Despite of their benefits, existing Trusted Execution Environments (TEE) or enclaves have been criticized for lack of transparency, vulnerabilities, and various restrictions. A significant limitation is that they only provide a static and fixed hardware Trusted Computing Base (TCB) that cannot be customized for different applications. The design violates the principle of least privilege by including unnecessary peripherals in the hardware TCB and buggy peripheral drivers in the software TCB. Additionally, Existing TEEs time-share a processor core with the Rich Execution Environment (REE), making execution less efficient and vulnerable to cache side-channel attacks. Although many previous projects have focused on addressing software issues in TEEs on SGX, TrustZone, or RISC-V, some TEE issues are inherent in the hardware system's design, making them impossible to resolve with software alone.

In this paper, we present BYOTee (Build Your Own Trusted Execution Environments), which is an easy-to-use hardware and software co-design infrastructure for building enclaves using Field Programmable Gate Arrays (FPGA). BYOTee creates enclaves with customized hardware TCBs and establishes a dynamic root of trust that allows untampered execution of Security-Sensitive Applications (SSA) from preexisting software on the hardcore system. Additionally, BYOTee provides mechanisms to attest the integrity of enclaves' hardware and software stacks. 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.

CCS CONCEPTS

• **Security and privacy** → **Systems security**.

KEYWORDS

Trusted execution environment; field-programmable gate array

ACM Reference Format:
Md Armanuzzaman, Ahmad-Reza Sadeghi, and Ziming Zhao. 2024. Building Your Own Trusted Execution Environments Using FPGA. In *Proceedings of the 2024 ACM ASIA Conference on Computer and Communications Security (AsiaCCS '24), July 1–5, 2024, Singapore.* ACM, New York, NY, USA, 16 pages. https://doi.org/xx.xxx/xx.xxx

1 INTRODUCTION

Existing Trusted Execution Environments (TEEs) on commodity computing devices rely on CPU hardware security primitives to ensure the confidentiality and integrity of code and data loaded within them, while protecting them from the Rich Execution Environment (REE). These hardware security primitives are provided by the CPU and can either be proprietary, as in Intel SGX [1] and Arm TrustZone [2], or open-sourced, as in RISC-V [3]. In recent years, we have witnessed unprecedented growth in using such TEEs in real-world products and academic projects, which include real-time kernel protections [4, 5], securing containers and libraries [6–8], shielding applications from attacks [9–11]. However, the hardware layer of current TEEs suffers from various issues, rendering them untrustworthy or ineffective.

Firstly, they only offer a static and fixed hardware Trusted Computing Base (TCB) that cannot be customized for different applications at *runtime*. Although RISC-V allows for hardware customization at design and manufacturing time, the resulting hardware configuration is static and cannot be changed after manufacturing. The hardware primitives of TrustZone give the TEE the highest privilege to control the REE and communicate with all peripherals, thereby violating the principle of least privilege. It also includes unnecessary peripherals and buggy peripheral drivers in the software TCB [12], exposing the TEE to malicious peripheral inputs [13]. Meanwhile, SGX's hardware requires applications in enclaves to trust the REE Operating System (OS) to communicate with peripherals [14], bloating the size of the software TCB by including a usually monolithic REE OS kernel.

Another significant issue is that most commercially popular TEE hardware, such as SGX and TrustZone, are proprietary and limited to specific architectures, which require users to place blind trust in their security. As a result, users cannot verify the correctness of the TEE designs. Unfortunately, vulnerabilities, such as cache side-channels, have been discovered in both SGX and TrustZone [15–19], which undermines their security promises. Additionally, the proprietary nature of such TEE systems poses a challenge for researchers to explore the security properties and capabilities of different TEE configurations. Although open-sourced TEE designs based on RISC-V can be verified at the design stage, dynamically attesting hardware states at runtime remains an unsolved problem.

Software-based solutions alone on existing hardware TEEs cannot address the aforementioned issues, which are rooted in the design of their respective hardware systems. For example, Sanctuary [20] configures the memory access controller to provide multi-domain isolation for sensitive applications on TrustZone, and

Cure [3] enables the exclusive assignment of system resources, e.g., peripherals, CPU cores, or cache resources, to each enclave on RISC-V. While these solutions are noteworthy, they do not provide a customizable and attestable hardware TCB at runtime. Other hardware-based solutions, such as HECTOR-V [21] and Graviton [22], do not address these issues either, and they cannot be deployed on commodity devices due to the need for hardware modifications.

In this paper, we present a hardware and software co-design framework to Build Your Own Trusted Execution Environments (BYOTee). BYOTee utilizes commodity System-on-Chip (SoC) Field Programmable Gate Arrays (FPGAs), e.g., AMD EPYC FPGA-infused CPU [23] and Xilinx SoC FPGA, without requiring any hardware changes. With the BYOTee toolchain, users can quickly and easily build multiple secure and customized enclaves on-demand to execute their Security-Sensitive Applications (SSA). Each enclave is designed to include only the hardware and software necessary for the SSA and excludes other hardware and software components on the system, minimizing the sizes of hardware and software TCB.

A SoC FPGA system, including FPGA-infused CPUs, integrates both a hardcore CPU, e.g., x86/64, Cortex-A or RISC-V, and an FPGA programmable logic architectures. The nature of FPGA enables on-demand configurations 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. Each enclave in BYOTee has its own isolated physical address space, which maps its own dedicated main memory, system configuration registers, and peripherals. The software on the hardcore CPU and other enclaves cannot access an enclave's address space unless it is explicitly specified in the design. By assigning the hardware resources to co-resident enclaves, BYOTee creates a multiprogramming environment, isolates software faults, and provides memory protection on FPGAs. Enclaves in BYOTee do not share processors with each other or the hardcore system, which eliminates the cache side-channel attack vector. Additionally, due to the characteristics of BRAM, cold-boot attacks on these enclaves are challenging.

BYOTee utilizes the *secure configuration* process of the FPGA to establish a *dynamic root of trust* that ensures complete isolation and untampered execution of Security-Sensitive Applications (SSAs) in enclaves from preexisting software on the hardcore system, including the hypervisor and operating system. Additionally, BYOTee offers both software- and hardware-based remote attestation mechanisms that operate under two threat models. To enable execution of SSAs, external libraries and drivers for peripherals are required. On the software front, the configurable Firmware component of BYOTee provides essential software libraries such as libc, as well as a Hardware Abstraction Layer (HAL), to minimize the software Trusted Computing Base (TCB).

We have implemented the BYOTee infrastructure and toolchain for the Xilinx SoC FPGA. The toolchain comprises several components, including HardwareBuilder, which takes developers' hardware resource requirements as input and generates hardware modules and interconnections. By automating this process, HardwareBuilder reduces the likelihood of developer-induced misconfigurations and allows developers to focus on SSA development, thereby increasing the usability of BYOTee. Additionally, the system and toolchain include Hw-Att, a trusted decryption and attestation hardware module implemented in bitstream, Firmware, a software runtime for SSAs, and SSAPacker, a tool used to encrypt and sign an SSA binary. The contributions of this paper are as follows:

- We present BYOTee, a framework to create enclaves with minimal hardware and software TCBs on commodity SoC FPGA. The framework can also help researchers who want to explore the capabilities and security properties of different TEE hardware configurations. The idea of BYOTee can be implemented on FPGA systems with a secure configuration process from any vendor;
- BYOTee establishes a dynamic root of trust that allows full isolation and untampered execution of SSAs in enclaves from preexisting software on the hardcore system;
- We present software- and hardware-based remote attestations for two threat models to capture the identities of the enclave hardware configurations, firmware, and SSAs;
- We implement the BYOTee system and toolchain for the Xilinx SoC FPGA. We open-source the BYOTee system and toolchain[1];
- We demonstrate BYOTee's usage, security, effectiveness, and performance with the Embench-IoT benchmark and four SSAs on the low-end MicroBlaze softcore CPU and Zynq-7000 system.

2 SOC FPGA AND ROOT OF TRUST

In this section, we provide an overview of FPGA and the hardware modules and root of trust commonly found on a SoC FPGA. We will also discuss the workflow involved in designing, developing, and securely configuring a SoC FPGA.

2.1 Benefits of FPGA and FPGA in End Products

FPGA is designed to be configured by users using Register-Transfer Level (RTL) code after manufacturing. Besides reconfigurability, it has advantages of high performance, fast development round, etc. While FPGAs were mainly used for hardware prototyping years ago, their benefits and reduced costs have made them practical for end products recently [24, 25]. A variety of FPGA products ranging from embedded systems, i.e., Xilinx Zynq-7000 with only 3,600 logic elements (≈\$70), to data center devices, i.e., Agilex F R25A with 2.6 millions of logic elements (≈\$10k), are available. Amazon Elastic Compute Cloud has been offering FPGAs to their customers in F1 instances since 2017 [26], and AMD starts infusing EPYC CPUs with FPGA in 2023 [23]. Various FPGA-based application-specific accelerators, such as deep neural networks [27–29], classic and post-quantum cryptographic algorithms [30, 31], Memcached [32], have been proposed and deployed.

FPGA can be used to build general-purpose computing platforms, in which users can design and implement their own softcore CPUs or customize existing open-sourced [33–40] or proprietary ones [41, 42]. The available softcore CPUs range from the partially configurable (e.g., cache size, pipeline depth) and proprietary low-end 32-bit MicroBlaze [41], to the fully customizable and open-sourced mid-end 32-bit RISC-V [34, 43] and high-end 64-bit A2I POWER processor [37]. Even though existing softcore CPUs on FPGA have a lower maximum clock frequency, i.e., 500MHz, than their hardcore counterparts, i.e., 4GHz, softcore and hardcore CPUs with similar complexity and frequency have comparable performance. While the low-end softcore CPUs are comparable to microcontrollers, the mid-end and high-end ones have performances

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

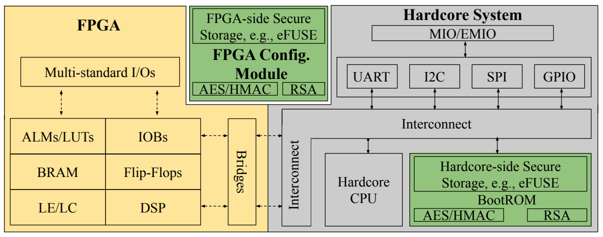


Figure 1: SoC FPGA architecture. The modules in green are the *root of trust* for the hardcore system and FPGA, respectively. Note that BYOTEE only relies on the root of trust for FPGA. Solid lines represent hard-wired connections, whereas dashed lines represent configurable connections.

comparable to hardcore microprocessors [44]. Because it is possible to formally verify RTL implementations [45, 46], users do not have to blindly trust a whole CPU but just the FPGA configuration modules and RTL verifiers.

2.2 Hardware Modules and Root of Trust

Figure 1 shows the three main modules of a SoC FPGA. A *hardcore system* is formed around hard processors, such as the x86/64 processor on AMD EPYC and the Cortex-A processor on Xilinx Zynq-7000 SoC [47], which includes a BootROM that contains the immutable BootROM code. In the *secure boot* process of the hardcore system, the BootROM code serves as the *Static Root of Trust for Measurement* (SRTM) and uses the keys stored in the hardcore-side secure storage, e.g., one-time programmable electronic fuse (eFUSE).

An *FPGA* can implement arbitrary systems, including softcore CPUs. To this end, the FPGA is composed of configurable logic blocks, Adaptive Logic Modules (ALMs), Lookup Tables (LUTs), flip-flops, etc. 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 [48], SRAM decays faster [49]. The FPGA also includes Input/Output Blocks (IOB) for interfacing.

An *FPGA configuration module* configures the FPGA with a *bitstream*. The module not only configures the hardware but also loads optional software onto the BRAM. Therefore, a bitstream can include: (i) hardware configurations, which are programmed in RTL hardware design description languages, such as Verilog and VHDL; and (ii) optional software, e.g., firmware, that runs on the hardware configurations. In the *secure configuration* process of the FPGA, the FPGA configuration module (FCM) serves as the *Root of Trust for Measurement* and uses the keys in the FPGA-side secure storage, e.g., BBRAM, eFUSE, to verify, decrypt, and configure an encrypted bitstream. Note that the BootROM code cannot access the FPGA-side secure storage, and vice versa.

2.3 Secure Configuration of FPGA

A typical design and development flow of general-purpose computing platforms on SoC FPGA involves: (i) the development of the hardware on the FPGA, including designing the peripheral blocks and creating the connections. In this step, a developer can use and customize open-sourced and proprietary hardware IPs; and (ii) the development of the software on the hardcore and softcore CPUs.

As aforementioned, the FPGA has a *secure configuration* process, which is independent from the *secure boot* process of the hardcore system. To facilitate the secure boot and secure configuration processes, cryptographic keys can be generated off the device and programmed to the corresponding secure storage on the device during manufacturing or through the JTAG interface if available. These keys and the FPGA configuration module are the *root of trust* of BYOTEE. When a SoC FPGA device is powered on, the hardcore system boots. Privileged software, e.g., First Stage Boot Loader (FSBL) or OS, on the hardcore system can read a bitstream at any time from the persistent storage and send it to the FPGA configuration module, which verifies and configures the FPGA.

3 SYSTEM, THREAT AND DEPLOYMENT MODEL

System Model. We assume the secure configuration process of FPGA discussed in §2. 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 [50]. Even though recent research [51] discovered dumping content from the FPGA is possible due to some implementation bugs, it was not intended to be a feature. We assume the cryptographic algorithms are secure.

Threat Models. We assume adversaries can compromise the hardcore system at boot-time or runtime, which means applications, kernel, and hypervisor are malicious. The compromised software on the hardcore system can send arbitrary data to the firmware and SSAs in enclaves via shared DRAM regions and to the enclave hardware pins, such as interrupts. Adversaries can also perform cold-boot attacks to dump the content in DRAM.

For software running on the softcore CPUs, we first consider a baseline model (BaseModel) as the baseline design. We then consider BYOTEE under an enhanced attack model (EnhancedModel). In BaseModel, the software in an enclave, including the firmware and SSA, are trusted and bug-free. The hardcore system cannot compromise the firmware or SSA at runtime, and remote attestation can be implemented in the firmware. This model is similar to the Arm TrustZone model where software-based attestation is trustworthy [52, 53]. However, this model is not realistic as the firmware and SSAs may have bugs that can be exploited by REE inputs [12, 54]. In EnhancedModel, we assume that the firmware and SSAs are buggy and can be compromised. Therefore, measurement code and keys cannot be kept in the same address space as the firmware and SSAs. This model is similar to the Intel SGX and Arm CCA [55] model where trusted hardware components of the CPU perform remote attestation. We do not consider the Time-Of-Check-Time-Of-Use

(TOCTOU) attacks on hybrid remote attestation; however, we will discuss possible solutions in §9.

Key Management Model. BYOTee provides the necessary mechanisms to build a secure system, which can integrate different deployment and key management models that are related to the user and application's policies and needs. In this paper, we discuss two key management models for local and cloud deployment scenarios. We assume that the FPGA configuration module securely stores a device key, such as AES (k_d) or RSA (sk_d, pk_d), which is used to encrypt/decrypt and sign/verify a bitstream. These device keys are unique to each device and are programmed in a secure storage using hardware interfaces with physical access, such as during manufacturing. We also assume that developers can be identified by a developer key, such as AES (k_u) or RSA (sk_u, pk_u), which they use to encrypt and sign the SSAs. In BaseModel, the Firmware uses the developer key to decrypt and verify an SSA, and developer keys are embedded into the Firmware during the development stage. In EnhancedModel, the developer keys are embedded in the trusted hardware-based module Hw-Att for decryption and attestation.

4 BYOTEE ARCHITECTURE

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

4.1 Design Goals

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

G1. Customizable hardware TCB. The hardware TCB of each enclave should be customizable, allowing for a minimum TCB that only includes the necessary hardware, e.g., peripherals, for the SSA and excludes other hardware on the system. Please note that formally verifying the customized RTL hardware design [45, 46] is beyond the scope of BYOTee.

G2. Remote attestation mechanisms. The mechanisms in BYOTee should support protocols for remote verifiers to attest to the integrity of an enclave's hardware and software stacks. This includes the bitstream, Firmware, SSAs, and their inputs and outputs.

G3. General-purpose execution environments. BYOTee should provide general-purpose execution environments, similar to SGX and TrustZone, and not limited to application-specific accelerators [56]. The SSAs can be implemented in any programming language, as long as they can be linked against the firmware.

G4. Multiple isolated execution environments. BYOTee should provide multiple execution environments, similar to SGX, to ensure the confidentiality and integrity of the SSA running inside each environment. TrustZone and Ambassy [56] only provide a single execution environment, which can limit the flexibility of the system.

G5. Circuit-level execution isolation. BYOTee should provide dedicated CPUs for each execution environment, ensuring that all hardware resources for each enclave are isolated from the REE and from other enclaves at the circuit level. This approach mitigates micro-architectural side-channel attacks, such as cache side-channel attacks, that are prevalent in CPU-sharing TEEs such as SGX and TrustZone. However, please note that power side-channel attacks [57] may still be possible, which will be discussed in §6.

G6. Isolated path between SSA and peripherals. An enclave should isolate the communication path between the SSA and peripherals from the hardcore system and other enclaves, preventing software-based eavesdropping and tampering.

G7. Enclave-to-hardcore and Inter-enclave communication. The SSA in an enclave should be able to communicate with software on the hardcore system and other enclaves. The inter-enclave communication should be isolated from the hardcore system and non-participating enclaves.

G8. Allowing for minimum software TCB. The firmware serving an SSA should only include necessary housekeeping libraries and drivers that are necessary for the SSA execution and exclude other software on the system.

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

4.2 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, Hw-Att for the EnhancedModel, Firmware, and SSAPacker. During the development stage, the HardwareBuilder generates synthesizer commands based on the SSA's needs specified in the developer's hardware description JSON input. Then, the vendor-provided synthesizer, e.g., Xilinx Vivado [58], Intel Quartus Prime [59], generates the bitstream file using the synthesizer commands. The bitstream and Firmware binary are encrypted, signed, and packed by the vendor-provided merger, e.g., `UpdateMEM` from Xilinx, into a protected bitstream. The SSA binary is encrypted, signed, and packed by the SSAPacker into a protected SSA. When the bitstream is loaded onto the FPGA, multiple enclaves can be created and Firmware starts running. Then, an untrusted application can trigger the loading of a protected SSA into an enclave.

BYOTee meets *G1*, *G3*, *G4*, *G5*, and *G6* by configuring the FPGA to build enclaves. Enclaves are constructed with softcore CPUs, which provide a general-purpose computing environment (*G3*). Each enclave has its own set of hardware (*G4*), including a softcore CPU (e.g., MicroBlaze, UltraSPARC), Block RAM, and peripherals. Using FPGA routing, these hardware resources within an enclave are connected together, but isolated from the hard-core system and other enclaves (*G5*). The softcore CPU in an enclave is not time-shared with the hard-core system and other enclaves, mitigating cache side-channel attacks (*G5*). No additional hardware modules, such as debuggers, can be connected to an enclave unless explicitly specified by the developer (*G1*). Furthermore, all connections among these resources are isolated at the circuit level from the hard-core system and other enclaves, preventing eavesdropping and tampering (*G6*).

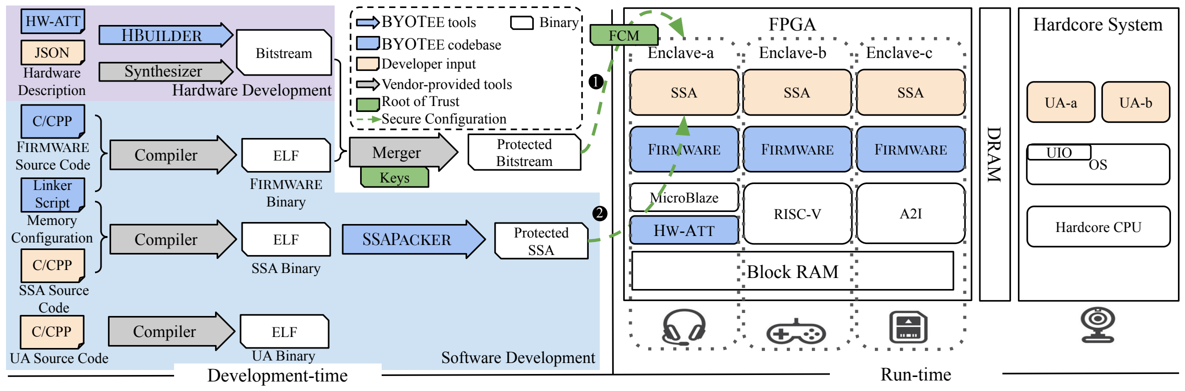


Figure 2: The architecture and workflow of the BYOTee framework at development- and run-time. During development, BYOTee and vendor-provided tools are used to generate the protected FPGA image and protected SSAs, which are loaded onto the FPGA (❶) and enclaves (❷), respectively. In this runtime architecture 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).

Enclaves in BYOTee use interrupts on softcore CPUs and shared physical memory regions on DRAM to communicate with the REE. In contrast, 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 REE and other enclaves, satisfying *G7*. The BYOTee firmware can be customized and only consists of libraries, a HAL, and a loader for the SSA, meeting *G8*. To prove the integrity of enclaves, BYOTee provides remote attestation mechanisms under two models, as required by *G2*. Additionally, 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, which satisfies *G9*.

4.3 Customizing Hardware TCB for Enclaves

To customize the hardware TCB, the developer designs enclaves using a hardware description language, e.g., Verilog or VHDL. The output is a hardware configuration bitstream file. To facilitate this step, HardwareBuilder takes developer-specified hardware description in JSON format as input (See an example in §5.2), 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 not limited to: (i) a softcore CPU and its configurations, e.g., clock frequency, cache size, etc; (ii) the selection of software- or hardware-based attestation mechanism; (iii) a corresponding debug IP to enable software debugging on the softcore CPU; iv) its main BRAM memory address and size; (v) the address and size of the shared DRAM with the hardcore system; (vi) the address and size of the shared BRAM with other enclaves; and (vii) connected peripherals. The HardwareBuilder assigns a contiguous address space of the BRAM to each enclave and connects the hardware components automatically.

The Hw-Att Module. If an enclave uses hardware-based attestation, HardwareBuilder will automatically connect the trusted hardware Hw-Att module to it. The Hw-Att is implemented in RTL and synthesized to a bitstream, and it can be formally verified. It is connected directly to the entire BRAM of the enclave and can operate on the enclave's BRAM directly. The Hw-Att is responsible for decrypting an SSA, computing the measurements, and signing a measurement report. Cryptographic keys can be embedded in the Hw-Att's own BRAM to prevent other hardware or software components from accessing them. In attestation under the EnhancedModel, the Hw-Att serves as the *root of trust for measurement* and *reporting*.

4.4 Bootstrapping Trust in Enclaves

Software running on the hardcore system can configure the FPGA by sending a protected bitstream to the FPGA configuration module (FCM), which is a trusted hardware module. The FCM verifies the bitstream using the device keys. Upon a successful verification, the FCM decrypts the bitstream using the device key and configures the FPGA. After this step, the softcore CPU, Hw-Att in the EnhancedModel, and the interconnections among hardware modules will be configured. In addition, Firmware on the softcore CPU starts execution. A measurement $m = H(bitstream)$ is generated by the FCM and placed on the shared DRAM region with the enclave for the future use of attestation report generation. In the BaseModel, 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. Under the EnhancedModel, Firmware can be compromised, so it requests the Hw-Att to decrypt and verify the SSA. Because the Hw-Att is also encrypted at rest and only has the keys on its own BRAM at runtime, the keys are secure.

4.5 Executing SSAs in Enclaves

To create an SSA, the developer links the code against the Firmware. After launching 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 3, to have two-way data transmissions with UA on the hardcore

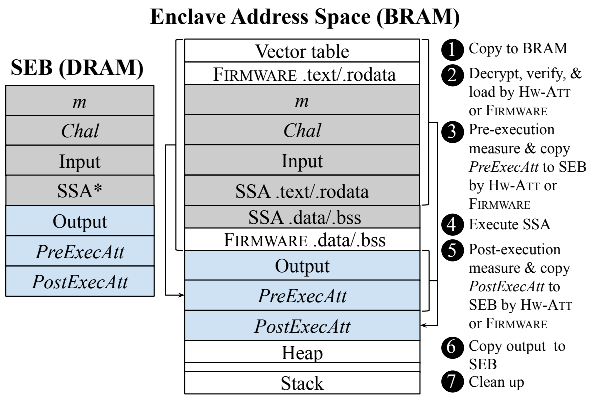


Figure 3: SSA Execution Block (SEB) layout, simplified enclave address space layout, and steps in executing an SSA.

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 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 §4.6.

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 3, 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 copies SSA*, *Chal*, and input data in the SEB from DRAM to its own BRAM ❶. The FIRMWARE can disable LdExec* interrupts after data is copied.

Under the BaseModel, the FIRMWARE then decrypts and verifies the encrypted SSA* using the corresponding developer's keys ❷. Under the EnhancedModel, the FIRMWARE requests the HW-ATT to decrypt and verify the encrypted SSA*. 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.

4.6 Remote Attestation Mechanisms

BYOTEE provides two attestation mechanisms, namely pre-execution and post-execution attestations. The former extends remote attestation of code and input integrity with bitstream, whereas the latter extends the output data integrity attestation [60]. Note that BYOTEE only provides the mechanism for attestation, which can support sophisticated attestation protocols. With the help of trust bootstrapping discussed in §4.4, the measurement mechanism not only captures the identity of the loaded SSA but also the bitstream, including hardware configurations and the FIRMWARE. Note that it is critical to perform measurement on the BRAM since the DRAM can be changed asynchronously by the hardcore system.

Software-based Attestation under the BaseModel. 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*, *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 developer key or other shared keys to compute the measurement. In post-execution attestation, after the SSA finishes execution ❹ the FIRMWARE computes a measurement *PostExecAtt* on the vector table, FIRMWARE code and read-only data, *m*, *Chal*, input data, output data generated by the SSA, SSA's read-only sections and *PreExecAtt*, and copies the measurement to the DRAM ❺.

Hardware-based Attestation under the EnhancedModel. The FIRMWARE does not perform the measurements under the EnhancedModel because it does not possess keys. Instead, it requests the HW-ATT to compute the measurements. As discussed in §4.3 and shown in §7.4, the HW-ATT is directly connected to the BRAM of the enclave, and it computes the measurements by reading the contents from the BRAM. The HW-ATT then signs the measurements with the its keys and returns them to the FIRMWARE or REE, which then sends the signed measurements to the remote verifier.

4.7 Multiple Inputs to SSA

In the case that the UA on the hardcore system needs to continuously send data to the SSA, e.g., not all input data is available at the beginning, the size of 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 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.

4.8 Optional Multiple Protected SSA Sessions

As an option, an enclave can also interleave the execution of multiple SSAs with proper hardware re-initialization, e.g., flush cache, reset all memory, etc. To this end, BYOTee defines two 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 Firmware cleans up BRAM and awaits new requests. When a ReExec interrupt is raised, Firmware retrieves an encrypted blob from SEB. Upon a successful signature verification, Firmware loads the decrypted memory contents to the right locations, restores the registers, and resumes the SSA execution.

5 APPLICATIONS AND DEVELOPER'S PERSPECTIVE

In this section, we use four 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, SSAs, and UAs using the BYOTee toolchain.

5.1 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 runtime. We implemented 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. These applications interact with peripherals but do not communicate with other SSAs or the hardcore system. For instance, cyber-physical applications that read from sensors, make local decisions, and control an actuator fall under this category. Besides the attacks BYOTee protects the computational applications from, BYOTee protects the paths between the SSA and peripherals from attacks. To demonstrate this protection, we developed an LED toggler SSA (SSA-2) that utilizes a button and an LED. Both the button and LED are solely connected to the enclave of SSA-2, rendering them inaccessible by the hardcore system or other enclaves.

Peripheral- and Hardcore System-Interacting Applications. For demonstration, we developed a music player with digital rights management that guarantees the confidentiality, integrity, and authenticity of songs. 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

```
{"Enclaves": [
  {"Name": "Enclave-a",
   "Processor":
    {"Type": "MicroBlaze 32bit", "Debugging": "Enabled"},
   "Memory Size": "512KB",
   "Shared DRAM SEB":
    {"Base": "0x20000000", "Size": "2MB"},
   "Attestation": "Hardware"},
  {"Name": "Enclave-b",
   "Processor":
    {"Type": "VexRisc 32-bit",
       "Data Cache": "16KB", "Instruction Cache": "16KB",
       "FPU": "F32", "Debugging": "Disabled"},
   "Memory Size": "32MB",
   "Shared DRAM SEB": {
     "Base": "0x20000800", "Size": "128MB"},
   "Attestation": "Software"}},
  {"Name": "Enclave-c",
   "Processor":
    {"Type": "A2I 64bit", "Data Cache": "64KB",
       "Instruction Cache": "64KB",
       "MMU": "Enabled", "MMU Page Size": "4KB",
       "FPU": "AXU", "Debugging": "Disabled"},
   "Memory Size": "64MB",
   "Shared DRAM SEB": {
     "Base": "0x20020800", "Size": "256MB"},
   "Attestation": "Software"}}],
 "Peripherals": [
  {"Type": "AXI Gpio",
   "Board Interface": "Btns 2bits",
   "Access": ["Hardcore system", "Enclave-b"]},
  {"Type": "Uart Lite 8bit",
   "Baud Rate": "115200",
   "Access": ["Enclave-a"]},
  {"Type": "Dual Port BRAM Generator",
   "Base Address": "0x1F0000", "Size": "2MB",
   "Access": ["Enclave-a", "Enclave-c"]}]}
```

Listing 1: An example hardware description defining three enclaves in JSON format.

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.

We emphasize that any software solution that solely trusts SGX or TrustZone cannot meet the security requirements of this music player because (i) there is no 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 [61], attempt to address this issue but they 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 developed 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.

5.2 Developer's Perspective

Creating Enclave Hardware. A developer can use the HARDWAREBUILDER to design and create the hardware consisting of one or multiple enclaves. Listing 1 shows an example hardware description in JSON format of three enclaves. Enclave-a has a 32-bit MicroBlaze softcore CPU [62] and uses hardware-based attestation. Enclave-b has a 32-bit VexRisc softcore CPU [34], Enclave-c uses a 64-bit A2I softcore CPU [37]. The softcore CPUs of Enclave-b and Enclave-c have FPUs, instruction, and data caches. Hardware-based attestation and debugging is enabled on Enclave-a only, for which HARDWAREBUILDER inserts the Hw-ATT and a debugging module. A DRAM region is reserved for the SEB of each enclave, respectively. A UART peripheral is only connected to the Enclave-a 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-b but cannot be accessed by Enclave-a or Enclave-c. Each enclave shares a DRAM region with the hardcore system for two-way enclave-to-hardcore System communication. Enclave-a and Enclave-c can also use the shared BRAM region to communicate.

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

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

Then, the developer invokes the synthesizer tool with the script as input. The `-n` parameter specifies the name of the hardware project, and the `bf` parameter specifies the mode of operation, which includes generating bitstream, combining bitstream with FIRMWARE, etc. The output of the HARDWAREBUILDER 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. Software modules developed in any language that can be linked against the FIRMWARE can be included in an SSA. For example, SSAs developed in C can have their own `main` functions with a declaration of `int main() __attribute__ ((section (".text.ssa_entry")))`. Not all libc functions are available for the SSAs to use. 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 hardcore system and uses a UIO interface to communicate with the FIRMWARE and SSA. The developer uses the SSAPACKER to generate protected SSA binaries.

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

6 SECURITY ANALYSIS

We conduct an informal security analysis of BYOTEE, in which we discuss the attacks BYOTEE can and cannot defend.

Compromised Hardcore System. Even if the hardcore system software is compromised at runtime, the attacker cannot access the data on/from enclave hardware resources, because they are in the isolated address space of the target enclave. The attacker cannot breach the confidentiality of bitstream, SSA code and data at rest as well, because they are encrypted at build time. The enclave-hardcore system's two-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.

Compromised FIRMWARE and SSAs. In the BaseModel, we assume FIRMWARE and SSAs are bug-free and cannot be compromised. Under the EnhancedModel, if FIRMWARE and SSAs are compromised at runtime by malicious input sent by the hardcore system, they 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. Therefore, the attack is confined within the compromised enclave. The compromised FIRMWARE and SSA cannot extract the keys from the Hw-ATT to forge measurement reports, since the BRAM of Hw-ATT is not connected to the enclave.

Malicious Hardware IPs and 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.

Cold-boot Attack. While cold-boot attacks on DRAM at room temperature are proven very effective [48], attacks on SRAM without external power sources are less feasible [49]. 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, the program states, are placed on an enclave or the Hw-ATT'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 [63]; (ii) even without initialization, the contents in BRAM decays faster [49]; (iii) BRAM is embedded on-chip and cannot be physically taken out, so attackers have to bypass software protections to dump its content.

Cache Side-channel. Because the CPU is time-shared between the REE and TEE in SGX and TrustZone, cache side-channel attacks are effective [15–18]. 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.

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 on other modules on the same FPGA or a CPU on the same SoC [57]. BYOTee cannot mitigate these attacks but can prevent 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 [64, 65]. Similar to power side-channel attacks, BYOTee cannot prevent these attacks directly.

7 IMPLEMENTATION AND EVALUATION

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 (≈$130).

7.1 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. 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, and two Pmod connectors. We connect a Pmod I2S2 audio input and output device [66] to the board for SSA-3 evaluation. Figure 7 shows the top and bottom view of the board with the connected audio device.

7.2 BYOTee Implementation

We implemented the BYOTee infrastructure and toolchain, which include HardwareBuilder, Hw-Att for the EnhancedModel, SSAPacker, and Firmware, for the Xilinx SoC FPGA. The HardwareBuilder was developed in Python (2.5K SLOC). Hw-Att can be developed in VHDL or C on a softcore CPU. In our implementation, Hw-Att includes 1400 SLOC C code. The SSAPacker includes 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 for the BaseModel (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 uses AES-256 for SSA encryption and SHA512-HMAC to protect the integrity and authenticity of SSAs. We use the BLAKE2 [67] hash algorithm to implement the pre-execution- and post-execution-attestations. 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. The BYOTee toolchain also includes scripts to automate the steps from synthesizing the hardware, compiling SSAs and Firmware, and formatting the SD card with partitions.

7.3 Customized Enclaves for the Example SSAs

We specified the hardware description for the SSAs in §5.1 and used the HardwareBuilder and synthesizer to generate the bitstream. All the enclaves are configured with a 32-bit Microblaze CPU (version 10.0, 100MHz, no instruction/data cache, no 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 4 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 isolation at circuit level of the enclaves from each other and from the hardcore processor. Figure 4(a) shows the hardware design with a debugger, whereas all other designs do not have a debugger for the minimum hardware TCB. Table 1 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, etc. 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.

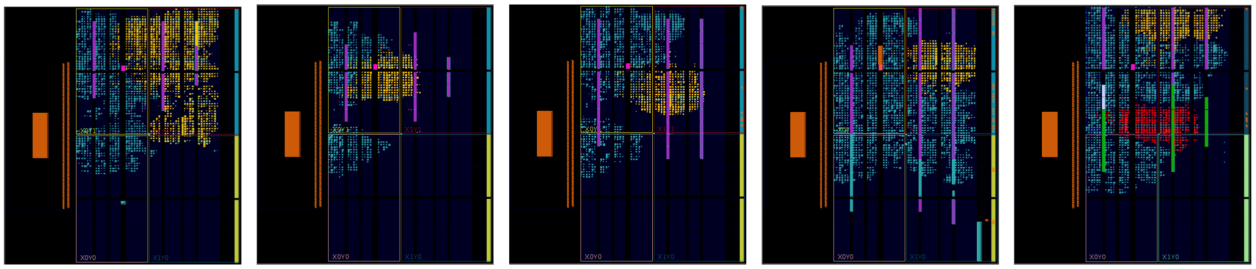
7.4 Security Evaluation

Circuit-level Execution Isolation. As shown in Figure 8(e), the two enclaves for SSA-4 are isolated at the circuit level from each other and the hardcore system (`processing_system7_0`). The figure shows Enclave-1's instruction memory controller (`Enclave_1_ILMB`), data memory controller (`Enclave_1_DLMB`), and memory generator (`blk_mem_gen_0`) are only connected to the `Enclave_1` CPU, where `Enclave_4_local_memory` (the combination of two memory controllers and one generator) is only connected to the `Enclave_4` CPU. To share a BRAM region for communication, each enclave has another memory controller, i.e., `share_axi_bram_ctrl_0` and `share_axi_bram_ctrl_0`, which is connected to the shared memory generator (`share_blk_mem_gen_1`).

Isolated Path to Peripherals. The hardware design in BYOTee ensures isolated paths between SSAs and peripherals. As shown in Figure 8(d), the I2S output audio peripheral (`i2s_output_1`) for SSA-3 is only connected to its enclave but is isolated at the circuit level from the hardcore system.

Hardware-based Attestation. Figure 5 shows the block diagram of the SSA-1 with hardware-based attestation. Compared with Figure 8(b), which uses software-based attestation, we can see the Hw-Att is connected to the BRAM of Enclave-1 (`Enclave_1_exe_memory`).

Software TCB Size. Table 2 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 runtime 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 [69] has over 117K SLOC, or SGX application, e.g., the Gramine library OS [70] has 83K SLOC.



(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

Figure 4: Resource footprints of the enclaves (software-based attestation profile) with MicroBlaze softcore CPUs for the example SSAs on the Cora Z7-07S. The yellow and red portions represent CPU cells. The purple portion represents BRAM cells. The pink rectangle represents 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 debugging modules, interconnects.

Table 1: Resource utilization of enclaves (software-based attestation profile) 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 2: 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

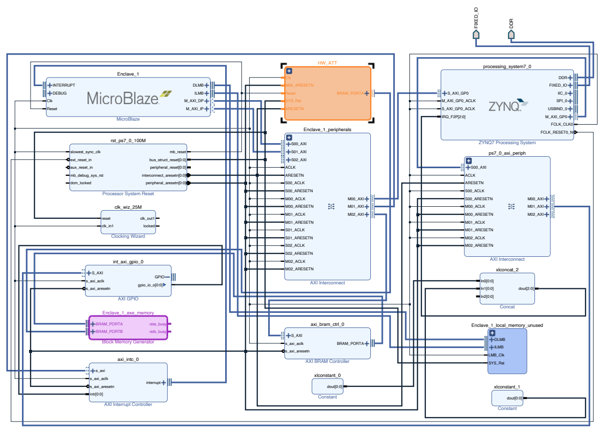
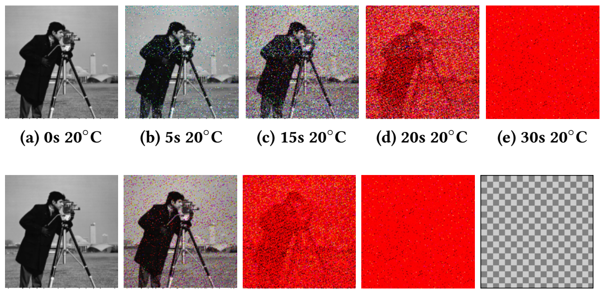


Figure 5: Block diagram of SSA-1 with the Hw-Att module (hardware-based attestation profile).



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

(f) 0s -18°C (g) 1m -18°C (h) 10m -18°C (i) 13m -18°C (j) BRAM initialization

Figure 6: Visualizing cold-boot attacks on DRAM and BRAM on the same Z7-07S board. We loaded a bitmap image (150×150 pixel; 90.1kB) on the DRAM and BRAM. The reconstructed image from the fully decayed DRAM is red because half of the cells are 1s and the other half are 0s. The image from the BRAM is transparent because it is initialized to 0s.

Cold-boot Attacks on DRAM and BRAM. Cold-boot attacks on DRAM are a serious problem, especially when an attacker has physical access to the device. We evaluated the feasibility of cold-boot attacks on DRAM and BRAM on the same board. In these experiments, we loaded a bitmap image (150×150 pixel; 90.1kB) and measured the DRAM decay at room temperature (20°C/68°F) and -18°C/0°F after power reset (0 second) and losing power for different intervals, e.g., 30 seconds, 13 minutes. 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 §6, even if the BRAM is not initialized, cold-boot attacks on it are more difficult than on DRAM. Figure 6

Table 3: Performance Evaluation of Firmware on MicroBlaze CPU (Software-based attestation; Time in Milliseconds; Size in Bytes). The experiments demonstrate that SSA-3 efficiently verify, decrypt, and play 48KHz WAV on a 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 4: Performance evaluation of Embench-IoT on softcore MicroBlaze CPU (Version 10.0, 100MHz, no Cache, no FPU) and hardcore Cortex-M4 CPU (16MHz, no Cache, no FPU, officially reported performance from [68]) in Milliseconds

Application	Description	M4 [68]	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

visualizes the cold-boot attack results, which confirms cold-boot attacks on DRAM are feasible but not on BRAM.

7.5 Performance and Power Evaluation

We evaluate the performance of the low-end MicroBlaze-powered enclaves, which provides a lower-bound performance estimation of available softcore CPUs. We evaluate the performance using 12 Embench-IoT benchmark applications [71] and evaluate the BYOTee software performance by measuring the time cost of different Firmware operations.

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 [71]. As Table 4 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.

Firmware Performance. We evaluate the time Firmware spends on the loading, decrypting, integrity and authenticity verification, attestation, cleaning up, suspending, and restoring operations of three SSAs. As Table 3 shows, the time spent by Firmware is linear to the size of the SSA and its data. 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 attestations cost around 124.77 ms for every 10,000 bytes. Cleaning up 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.

Power Consumption. The Xilinx Vivado (2017.4) tool provides the estimated power consumption of the hardcore CPU and example enclaves. The 667MHz Cortex-A9 hardcore CPU uses around 1,255 mW. The SSA-1 enclave, which includes a 100MHz MicroBlaze, BRAM, etc., uses 66 mW. The SSA-2 consumes 72 mW, SSA-3 consumes 205 mW, and SSA-4 consumes 117 mW. The hardware-based attestation profile of SSA-1 consumes 101 mW. For reference, a 16MHz Cortex-M4 consumes 0.66 mW [72].

8 RELATED WORK

Many software- or hardware-based solutions have been proposed to address one or more limitations of existing TEEs. Among them, TEEOD [83] is most related. Compared to TEEOD, BYOTee offers additional security features, such as trust bootstrap, software- and hardware-based attestation. Table 5 highlights the advantages of BYOTee and compares it to related work. Moreover, we discuss previous efforts on addressing the single TEE issue, isolated I/O paths, and the limitations of other hardware-based solutions.

The Single TEE Issue of TrustZone. vTZ [80] provides each virtual machine with a virtualized TEE by running a monitor within the secure world. Sanctuary [20] utilizes the memory access controller to provide multi-domain isolation. TrustICE [79] creates multiple computing environments in the normal domain and runs a monitor in the secure world. uTango [81] use the secure attribution unit of Cortex-M to create multiple secure execution environments. On RISC-V, KeyStone [77] 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.

Isolated I/O Paths and Mitigating Side-channel Attacks. Cure [3] enables the exclusive assignment of system resources to single enclaves. Composite Enclaves [14] builds on top of KeyStone [77] and extends the TEE to several hardware components. HECTOR-V [21] uses a dedicated processor as a TEE with configurable peripheral permissions. Cure, Composite Enclaves, and HECTOR-V rely on the PMP feature of RISC-V. SGXIO [61] presents a hypervisor-based trusted path architecture for SGX. SGX-FPGA [76]

Projects		Benefits											
	Underlying Hardware Primitive	Customizable CPU and Memory (*G1*)	Customizable Peripheral Connections (*G1*)	Remote Hardware Attestation (*G2*)	Remote Software Attestation (*G2*)	Post-Execution Attestation (*G2*)	Multiple TEEs (*G4*)	Cache Side-channel Attack Resistant (*G5*)	Concurrent TEE and REE Execution (*G5*)	TEE with Dedicated Hardware (*G5, G6*)	Allowing for Minimum Software TCB (*G8*)	Cold-boot Attack Resistant	Deployable on Commodity Devices
Flickr [73]	S/T				✓	✓					✓		✓
TrustVisor [74]	S/T				✓	✓					✓		✓
Haven [75]	X				✓	✓	✓						✓
SGXIO [61]	X+H		✓				✓						✓
SGX-FPGA [76]	X+F		✓	✓	✓		✓						✓
KeyStone [77]	R				✓		✓						✓
Sanctum [78]	R		✓		✓	✓	✓	✓					
CURE [3]	R		✓				✓	✓					✓
Composite Encl. [14]	R		✓	✓	✓		✓						✓
SANCTUARY [20]	A				✓		✓				✓		✓
TrustICE [79]	A				✓	✓	✓						✓
vTZ [80]	A+H						✓						✓
Ambassy [56]	A+F	–	✓	✓	✓			✓	✓	–	–	✓	✓
uTango [81]	M				✓		✓						✓
Graviton [22]	G				✓		✓		✓			✓	
StrongBox [82]	G				✓		✓		✓			✓	✓
HECTOR-V [21]	N		✓				✓	✓	✓	✓	✓	✓	
TEEOD [83]	F	✓					✓	✓	✓	✓	✓		✓
Sancus [84]	N				✓						✓		
SecureBlue++ [85]	-										✓		✓
TrustLite [86]	N				✓		✓				✓		
SMART [87]	N				✓	✓							
MyTEE [88]	H		✓								✓		✓
MeetGo [89]	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 5: Comparing the security goals and benefits of BYOTEE with other software- and hardware-based TEE solutions

builds a secure path between CPU and FPGA. To eliminate side-channel attacks, Sanctum [78] combines invasive hardware modifications with a trusted software monitor on RISC-V.

Building TEEs with Other Hardware. Graviton [22] and StrongBox [82] offload security-sensitive code and data to a GPU. Ambassy [56] and MeetGo [89] use FPGA to construct TEEs, but they do not include softcore CPUs. Dedicated processor solutions, such as Google Titan [90], Samsung eSE [91], and Apple SEP [92], use external connections between the REE and TEE, making them vulnerable to physical probing attacks [21]. BYOTEE is also inspired by other isolated execution environment solutions, including Flickr [73], TrustVisor [74], and Haven [75].

9 LIMITATIONS AND DISCUSSIONS

TOCTOU. The presented hardware-based attestation is susceptible to TOCTOU attacks, which can be addressed by implementing SACHa [93] or RATA [94] on top of BYOTEE. SACHa presents a self-attestation framework of FPGA without a trusted hardware module, while RATA addresses the TOCTOU attacks by using a hardware component to provide the context of software.

Low Maximum Clock Frequency and Power Consumption of FPGA. Even though softcore CPUs on FPGA have a low maximum clock frequency and their power consumption is always higher than hardcore CPUs with comparable performance, our experiments on a very low-end SoC FPGA device in §7 demonstrate the practicality of BYOTEE. As the gap between FPGAs and Application Specific Integrated Circuits (ASICs) keeps reducing [95] and vendors integrate FPGAs into products, e.g., AMD EPYC CPUs, it is feasible to deploy BYOTEE on end devices.

Preventing Replay Attacks on Encrypted SSAs with Reference Numbers. The current design of the SEB in SSAs is vulnerable to replay attacks. However, this issue can be effectively addressed by including a pair of unique reference numbers for the communicating parties. These reference numbers act as identifiers that help prevent attackers from intercepting and replaying the messages.

10 CONCLUSION

Even though hardware-assisted TEEs have been widely adopted, they suffer from several issues that make them untrustworthy and ineffective, including static and fixed hardware TCBs and a lack of dynamic attestation of the hardware. In this paper, we present BYOTEE, a framework for building multiple TEEs on-demand with configurable hardware and software TCBs, utilizing commodity SoC FPGA devices. BYOTEE establishes a dynamic root of trust that allows for full isolation and untampered execution of security-sensitive applications in enclaves from pre-existing software on the hardcore system. In BYOTEE, enclaves, which include softcore CPUs, memory, and peripherals, are created on the FPGA, and the BYOTEE firmware provides necessary software libraries for the applications to use. Additionally, BYOTEE offers software- and hardware-based attestation mechanisms to verify the hardware and software stacks. We implemented BYOTEE on the Xilinx FPGA, and our evaluation results on the low-end Zynq-7000 system for benchmark applications and example SSAs demonstrate the effectiveness and performance of BYOTEE.

ACKNOWLEDGMENT

We thank Paul Ratazzi, Zhenhua Wu, Kai Ni, and Arnob Paul for the discussions while this work was in progress. We thank Sandro Pinto for bringing TEEOD to our attention. We also thank the organizers of MITRE eCTF 2020 for inspiring this research. This material is based upon work supported in part by a National Science Foundation (NSF) grant (2237238), a National Centers of Academic Excellence in Cybersecurity grant (H98230-22-1-0307), and the Air Force Visiting Faculty Research Program. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government or any agency thereof.

REFERENCES

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

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

[3] 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.

[4] 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.

[5] X. Ge and T. Jaeger, "Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture," in *Mobile Security Technologies Workshop (MoST)*, 2014.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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)*, 2020.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *USENIX Workshop on Offensive Technologies*, 2017.

[18] 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.

[19] 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.

[20] 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.

[21] P. Nasahl, R. Schilling, M. Werner, and S. Mangard, "HECTOR-V: A heterogeneous CPU architecture for a secure RISC-V execution environment," in *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2021.

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

[23] J. Wilson, "AMD will infuse EPYC CPUs with Xilinx-based FPGA AI Engines, starting as early as 2023." https://wccftech.com/amd-will-infuse-epyc-cpus-with-xilinx-based-fpga-ai-engines-starting-as-early-as-2023/, -.

[24] "Project Catapult." https://www.microsoft.com/en-us/research/project/project-catapult/.

[25] "Project Brainwave." https://www.microsoft.com/en-us/research/project/project-brainwave/.

[26] "Amazon EC2 documentation." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/DocumentHistory.html, -.

[27] 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)*, 2015.

[28] 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)*, 2016.

[29] 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)*, 2019.

[30] W. N. Chelton and M. Benaissa, "Fast elliptic curve cryptography on fpga," *IEEE transactions on very large scale integration (VLSI) systems*, 2008.

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

[32] M. Lavasani, H. Angepat, and D. Chiou, "An fpga-based in-line accelerator for memcached," *IEEE Computer Architecture Letters*, 2013.

[33] "Open Cores." https://opencores.org/.

[34] "VexRiscv." https://github.com/SpinalHDL/VexRiscv, 2022.

[35] "Neo430." https://github.com/stnolting/neo430, 2020.

[36] "Microwatt." https://github.com/antonblanchard/microwatt.

[37] "A2I." https://github.com/openpower-cores/a2i.

[38] "A2O." https://github.com/openpower-cores/a2o.

[39] "OpenSPARC T1 Softcore Processor." https://www.oracle.com/servers/technologies/opensparc-t1-page.html.

[40] "libreSOC." https://libre-soc.org/.

[41] R. Lysecky and F. Vahid, "Design and implementation of a microblaze-based warp processor," *ACM Transactions on Embedded Computing Systems (TECS)*, 2009.

[42] "Intel NIOS softcore." https://www.intel.com/content/www/us/en/products/details/fpga/nios-processor/, 2020.

[43] 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)*, 2017.

[44] 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 *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019.

[45] V. Sieh, O. Tschache, and F. Balbach, "Verify: Evaluation of reliability using vhdl-models with embedded fault descriptions," in *IEEE International Symposium on Fault Tolerant Computing*, 1997.

[46] 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)*, 1997.

[47] Xilinx, "Zynq-7000 SoC Technical Reference Manual." https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, 2021.

[48] 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.

[49] 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.

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

[51] 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.

[52] "Arm Platform Security Architecture Security Model." https://armkeil.blob.core.windows.net/developer/Files/pdf/PlatformSecurityArchitecture/Architect/DEN0079-PSA_SM_ALPHA-02.pdf.

[53] "PSA Attestation API ." https://armkeil.blob.core.windows.net/developer/Files/pdf/PlatformSecurityArchitecture/Implement/IHI0085-PSA_Attestation_API-1.0.1-2.pdf.

[54] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments," in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[55] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell, "Design and verification of the arm confidential compute architecture," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 465–484, 2022.

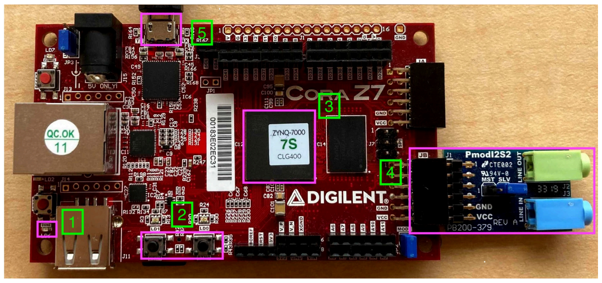
[56] D. Hwang, S. Yeleuov, J. Seo, M. Chung, H. Moon, and Y. Paek, "Ambassy: A Runtime Framework to Delegate Trusted Applications in an ARM/FPGA Hybrid System," *IEEE Transactions on Mobile Computing (TMC)*, 2021.

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

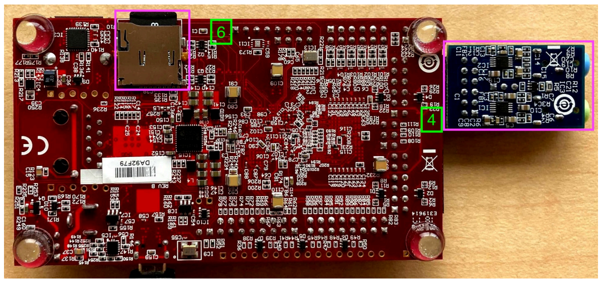
[58] Xilinx, "Xilinx Vivado Toolkit." https://www.xilinx.com/products/design-tools/vivado.html.

[59] 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?
[60] 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.
[61] 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.
[62] Xilinx, "MicroBlaze." https://www.xilinx.com/products/design-tools/microblaze.html, 2018.
[63] Xilinx, "7 Series FPGAs Memory Resources.." https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf, 2019.
[64] 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.
[65] 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.
[66] "Pmod I2S2: Stereo Audio Input and Output." https://store.digilentinc.com/pmod-i2s2-stereo-audio-input-and-output/, 2018.
[67] "BLAKE2—fast secure hashing." https://www.blake2.net/, 2015.
[68] "Embench IoT benchmark Cortex-M4 data." https://gitlab.inria.fr/mescoute/embench-iot/-/tree/76e887fac691d3d3f42cd32636b347bf2626036b/doc.
[69] Linaro, "Trusted Firmware M (TFM) v1.3.0." https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git.
[70] "Gramine Project." https://github.com/gramineproject/gramine.
[71] "Embench IoT." https://github.com/embench/embench-iot, 2021.
[72] ARM, "Arm Cortex-M4 Processor Datasheet." https://developer.arm.com/documentation/102832, 2020.
[73] 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.
[74] 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.
[75] 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.
[76] 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.
[77] 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.
[78] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *USENIX Security Symposium*, 2016.
[79] 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.
[80] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing ARM trustzone," in *USENIX Security Symposium*, 2017.
[81] D. Oliveira, T. Gomes, and S. Pinto, "utango: an open-source tee for iot devices," *IEEE Access*, 2022.
[82] Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao, *et al.*, "Strongbox: A gpu tee on arm endpoints," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.
[83] C. R. Sergio Pereira, David Cerdeira and S. Pinto, "Towards a Trusted Execution Environment via Reconfigurable FPGA," *arXiv preprint arXiv:2107.03781*, 2021.
[84] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *2USENIX Security Symposium*, 2013.
[85] R. Boivie and P. Williams, "Secureblue++: Cpu support for secure execution," *IBM Research Division*, 2012.
[86] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *European Conference on Computer Systems*, 2014.
[87] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "Smart: secure and minimal architecture for (establishing dynamic) root of trust.," in *Network and Distributed System Security Symposium (NDSS)*, 2012.
[88] S. Han and J. Jang, "Mytee: Own the trusted execution environment on embedded devices,"
[89] H. Oh, K. Nam, S. Jeon, Y. Cho, and Y. Paek, "MeetGo: A Trusted Execution Environment for Remote Applications on FPGA," *IEEE Access*, 2021.
[90] 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*, 2018.
[91] Samsung, "eSE Safeguard against digital attacks.." https://www.samsung.com/semiconductor/security/ese/, 2020.
[92] Apple, "Security enclave processor for a system on a chip." https://patents.google.com/patent/US8832465, 2020.
[93] J. Vliegen, M. M. Rabbani, M. Conti, and N. Mentens, "SACHa: Self-attestation of configurable hardware," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.
[94] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the TOCTOU problem in remote attestation," in *ACM Conference on Computer and Communications Security*, 2021.
[95] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *IEEE Transactions on computer-aided design of integrated circuits and systems*, 2007.
[96] Xilinx, "MicroBlaze Debug Modulev3.2." https://www.xilinx.com/support/documentation/ip_documentation/mdm/v3_2/pg115-mdm.pdf, 2021.
[97] Xilinx, "LogiCORE IP Product Guide." https://www.xilinx.com/support/documentation/ip_documentation/axi_timer/v2_0/pg079-axi-timer.pdf, 2016.
[98] Xilinx, "AXI GPIO v2.0." https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf, 2016.
[99] Xilinx, "AXI DMA v7.1." https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf, 2019.
[100] Xilinx, "AXI4-Stream FIFO v4.1." https://www.xilinx.com/support/documentation/ip_documentation/axi_fifo_mm_s/v4_1/pg080-axi-fifo-mm-s.pdf, 2016.
[101] Xilinx, "XADC Wizard v3.3." https://china.xilinx.com/support/documentation/ip_documentation/xadc_wiz/v3_3/pg091-xadc-wiz.pdf, 2016.

A EXPERIMENT DEVICE



(a) Top view



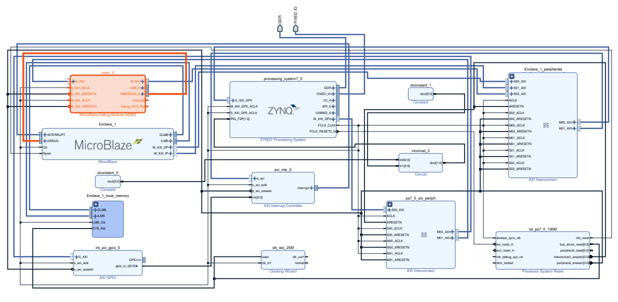
(b) Bottom view

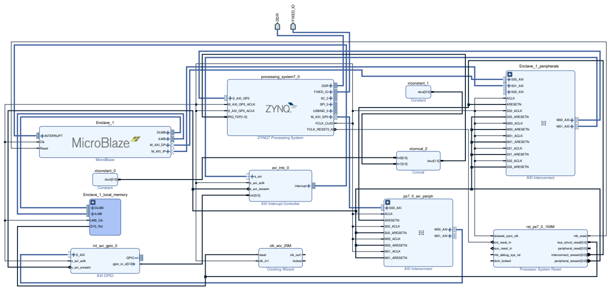
Figure 7: The Z7-07S board with a Pmod audio module for experiments and evaluation: (1) an LED as an Enclave-2 peripheral used by SSA-2, (2) a button as an Enclave-2 peripheral used by SSA-2, (3) the Zynq-7000 SoC with a hardcore CPU and FPGA, (4) a Pmod port and the I2S2 stereo audio input and output 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.

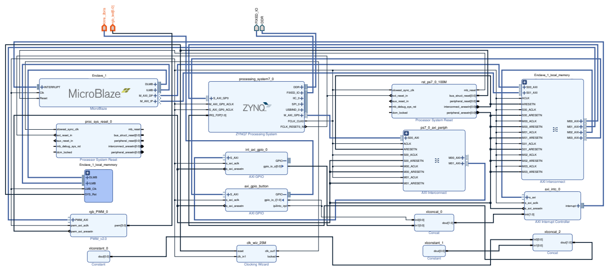
Figure 7 shows the top and bottom view of the Cora Z7-07S development board with a single-core 667MHz Arm Cortex-A9 processor, a Xilinx Zynq-7000 FPGA, and the connected Pmod I2S2 audio device.

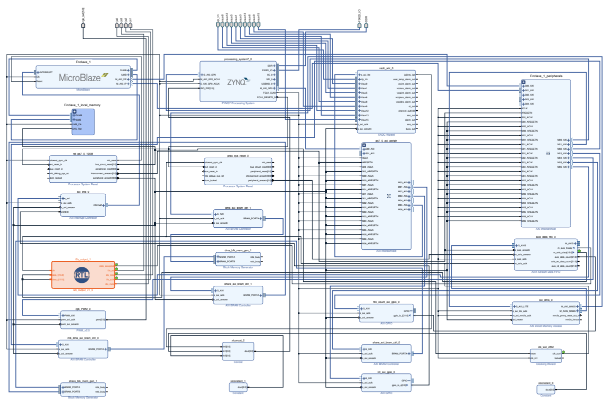
B AUTOMATICALLY GENERATED HARDWARE DESIGN OF THE EXAMPLE SSAS

Figure 8(a) shows the hardware for SSA-1 with a MicroBlaze Debugging Module (MDM) [96] 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 [97] connects the `RGB_LED` ports, and an AXI GPIO IP [98] 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 transmitter RTL (SPI) is added to interact with the Pmod I2S2 module Codec module. Additionally, an AXI direct memory access IP [99] with BRAM and an AXI stream data FIFO IP [100] 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 [101], 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 has 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

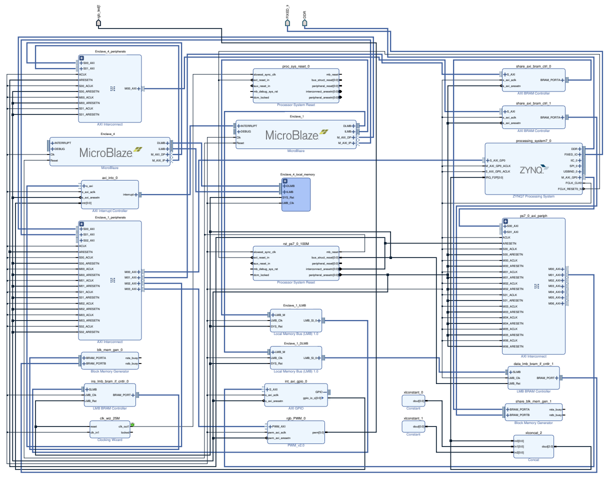

(e) Hardware for SSA-4 without a Debugging Module

Figure 8: Block diagrams of the hardware designs of the four enclaves for the four example SSAs (software-based attestation profile). The ZYNQ processing system (`processing_system7_0`) represents the hardcore Cortex-A processor. Each of the enclaves for SSA-1, SSA-2, and SSA-3 has one MicroBlaze softcore CPU, whereas SSA-4 has two enclaves. 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.