Trusted Execution Environments in Embedded and IoT Systems: A CactiLab Perspective

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

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

Xi Tan
CactiLab
University at Buffalo
Buffalo, USA
xitan@buffalo.edu

Zheyuan Ma
CactiLab
University at Buffalo
Buffalo, USA
zheyuanm@buffalo.edu

***Abstract*—While the benefits of networked embedded and Internet of Things (IoT) systems are unparalleled, they are susceptible to cyberattacks. In recent years, Trusted Execution Environments (TEE) have been offered in CPUs of embedded and IoT platforms as a foundational primitive for security to keep code and data loaded inside protected, with respect to confidentiality and integrity, from Rich Execution Environments (REEs). The hardware and software layers of existing TEEs nevertheless have been criticized for lack of transparency, full of vulnerabilities, and various restrictions, which means the existing TEEs and TEE-based security solutions are untrustworthy, ineffective, or inefficient. Failure to make TEEs trustworthy and effective will backfire instead of enhancing security because embedded TEEs usually have the highest privilege and a compromised TEE can completely sabotage the REE. In this paper, we present our perspective on the challenges and limitations related to embedded and IoT TEEs. Additionally, we delve into three recently published projects from CactiLab, which aim to tackle challenges presented in embedded and IoT TEEs and TEE-based security solutions at various layers.**

***Index Terms*—Trusted execution environment; embedded and IoT systems; Arm Cortex-M TrustZone**

I. Introduction

Networked embedded and Internet of Things (IoT) systems, including those based on microcontrollers, microprocessors, and Field Programmable Gate Arrays (FPGAs), are essential to everyday life and are predicted to reach 1 trillion by 2035 [1]. As shown in Figure 1, these systems power a variety of IoT devices, such as sensors, medical devices, wearables, smart family gadgets, industrial computing units, autonomous vehicles, and infotainment systems.

While the benefits of these systems are unparalleled, they are susceptible to cyberattacks, which are occurring at unprecedented levels and often have severe consequences ranging from loss of life [2] to homeland security breaches [3], [4]. For instance, in June 2019, the FDA issued an emergency warning about potential life-threatening cyberattacks on some diabetes patients' insulin pumps, affecting more than 4,000 individuals in the U.S. [2]. Therefore, it is imperative to ensure our embedded and IoT infrastructure and ecosystem are built on a trustworthy and secure foundation.

It is, however, difficult to secure these systems due to software issues and hardware constraints. On the software front, these systems are usually written in low-level languages,

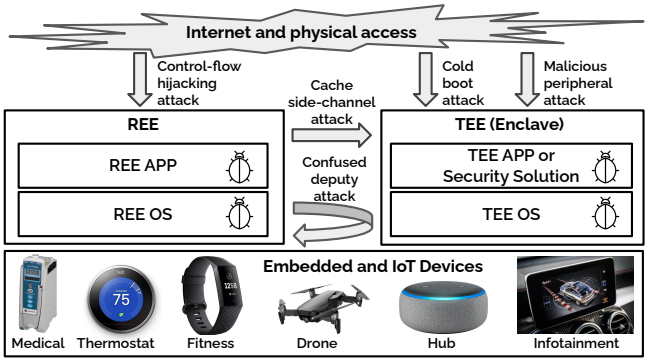

Fig. 1. The threats embedded and IoT Systems with TEEs face.

e.g., C/C++, whose lack of safety allows attackers to exploit memory corruption bugs to hijack the control flow [5]. While Data Execution Prevention (DEP) and W⊕X [6] can defeat code injection attacks, these systems are still vulnerable to code reuse attacks, e.g., Return-Oriented Programming (ROP) [7]–[9]. The situation is exacerbated when many of these systems are Real-Time Operating Systems (RTOS) that run applications at the privileged level or are bare-metal systems in which applications execute directly on hardware without an OS. Due to the lack of security and fault isolation, a bug anywhere may lead to a crash or to full control by attackers.

On the hardware front, many embedded and IoT systems are powered by microcontrollers, e.g., Arm Cortex-M microcontrollers. Such microcontrollers are streamlined from microprocessors, and they do not have some of the hardware units we take for granted on microprocessors. For example, the Arm Cortex-M microcontrollers do not have a Memory Management Unit (MMU), without which applications share the same physical address space, making it difficult to enforce isolation or to implement effective defenses for memory corruption attacks, such as Address Space Layout Randomization (ASLR) [10] due to the low entropy in the address space layout.

Trusted Execution Environments (TEEs, a.k.a. enclaves), an enabling technology for the nascent confidential computing paradigm [11], [12], are offered in CPUs as a foundational

primitive for security to keep code and data loaded inside protected, with respect to confidentiality and integrity, from Rich Execution Environments (REEs). TEEs are designed to provide a haven to execute software in security and fault isolation and serve as a trusted anchor to deploy security solutions that monitor the REE software, e.g., Control-Flow Integrity (CFI) enforcement [13]–[17]. As TEEs bring forward a new hope for trustworthy embedded and IoT systems, the semiconductor industry is integrating them into CPUs. For example, Arm offers the TrustZone TEEs for both Cortex-A and Cortex-M architectures [18]. Additionally, Arm Confidential Compute Architecture (CCA) [19] is supported on Cortex-A but not on Cortex-M.

The hardware and software layers of existing embedded TEEs nevertheless have been criticized for lack of transparency, full of vulnerabilities, and various restrictions, which means the existing TEEs [20] and TEE-based security solutions, such as for isolation [21]–[24] and CFI [25], [26] are untrustworthy, ineffective, or inefficient. Failure to make TEEs trustworthy will backfire instead of enhancing security because embedded TEEs usually have the highest privilege and a compromised TEE can completely sabotage the REE.

In this paper, we present our perspective on the challenges and limitations related to embedded and IoT TEEs. Utilizing Arm Cortex-M and Cortex-A TrustZone as examples, we draw comparisons between embedded TEEs and their microprocessor counterparts. Given the fundamental roles TEEs may play in securing embedded and IoT systems, it is thus imperative to increase the trustworthiness and deployability of embedded TEEs and TEE-based security solutions. To this end, we delve into three recently published projects from our research lab. These projects aim to tackle challenges presented in embedded TEEs and TEE-based solutions at various layers. Specifically, BYOTEE [27] strives to establish trustworthy embedded TEE *hardware*, laying a secure foundation for TEE software. RET2NS [28] investigates and eliminates exploitable confused deputy vulnerabilities in embedded TEE *software*, helping establish a secure software base for TEE-based security solutions. Lastly, SHERLOC [29] employs embedded TEEs to achieve system-oriented control-flow violation detection for IoT systems. We have open-sourced all three projects at CactiLab's GitHub webpage.[1]

II. BACKGROUND

In this section, we overview two embedded and IoT platforms: System-on-Chip (SoC) Field Programmable Gate Arrays (FPGA) and Arm Cortex-M. In addition, we compare mobile and embedded TEEs using Cortex-A and Cortex-M TrustZone as a case study.

A. Embedded and IoT Architectures: A Case Study of SoC FPGA and Arm Cortex-M

SoC FPGA. System-on-Chip implements the functionality of an entire system on a single silicon chip. Compared with system-on-printed-circuit-board, SoC is lower cost, enables more secure data transfers, and has higher speed and lower power consumption. However, traditional application-specific integrated circuit SoCs lack flexibility, making them suitable only for products with a limited lifetime. SoC FPGA is a type of flexible system-on-programmable-chip, where FPGAs can be reconfigured as desired. The market of SoC FPGA is expected to grow significantly with the increase in the global adoption of artificial intelligence and internet-of-things solutions.

A SoC FPGA comprises the following parts: (i) Processing System (PS), which is formed around hard processors, such as the Cortex-A processor on Xilinx Zynq-7000 SoC [30]. Operating systems and applications run on the PS; (ii) FPGA, which can implement any arbitrary system, including soft processors, e.g., Cortex-M or MicroBlaze, 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 [21], SRAM decays faster [31]. FPGA is configured with a bitstream, which is programmed in hardware design description languages, such as Verilog or VHDL; (iii) other integrated on-chip memory and high-speed communications interfaces.

Arm Cortex-M. Most Arm Cortex-M processors have thread and handler execution modes [32]. They also have privileged (kernel space) and unprivileged (userspace) levels, which are orthogonal to the execution mode. The current mode and privilege level are determined by the combination of the interrupt program status register (`IPSR`) and the `CONTROL` register. `IPSR` is part of the program status register (`xPSR`). `IPSR` indicates the exception number and handler mode if not 0. If `IPSR` is 0, the processor is in the thread mode, and the `nPRIV` bit of `CONTROL` determines whether the state is unprivileged or not.

To switch a processor's execution level from privileged to unprivileged, software can simply change `CONTROL[0]` to 1 using the `msr` (move-to-system-register) instruction. To switch from unprivileged to privileged, software makes a SuperVisor Call (SVC) with the `svc` instruction. When a higher priority interrupt or exception occurs, the processor automatically pushes eight registers, including `R0–R3`, `R12`, link register (`LR`), program counter (`PC`), and program status register (`xPSR`) to the current stack. Then, the processor generates a special exception return value named `EXC_RETURN` (`0xFFFFFF**`), stores it in `LR`, and executes the Interrupt Service Routine (ISR), e.g., SVC handler. When an ISR exits and `EXC_RETURN` is copied to the `PC`, the processor will automatically perform unstacking, which pops the eight registers off the stack. The hardware-assisted mechanism of stacking and unstacking makes it possible to develop C-language-only interrupt handlers.

[1] https://github.com/CactiLab/

B. Comparing Mobile and Embedded TEEs: A Case Study of Arm Cortex-A and Cortex-M TrustZone

Arm TrustZone is a hardware-assisted TEE that splits system-on-chip resources between two execution *states*, non-secure and secure. Software running in the secure state can access all resources, whereas software in the non-secure state can only access non-secure resources. First introduced with Cortex-A [18], TrustZone has been recently extended to Cortex-M [33]–[35], but streamlined and optimized for performance. In this section, we will discuss some differences between the Cortex-A and Cortex-M TrustZone architectures and their associated security features.

TrustZone State Switches. As shown in Figure 2, Cortex-A TrustZone uses a dedicated secure monitor mode (EL3) to handle the secure state transitions. The processor will be running at the secure state while in the monitor mode, and uses the `NS` bit in the Secure Configuration Register (`SCR`) to define the operation state to which the CPU will switch after the monitor mode. Therefore, any cross-state transitions will go through the single entry point – the privileged secure monitor mode – via the secure monitor call instruction (`smc`). Different from Cortex-A TrustZone, which indicates the security state in the secure configuration register, the division of states in Cortex-M TrustZone is based on memory regions. When running code in the secure memory, the processor is in the secure state. Otherwise, the processor is in the non-secure state. State switches in Cortex-M TrustZone can occur through function calls (`blxns`) and returns (`bxns`), resulting in an unlimited number of entries between secure and non-secure privilege levels.

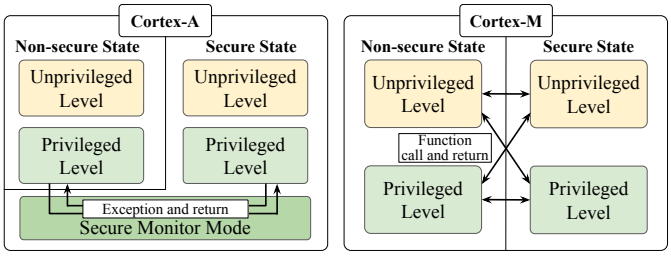

Fig. 2. State switches on Cortex-A TrustZone must go through the privileged secure monitor mode, whereas state switches on Cortex-M can occur between different secure and non-secure privilege levels.

Other Differences: Pointer Authentication Code (PAC) as an Example. In addition to TrustZone, there are variations in other security features between Cortex-A and Cortex-M. For instance, as shown in Table I and Table II, the key management mechanisms of the Pointer Authentication (PA) security extension for Cortex-A [36] and Cortex-M [37] with TrustZone are different. The newly introduced PA instructions can generate and verify a keyed tweakable pointer authentication code for a pointer or data with the QARMA block cipher. Cortex-M with TrustZone integrates four 128-bit PA key registers across two security states. In contrast, the Cortex-A platform has five key registers without segregation based on security states. These registers can only be modified using the privileged `msr` instruction. The secure state privileged code can modify both keys of the non-secure state in Cortex-M. Additionally, in Cortex-M the PAC is stored separately from the pointer it protects, whereas in Cortex-A the PAC is embedded within the most significant bits of the pointer itself as no existing system uses the full 64-bit address space.

Cortex-A PA Key Registers	Used at	Config. at
`APIAKey_EL1`	All exception levels	EL1 and higher
`APIBKey_EL1`		
`APDAKey_EL1`		
`APDBKey_EL1`		
`APGAKey_EL1`		

TABLE I
THE FIVE 128-BIT ARMV8-A PA KEYS AND THEIR USAGE AND CONFIGURATION SETTINGS.

Cortex-M PA Key Registers	Used at	Config. at
`PAC_KEY_U_NS`	U-NS	P-NS and P-S
`PAC_KEY_P_NS`	P-NS	P-NS and P-S
`PAC_KEY_U_S`	U-S	P-S
`PAC_KEY_P_S`	P-S	P-S

TABLE II
THE FOUR 128-BIT ARMV8-M PA KEY REGISTERS AND THEIR USAGE AND CONFIGURATION SETTINGS. U-NS: UNPRIVILEGED NON-SECURE, P-NS: PRIVILEGED NON-SECURE, U-S: UNPRIVILEGED SECURE, P-S: PRIVILEGED SECURE.

III. LIMITATIONS OF EMBEDDED AND IOT TEES

In this section, we share a perspective on three key sources that contribute to the limitations observed in existing embedded TEEs and TEE-based solutions.

A. Security Limitations

Existing TEEs offer limited security guarantees at the hardware layer.

- (S1) The hardware of all existing TEEs are proprietary and must be trusted blindly. It is hence impossible to verify the correctness of the hardware design or attest the hardware states at run-time.
- (S2) The hardware Trusted Computing Bases (TCBs) of existing TEEs are static and cannot be customized for different applications. In particular, TrustZone TEE has the highest privilege to control the REE and communicate with all peripherals, which violates the principle of least privilege by including unnecessary peripherals and exposing them to *malicious peripheral attacks* [38].
- (S3) Embedded TEEs, e.g., Cortex-A and Cortex-M TrustZone, only provide *one* enclave and cannot meet the needs of multiple enclaves for sophisticated applications, in which the trusted firmware, OS, and applications execute.
- (S4) Embedded TEEs do not encrypt memory, leaving them vulnerable to *cold boot attacks* [21].
- (S5) The TEE shares a processor core with the REE in a time-sliced fashion, making it vulnerable to *cache side-channel attacks* [22]–[24].

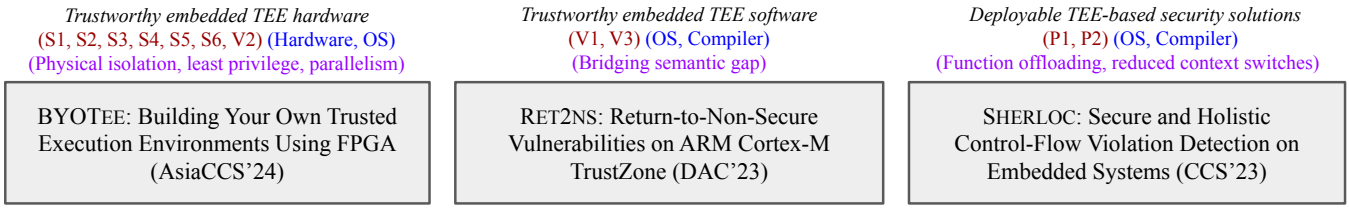

Fig. 3. Recently publications from CactiLab on embedded and IoT TEEs. The addressed issues, implementation layers, and guiding principles are shown in red, blue, and purple, respectively.

- (S6) Unlike SGX, embedded TEEs do not have native hardware support for remote software attestation.

B. Inherent Semantic Gap

The inherent semantic gap between the TEE and the REE, the vulnerabilities in the large TEE software TCBs, and the wide REE-TEE transition surface lead to confused deputy attacks with severe consequences.

- (V1) The TEE has very limited visibility into the REE's security mechanisms, introducing an inherent semantic gap. On Cortex-A, *confused deputy attacks* exploiting this semantic gap can allow an REE application to read and write any memory location in the kernel by tricking the TEE to perform the operations [39].
- (V2) The software TCBs in TEEs are large, creating big surfaces for *control-flow hijacking attacks* [40], [41]. For example, OP-TEE, a Cortex-A TEE OS, has 277K source lines of code (SLOC) [42], and TF-M [43], a Cortex-M TEE OS, has over 117K SLOC.
- (V3) Different from the Cortex-A TrustZone, which only has the secure monitor mode for REE-TEE context switches, embedded TEEs, e.g., Cortex-M TrustZone, support an unlimited number of TEE entrances through the `SG` instruction and exits through the `BXNS` and `BLXNS` instructions. We have found that the wide REE-TEE transition surface introduces new confused deputy vulnerabilities and also makes their exploitation easier. Therefore, it is challenging to implement a secure channel between a non-secure software component and a secure software component, as SeCReT [44] does for the Cortex-A TrustZone.

C. Performance Overhead

Existing TEE hardware and software introduce significant performance overhead, rendering TEE-based security solutions impractical for real-world deployment.

- (P1) Existing TEEs time-share a processor core with REEs, and the frequent context switches between them are very expensive [45], [46]. For example, the state-of-the-art TrustZone-based backward-edge only CFI solutions [25], [26] and CFA [47] introduce very high runtime overhead.
- (P2) The aforementioned large software TCBs in TEEs also lead to slow performance.

IV. Our Recent Attempts

Given the fundamental roles TEEs may play in securing embedded and IoT systems, it is thus imperative to take approaches to increase the trustworthiness and deployability of embedded TEEs and TEE-based security solutions. To this end, CactiLab conducted three projects in this direction recently as shown in Figure 3. In the first project, BYOTEE, we present an approach to build trustworthy embedded TEE *hardware*, attempting to lay a secure hardware foundation for the TEE software. In the second project, RET2NS, we address a particular type of software vulnerability in embedded TEE to help create trustworthy embedded TEE *software* and a secure software base for TEE-based security solutions. Building on top of the previous two projects, in the third project, SHERLOC, we develop a security solution with novel algorithms and architectures in the embedded TEE to secure the control-flow transfers in the embedded REE. In the remainder of this section, we delve into the research questions, challenges, and high-level concepts for each of these projects.

A. BYOTEE: Building Your Own Trusted Execution Environments Using FPGA

TEEs are a foundational primitive in confidential computing. Existing TEEs, however, have the aforementioned security limitations. This raises an important research question: ***How to design a novel TEE paradigm and infrastructure that overcomes the security limitations of existing TEEs?*** Specifically, the novel TEE should have the following security and functional properties: (i) unlike TrustZone, the paradigm should offer multiple enclaves at the same time and guarantee the secrecy and integrity of the Security-Sensitive Applications (SSA) running inside each; (ii) the hardware resources of each enclave should be physically isolated from the REE and other enclaves to mitigate side-channel attacks; (iii) the hardware TCB of each enclave should be customizable, allowing for a minimal hardware TCB. Resorting to the formal verification of the customized hardware [48], [49], the size of hardware TCB can be further reduced; (iv) trusted I/O paths between an enclave and its peripherals; (v) enclaves should be able to securely communicate with each other; (vi) the paradigm should provide mechanisms to attest the integrity of an enclave's hardware and software stacks; and (vii) the infrastructure should be easy to use, especially for software developers who do not have hardware programming experience.

1) Related work by others and their limitations: Many software- or hardware-based solutions have been proposed to address one or more limitations of existing TEEs. Among them, TEEOD [50] is the most related. Compared to TEEOD, BYOTEE offers additional security features, such as trust bootstrap and software- and hardware-based attestation. 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 [51] provides each virtual machine with a virtualized TEE by running a monitor within the secure world. SANCTUARY [52] utilizes the memory access controller to provide multi-domain isolation. TrustICE [53] creates multiple computing environments in the normal domain and runs a monitor in the secure world. uTango [54] uses the secure attribution unit of Cortex-M to create multiple secure execution environments. On RISC-V, KeyStone [55] 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 [56] enables the exclusive assignment of system resources to single enclaves. Composite Enclaves [57] builds on top of KeyStone [55] and extends the TEE to several hardware components. HECTOR-V [58] 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 [59] presents a hypervisor-based trusted path architecture for SGX. SGX-FPGA [60] builds a secure path between the CPU and FPGA. To eliminate side-channel attacks, Sanctum [61] combines invasive hardware modifications with a trusted software monitor on RISC-V.

Building TEEs with Other Hardware. Graviton [62] and StrongBox [63] offload security-sensitive code and data to a GPU. Ambassy [64] and MeetGo [65] use FPGA to construct TEEs, but they do not include softcore CPUs. Dedicated processor solutions, such as Google Titan [66], Samsung eSE [67], and Apple SEP [68], use external connections between the REE and TEE, making them vulnerable to physical probing attacks [58]. BYOTEE is also inspired by other isolated execution environment solutions, including Flickr [69], TrustVisor [70], and Haven [71].

2) Our Approach: We presented 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 or 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.

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 the 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).

In BYOTEE, 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, is 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 [72], [73]. However, this model is not realistic as the firmware and SSAs may have bugs that can be exploited by REE inputs [20], [39]. In EnhancedModel, we assume that the firmware and SSAs are buggy and can be compromised. Therefore, the 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 Cortex-A CCA model where trusted hardware components of the CPU perform remote attestation.

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 or Intel Quartus Prime, 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.

3) Results: We implemented the BYOTEE system and toolchain for the Xilinx SoC FPGA. We open-sourced the

system and toolchain[2]. We have also demonstrated BYOTee's usage, security, effectiveness, and performance with the Embench-IoT benchmark and four SSAs, i.e., a computational application, a peripheral-interacting application, a peripheral- and hardcore system-interacting application, and a distributed application. on the low-end MicroBlaze softcore CPU and Zynq-7000 system.

B. RET2NS*: Return-to-Non-Secure Vulnerabilities on Arm Cortex-M TrustZone*

Confused deputy vulnerabilities exist when a more privileged program is tricked by a less privileged but malicious program into misusing its authority. The inherent semantic gap between a privileged program and a less privileged domain, e.g., hypervisor's view of a virtual machine [74] and TrustZone TEE's view of the REE memory [39], inevitably leads to confused deputy vulnerabilities. To achieve high performance, the embedded TEEs have a wide REE-TEE transition surface, which introduces new types of confused deputy vulnerabilities. This raises a key question: ***Will these confused deputy vulnerabilities on embedded TEEs lead to serious attacks? If yes, how to mitigate such attacks?*** Specifically, we explore if Cortex-M TrustZone's support of an unlimited number of TEE entrances through the `SG` instruction and TEE exits through the `BXNS` and `BLXNS` instructions can be easily exploited for privilege escalation. Additionally, we design (i) a detection system that performs data-flow analysis to track data passed from the REE application and annotates the `BXNS` and `BLXNS` instructions that use the data as the branch destination; and (ii) an address sanitization approach that instruments checks before the unsafe `BXNS` and `BLXNS` instructions.

1) Related work by others: Ret2usr [75], ret2dir [76], and boomerang [39] are disclosed confused deputy attacks on microprocessors with MMUs, e.g., x86 or Cortex-A, and on modern operating systems, e.g., Linux. The premise of the ret2usr attack [75] lies in leveraging a legitimate but compromised kernel function, which is manipulated to return not to its original caller in the kernel space, but instead to an attacker-specified location in the user space. This redirection allows malicious user-level code to be executed with the elevated privileges of kernel mode, effectively bypassing the protections inherent in separating user and kernel spaces.

Ret2dir attacks are a form of confused deputy attacks that exploit a feature of modern operating systems: direct kernel mappings of physical memory [76]. These attacks maneuver around protections against ret2usr attacks by redirecting kernel operations not to user space, but to a virtual memory region within the kernel space itself. This region directly maps all or part of the physical memory, allowing for unauthorized access or modifications. Unlike ret2usr, ret2dir attacks bypass traditional defenses, as the malicious code remains within the kernel space.

Boomerang attacks represent a type of confused deputy attacks that target systems implementing Cortex-A TrustZone [39]. These attacks manipulate the TrustZone's secure state applications to gain unauthorized access to restricted memory regions. This is achieved by making a non-secure application request the secure state application to execute an operation that shouldn't be permissible from the non-secure state. In effect, the secure application acts as a "boomerang," returning with sensitive data or causing alterations that the non-secure application should not access or perform.

To mitigate ret2usr attacks, kGuard [75] instruments runtime control-flow checks to verify the indirect branch target is always in kernel space and enforces lightweight address space segregation. To patch the ret2dir vulnerability, XPFO [76] uses an exclusive ownership scheme for the Linux kernel that prevents the implicit sharing of physical memory. To thwart boomerang attacks, a cooperative approach [39] requires that all of the non-secure memory accesses from the secure state need to query a non-secure callback function to verify the access permission of the referenced memory region.

2) Our Approach: In RET2NS, we report a new class of confused deputy attacks, namely return-to-non-secure attacks, that exploit the fast state switch mechanism of the Cortex-M TrustZone. More dangerous than boomerang, RET2NS can lead to arbitrary code execution with escalated privilege in the non-secure state. Ret2ns is a new type of return-to-user (ret2usr) attacks [75], [76] that redirects compromised secure state pointers to code residing in non-secure state userspace. The wide state transition surface on Cortex-M also makes the exploitation of ret2ns vulnerabilities easier than exploiting ret2usr on x86 or boomerang on Cortex-A. Ret2ns attacks affect all Cortex-M processors with TrustZone, including M23 [77], M33 [78], M35P [79], M55 [80], and M85 [81]. We also argue RET2NS vulnerabilities are likely to exist in any TEE implementations that allow direct control-flow transfers from secure state to non-secure userspace programs but keep executing at the privileged level.

Based on the non-secure execution mode that an attack originates from, we break RET2NS attacks into two categories: handler-mode-originated and thread-mode-originated attacks. Attacks in the former category are more likely to happen in RTOSes, whereas attacks in the latter category are likely to occur in security-enhanced bare-metal systems that support privilege separation. Attacks in either category can be further attributed to an indirect branch case using `bxns` or an indirect call case using `blxns`, resulting in four variants of RET2NS attacks.

In the handler-mode-originated attacks, a userspace program under the attacker's control makes a supervisor call, so the processor enters the handler mode and `IPSR` is updated to 11 (the interrupt number of SVC). The SVC handler in turn calls a non-secure callable (NSC) function, and the processor switches to the secure state. Because `IPSR` is shared between the secure and non-secure state, the secure state program keeps executing in the handler mode with privilege. In a legitimate control path, when the secure state program exits back to the non-secure state using `bxns`, the control returns to the SVC handler. However, if the `bxns` instruction uses a corrupted code pointer as the destination, the processor can return to

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

any location, e.g., userspace program, in the non-secure state and keep executing it in the handler mode with privilege. Another attack path exists when a secure state program makes an indirect call (`blxns`) with a corrupted code pointer. In this case, `IPSR` has the value of 1. In the thread-mode-originated attacks, the attacker-controlled unprivileged program uses an SVC call to escalate the non-secure privilege level with the `CONTROL_NS.nPRIV` bit cleared, after which a privileged program in the thread mode executes. The privileged program in turn calls an NSC function in the secure state. The NSC function will call the secure state program, which eventually returns the control to the non-secure state (using `bxns`) or calls a non-secure callback function (using `blxns`). When a memory corruption vulnerability in the secure state program leads to a corrupted code pointer, the control flow will transfer to an attacker-controlled program in the non-secure state. Since the non-secure state has `CONTROL_NS.nPRIV` cleared, the attacker-controlled program will keep executing in the privileged thread mode.

The key to preventing RET2NS attacks is to disallow the execution of non-secure userspace programs at the privileged level. On the planned Cortex-M55 and M85 microcontrollers, this can be achieved with negligible overhead by properly setting up the MPU regions with PXN. However, there are two limitations of the PXN approach: (1) Cortex-M23, M33, and M35P microcontrollers that hold a large market share do not have the PXN feature; (2) only a small number of MPU regions, e.g., 8 or 16, are supported, thus it is not fine-grained enough for complex RTOSes. To address these issues, we present two mechanisms, namely (i) MPU-assisted address sanitizer and (ii) address masking, which can effectively mitigate RET2NS attacks for all Cortex-M microcontrollers with TrustZone.

3) Results: We experimentally confirmed the feasibility of four variants of RET2NS attacks on two Cortex-M hardware systems. To defend against RET2NS attacks, we designed two address sanitizing mechanisms that have negligible performance overhead. We open-sourced our project[3], which includes vulnerable code examples, proof-of-concept exploits, and defense instrumentation.

C. SHERLOC: Secure and Holistic Control-Flow Violation Detection on Embedded Systems

Control-Flow Integrity (CFI) is a basic security property that can prevent control-flow hijacking by dictating that indirect control transfers, e.g., indirect call/branch and return, must follow a predetermined Control-Flow Graph (CFG). The CFI property can be enforced by inline instrumentation or trace-based control-flow violation detection. Existing CFI enforcement approaches in both categories for embedded and IoT systems are either incomplete (e.g., only consider unprivileged code or backward-edges), insecure (e.g., the trace can be overwritten by adversaries), or inefficient (e.g., overhead that makes them impractical for real-world deployments). This raises an important research question: ***How to design a secure and efficient control-flow integrity enforcement algorithm and mechanism on forward- and backward-edges of both privileged and unprivileged code for embedded devices?*** Specifically, the new algorithm and mechanism should have the following design goals: (i) it must enforce CFI for both unprivileged and privileged code; and (ii) it must enforce both forward and backward edge CFI.

1) Related work by others and their limitations: Inline instrumentation inserts a label at a destination and a check before a source [13]. It also inserts shadow stack [13], [82], [83] or return address [40] checks in the prologue and epilogue of a function. It is, however, impractical on embedded devices: (i) instrumentation increases the binary size and changes the memory layout, which is not an issue for memory-rich devices but infeasible for embedded devices with limited memory. To preserve the memory layout, CaRE [25] replaces function calls and indirect branches with dispatch instructions at the cost of a 513% performance overhead; (ii) shadow stacks need to be protected. RECFISH [84], which only protects unprivileged code, maintains shadow stacks at the privileged level and introduces a 30% overhead. TzmCFI [26] maintains shadow stacks in the TrustZone secure world and introduces an 84% overhead. Silhouette [41] and Kage [85] use unprivileged store instructions to achieve a low overhead of 3.4% and 5.2%, respectively; nonetheless, Silhouette only works for bare-metal applications, and Kage only supports a small number of tasks.

Trace-based Control-Flow Violation Detection (CFVD) does not instrument the target software or change its memory layout, and it has the potential for superior performance. Existing approaches, e.g., CFIMon [86], FlowGuard [87], GRIFFIN [88], and PT-CFI [89], nevertheless have these limitations: (i) they only monitor unprivileged code because the traces need to be stored at secure memory regions, which leads to three issues: (1) they do not directly apply to embedded systems, which mostly execute at the privileged level; (2) they require kernel changes and bloat the size of TCB by including the kernel [86]–[91]; (3) to reduce the number of context switches between userspace and kernel, their violation analysis is only triggered by an incomplete list of system calls [86]–[89], [91]. Intel's Last Branch Record (LBR)-based approaches, e.g., CFIGuard [90] and PathArmor [91], store traces in several registers, but they are vulnerable to flushing attacks [92], [93]; (ii) they rely on advanced debugging features, e.g., trace filtering and labeling of Intel Branch Trace Store (BTS) and Processor Trace (PT) [94], to filter legitimate asynchronous noises [87]–[89], e.g., interrupts and process/task context switches. But these features are not available on embedded debugging units.

2) Our Approach: In SHERLOC, we formalized the definitions of the existing application-oriented CFVD mechanism and the proposed system-oriented CFVD mechanism. Current CFVD mechanisms for desktop systems with memory virtualization only monitor a specific userland application. We model the interprocedural CFG of such an application $\mathcal{A}$ as $G_{\mathcal{A}} = (V_{\mathcal{A}}, E_{\mathcal{A}})$, where $V_{\mathcal{A}}$ is the set of basic blocks and $E_{\mathcal{A}}$ is the set of control-flow transfers defined by the

[3]https://github.com/CactiLab/ret2ns-Cortex-M-TrustZone

application. Existing approaches configure hardware tracing units, such as Intel Processor Trace [94], to record only certain control-flow transfers, such as indirect calls/jumps and returns, within a single application $\mathcal{A}$ and exclude synchronous and asynchronous control-flow transfers of other applications or the kernel. We model each trace record as a 2-tuple $r = \langle s, d \rangle$ where s is the virtual source address and d is the virtual destination address. The ACFVD solution verifies each indirect control-flow transfer must match an edge in the $G_{\mathcal{A}}$:

Application-oriented CFVD (ACFVD). Given the trace $R_{\mathcal{A}} = (r_0, r_1, \ldots, r_n)$ of an application $\mathcal{A}$, ACFVD verifies that $r_i \in E_{\mathcal{A}}$, $\forall i \in \{0, 1, \ldots, n\}$.

ACFVD approaches require the modification of the kernel to capture and analyze traces, which makes it difficult to extend them to monitor privileged code or the kernel itself. Furthermore, to achieve better performance, many solutions perform incomplete monitoring and mandate analyzing only the traces that lead to specific system calls, such as *execve*, *mmap*, and *mprotect*.

We model a microcontroller-based software system $\mathcal{S}$ including a kernel $\mathcal{K}$ and tasks $\mathcal{T}$ as $(G_{\mathcal{S}}, I_{\mathcal{K}}, Y_{\mathcal{T}})$, where $G_{\mathcal{S}}$ is its interprocedural CFG, $I_{\mathcal{K}}$ is the set of *asynchronous* kernel interrupt service routine addresses (e.g., timer handler), and $Y_{\mathcal{T}}$ is the set of task entry or re-entry addresses. In particular, $G_{\mathcal{S}}$ is modeled as $(V_{\mathcal{S}}, E_{\mathcal{S}})$, where $V_{\mathcal{S}}$ is the set of basic blocks of tasks and the kernel, and $E_{\mathcal{S}}$ is the set of control-flow transfers defined by the tasks and the kernel. Please note that $E_{\mathcal{S}}$ also models *synchronous* exceptions that involve control-flow transfers across privilege levels, such as system calls. At system boot, $Y_{\mathcal{T}} = Y_{\mathcal{TE}} \bigcup Y_{\mathcal{TR}}$ is initially composed of statically retrieved entry addresses of all tasks $Y_{\mathcal{TE}}$, which may be dynamically replaced by re-entry addresses $Y_{\mathcal{TR}}$ when context switches occur. The SCFVD solution verifies each indirect control-flow transfer must match an edge in the $G_{\mathcal{S}}$ or the destination of each asynchronous control-flow transfer must match an address in the $I_{\mathcal{K}}$ or $Y_{\mathcal{T}}$:

System-oriented CFVD (SCFVD). Given the trace $R_{\mathcal{S}} = (r_0, r_1, \ldots, r_n)$ of a system $\mathcal{S}$ including a kernel $\mathcal{K}$ and tasks $\mathcal{T}$, SCFVD verifies that $r_i \in E_{\mathcal{S}} \bigvee r_i.d \in I_{\mathcal{K}} \bigcup Y_{\mathcal{T}}$, $\forall i \in \{0, 1, \ldots, n\}$.

We presented Secure and Holistic Control-Flow Violation Detection (SHERLOC) for microcontroller-based embedded systems. To ensure *security*, SHERLOC configures the hardware tracing unit, stores the trace records, and executes the CFVD algorithm in a trusted execution environment (TEE), so even the non-secure state privileged program cannot bypass the monitoring or tamper the traces. To achieve *holistic* monitoring, SHERLOC provides a mechanism that not only monitors the forward and backward edges of unprivileged and privileged programs but also the control-flow transfers among unprivileged and privileged components. Specifically, SHERLOC addresses the challenges of identifying legitimate asynchronous interrupts and context switches among applications at run-time with an interrupt- and scheduling-aware violation detection algorithm. To improve performance, SHERLOC can also enforce more practical policies, such as analyzing traces only when certain operations, such as changing system registers, are triggered.

SHERLOC comprises offline analysis and runtime configuration and enforcement modules. To validate an indirect or asynchronous control-flow transfer, SHERLOC requires an over-approximated CFG, a list of asynchronous ISR addresses, and entry and re-entry addresses of all RTOS tasks. An over-approximated CFG may result in a higher false negative rate, but it guarantees no false positives. The offline analysis module only produces the set of legitimate indirect forward edges from the generated CFG and entry addresses of all RTOS tasks. SHERLOC runtime modules can retrieve asynchronous ISR addresses directly from the non-secure state vector table (VT) on SRAM or flash by reading the vector table offset register (`VTOR_NS`). The indirect backward edges, i.e., returns, and the re-entry addresses of tasks are maintained dynamically by the SHERLOC runtime enforcement module.

Besides holistic runtime enforcement, SHERLOC also supports event-triggered runtime enforcement, providing a trade-off between security and performance. Similar to the existing implementations of application-oriented CFVD [86]–[88], [90], [91], the enforcement can be triggered by some sensitive events, such as calling a certain kernel API (similar to system calls on desktop) and modifying a particular system register (e.g., changing memory permissions).

3) Results: We implemented SHERLOC for the ARMv8-M architecture and evaluate its performance using embedded benchmark programs, bare-metal systems, and a real-time operating system. Our evaluation results demonstrated that SHERLOC is secure, effective, and efficient. We open-sourced SHERLOC[4].

V. Recommendation and Future Directions

Explore the pros and cons of new trusted execution environment features: The hardware features of each TEE design differ, especially with embedded TEEs exhibiting streamlining and distinctions from their microprocessor counterparts. This distinction encompasses various aspects, extending from the microarchitectural layer up to the instruction set architecture. It influences not only the underlying hardware design but also the operational characteristics and capabilities of the TEEs. Understanding these differences is crucial for comprehending the unique challenges and opportunities associated with embedded TEEs as opposed to their microprocessor counterparts.

Explore the integration of TEE with additional security features: Embedded platforms bring forth a multitude of distinct hardware and security features that set them apart from their microprocessor counterparts. Investigating the synergies between TEEs and these additional security features becomes imperative for comprehensively understanding the security landscape in embedded systems. This exploration will shed

[4]https://github.com/CactiLab/Sherloc-Cortex-M-CFVD

light on how the integration of TEEs with other security measures can enhance overall system resilience and protection against diverse threats.

VI. Conclusion

TEEs have been offered in CPUs of embedded and IoT platforms as a foundational primitive for security. The hardware and software layers of existing TEEs nevertheless have been criticized for lack of transparency, full of vulnerabilities, and various restrictions, which means the existing TEEs and TEE-based security solutions are untrustworthy, ineffective, or inefficient. In this paper, we share our perspective on the challenges and limitations related to embedded TEEs. Additionally, we delve into three recently published projects from CactiLab, which aim to tackle challenges presented in embedded TEEs and TEE-based solutions at various layers.

Acknowledgment

This material is based upon work supported in part by National Science Foundation (NSF) grants (2237238 and 2329704), 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] P. Sparks, "The route to a trillion devices," https://community.arm.com/iot/b/blog/posts/whitepaper-the-route-to-a-trillion-devices, online; accessed 20 Apr 2021.

[2] FDA, "Certain Medtronic MiniMed Insulin Pumps Have Potential Cybersecurity Risks: FDA Safety Communication," https://www.fda.gov/medical-devices/safety-communications/certain-medtronic-minimed-insulin-pumps-have-potential-cybersecurity-risks-fda-safety-communication, 2019.

[3] I. Stellios, P. Kotzanikolaou, M. Psarakis, C. Alcaraz, and J. Lopez, "A survey of iot-enabled cyberattacks: Assessing attack paths to critical infrastructures and services," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3453–3495, 2018.

[4] Wikipedia, "Colonial pipeline cyber attack," https://en.wikipedia.org/wiki/Colonial_Pipeline_cyber_attack.

[5] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[6] A. van de Ven and I. Molnar, "Exec shield," http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf, 2004.

[7] N. R. Weidler, D. Brown, S. A. Mitchel, J. Anderson, J. R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, and R. Gerdes, "Return-oriented programming on a cortex-m processor," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 823–832.

[8] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Return-oriented programming without returns on arm," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2010.

[9] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 227–242.

[10] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, 2004, pp. 298–307.

[11] F. Y. Rashid, "The rise of confidential computing: Big tech companies are adopting a new security model to protect data while it's in use-[news]," *IEEE Spectrum*, vol. 57, no. 6, pp. 8–9, 2020.

[12] D. P. Mulligan, G. Petri, N. Spinale, G. Stockwell, and H. J. Vincent, "Confidential computing—a brave new world," in *2021 international symposium on secure and private execution environment design (SEED)*. IEEE, 2021, pp. 132–138.

[13] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[14] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *USENIX Security Symposium, pages=337–352, year=2013*.

[15] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 575–589.

[16] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 577–587.

[17] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 559–573.

[18] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[19] 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)*, 2022, pp. 465–484.

[20] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1416–1432.

[21] 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*, pp. 91–98, 2009.

[22] 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*, vol. 2016, p. 980, 2016.

[23] 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 *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2018, pp. 441–452.

[24] 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.

[25] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "Cfi care: Hardware-supported call and return enforcement for commercial microcontrollers," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 259–284.

[26] T. Kawada, S. Honda, Y. Matsubara, and H. Takada, "Tzmcfi: Rtos-aware control-flow integrity using trustzone for armv8-m," *International Journal of Parallel Programming*, pp. 1–21, 2020.

[27] M. Armanuzzaman, A.-R. Sadeghi, and Z. Zhao, "Building Your Own Trusted Execution Environments Using FPGA," in *ACM ASIA Conference on Computer and Communications Security*, 2024.

[28] Z. Ma, X. Tan, L. Ziarek, N. Zhang, H. Hu, and Z. Zhao, "Return-to-non-secure vulnerabilities on arm cortex-m trustzone: Attack and defense," in *ACM/IEEE Design Automation Conference*, 2023.

[29] X. Tan and Z. Zhao, "Sherloc: Secure and holistic control-flow violation detection on embedded systems," in *ACM Conference on Computer and Communications Security*, 2023.

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

[31] 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.

[32] J. Yiu, *Definitive Guide to Arm Cortex-M23 and Cortex-M33 Processors*. Newnes, 2020.

[33] ARM, "Armv8-m architecture reference manual," https://developer.arm.com/documentation/ddi0553/bm/, online; accessed 15 Dec 2020.

[34] Arm, "Armv8-M Architecture Technical Overview," https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/0

1-2142-00-00-00-00-66-90/Whitepaper-_2D00_-Armv8_2D00_M-Architecture-Technical-Overview.pdf.

[35] ARM, "Trustzone technology for the armv8-m architecture version 2.1," https://developer.arm.com/documentation/100690/latest/, online; accessed 15 Dec 2020.

[36] Q. Technologies, "Pointer Authentication on ARMv8.3," https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[37] Arm, "Armv8.1-M Pointer Authentication and Branch Target Identification Extension," https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension.

[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] 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.

[40] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μ RAI: Securing Embedded Systems with Return Address Integrity," in *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[41] 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, pp. 1219–1236.

[42] Linaro, "OP-TEE: Open Portable Trusted Execution Environment," https://www.op-tee.org/, online; accessed 19 April 2021.

[43] ——, "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.

[44] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[45] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity." in *NDSS*, 2016.

[46] P. Zhang, H. Cho, Z. Zhao, A. Doupé, and G.-J. Ahn, "icore: continuous and proactive extrospection on multi-core iot devices," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 851–860.

[47] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: control-flow attestation for embedded systems software," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 743–754.

[48] 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*. IEEE, 1997, pp. 32–36.

[49] 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)*, vol. 19, no. 4, pp. 586–616, 1997.

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

[51] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing ARM trustzone," in *USENIX Security Symposium*, 2017, pp. 541–556.

[52] 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.

[53] 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.

[54] D. Oliveira, T. Gomes, and S. Pinto, "utango: an open-source tee for the internet of things," *arXiv preprint arXiv:2102.03625*, 2021.

[55] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387532

[56] 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.

[57] M. Schneider, A. Dhar, I. Puddu, K. Kostiainen, and S. Capkun, "Pie: A dynamic tcb for remote systems with a platform isolation environment," *arXiv preprint arXiv:2010.10416*, 2020.

[58] 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.

[59] 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.

[60] 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.

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

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

[63] 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.

[64] 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.

[65] H. Oh, K. Nam, S. Jeon, Y. Cho, and Y. Paek, "Meetgo: A trusted execution environment for remote applications on fpga," *IEEE Access*, 2021.

[66] 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*, vol. 194, 2018.

[67] Samsung, "eSE Safeguard against digital attacks." https://www.samsung.com/semiconductor/security/ese/, 2020.

[68] A. Inc, "Security enclave processor for a system on a chip. US8832465B2." https://patents.google.com/patent/US8832465, 2020.

[69] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, 2008, pp. 315–328.

[70] 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.

[71] 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.

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

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

[74] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 605–620.

[75] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight Kernel Protection against Return-to-User Attacks," in *USENIX Security Symposium*, 2012, pp. 459–474.

[76] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *USENIX Security Symposium*, 2014, pp. 957–972.

[77] Arm, "Arm Cortex-M23 Processor Technical Reference Manual," https://developer.arm.com/documentation/ddi0550/c.

[78] ——, "Arm Cortex-M33 Processor Technical Reference Manual Revision r1p0," https://developer.arm.com/documentation/100230/latest.

[79] ——, "Cortex-M35P," https://developer.arm.com/ip-products/processors/cortex-m/cortex-m35p.

[80] ——, "Arm Cortex-M55 Processor Technical Reference Manual Revision r0p2," https://developer.arm.com/documentation/101051/0002/.

[81] ——, "Arm Cortex-M55 Processor Technical Reference Manual Revision r0p2," https://developer.arm.com/documentation/101924/0002/?lang=en.
[82] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015, pp. 555–566.
[83] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 985–999.
[84] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-flow integrity for real-time embedded systems," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
[85] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell, "Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage," in *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022.
[86] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 2012, pp. 1–12.
[87] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient cfi enforcement with intel processor trace," in *2017 IEEE International Symposium on High performance computer architecture (HPCA)*. IEEE, 2017, pp. 529–540.
[88] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 585–598, 2017.
[89] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent backward-edge CFVD using intel processor trace," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 173–184.
[90] P. Yuan, Q. Zeng, and X. Ding, "Hardware-assisted fine-grained code-reuse attack detection," in *International Symposium on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 66–85.
[91] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 927–940.
[92] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *20th USENIX Security Symposium (USENIX Security 11)*, 2011.
[93] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz, "Evaluating the effectiveness of current anti-ROP defenses," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 88–108.
[94] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf.