IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 9 November 2022, accepted 6 December 2022, date of publication 19 December 2022,
date of current version 23 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3230791

RESEARCH ARTICLE

Efficient CFI Enforcement for Embedded Systems Using ARM TrustZone-M

GISU YEO[1], YERYEONG KIM[1], SUHYEON SONG[2], AND DONGHYUN KWON[2]
[1]Department of Electrical and Computer Engineering, Pusan National University, Geumjeong-Gu, Busan 46241, Republic of Korea
[2]School of Computer Science and Engineering, Pusan National University, Geumjeong-Gu, Busan 46241, Republic of Korea

Corresponding author: Donghyun Kwon (kwondh@pusan.ac.kr)

This work was supported in part by the Institute for Information and Communications Technology Planning and Evaluation (IITP) Grant Funded by the Korea Government, Ministry of Science and ICT (MSIT) (Convergence Security Core Talent Training Business, Pusan National University), under Grant 2022-0-01201; and in part by the MSIT, South Korea, under the Information Technology Research Center (ITRC) Support Program Supervised by the IITP under Grant IITP-2022-2020-0-01797.

ABSTRACT Embedded systems are deployed in many fields, from industrial applications to personal products. However, there are growing concerns regarding the security of these embedded systems as the number of attacks targeting them has increased. Control flow integrity (CFI) is a well-known security solution against these attacks. However, according to our analysis, existing CFI methods cannot be widely used in embedded systems one or more of the following reasons. (1) They require special hardware features that are not available in embedded systems, (2) they require that the developer recompile the source code with their compiler toolchain and (3) they incur considerable performance overhead to ensure CFI at runtime. In this paper, we propose `CEST`, a new scheme to ensure CFI on embedded systems using ARM TrustZone-M, a security extension for embedded ARM processors. For better compatibility, we designed `CEST` to be binary compatible. The evaluation results show that `CEST` can effectively enforce CFI compared to the existing studies using SVC.

INDEX TERMS ARM TrustZone-M, control flow integrity, embedded system security, binary patch.

I. INTRODUCTION

In the era of the Internet of things (IoT), embedded systems have been used in various fields such as smart homes [1], [2], [3] and smart factories [4], [5], [6], [7]. The software on these systems is typically developed with C/C++ because these languages offer high performance and direct control over the hardware resources. However, much software, especially written in C/C++, is known for including many software vulnerabilities such as buffer overflows [8], [9]. Thus, attackers can launch software attacks on the system by exploiting such vulnerabilities. Among software attacks, *control hijacking attack* (CHA) is notorious for manipulating the target system's control flow and executing the attacker's payload or code gadgets.

Numerous studies have been conducted to deal with CHA and they can be categorized into two approaches; (1) *code diversification* approach and (2) *control flow integrity* (CFI) approach. In code diversification approach [10], [11], [12], [13], [14], they randomize the layout of the address space or shuffle the order of code regions to hide the exact memory address of the payload or code gadgets. However, as embedded systems usually have small physical memory and do not support virtual memory, this approach cannot diversify the code memory layout sufficiently to prevent CHA in embedded systems. On the other hand, in CFI approach [15], they instrument the program to insert checking instructions before control transfer instructions at compile time. By doing this, at runtime, they check the legitimacy of the target address of every control transfer instruction (e.g., *return*, *indirect jumps*), and they can detect CHA. Consequently, in contrast to the code diversification approach, CFI can thwart CHA in a deterministic way regardless of the memory size.

Indeed, many researchers [8], [15], [16], [17], [18], [19], [20], [21], [22] have defended CHA through CFI-based solutions. However, we found the following challenges exist when it comes to designing the CFI solution for embedded systems. First, many security hardware features in

The associate editor coordinating the review of this manuscript and approving it for publication was Alessandro Floris.

TABLE 1. Comparison of defensive mechanisms against code reuse attack. The STRT is a special ARM instruction that supports an unprivileged store. TZ-M refers to TrustZone-M technology for Arm Cortex-M processors and SVC refers to Supervisor Call.

	Silhouette [8]	uRAI [16]	Kage [17]	TZmCFI [18]	RECFISH [19]	CFI CaRE [20]	CEST (Ours)
Binary compatiable	X	X	X	X	O	O	O
H/W Feature	STRT	None	STRT	TZ-M	SVC	TZ-M + SVC	TZ-M
Overhead	Low	Low	Low	Low	High	High	Low
Security guarantee	Both	Backward Only	Both	Backward Only	Both	Both	Both

mobile and server systems are lacking in embedded systems. For example, ARM architecture recently announced a new instruction set extension, called branch target identification (BTI) [23], to enforce CFI more efficiently. However, BTI is available only in high-end ARM processors (i.e., Cortex-A series), not low-end embedded ARM processors (i.e., Cortex-M series). Therefore, the existing CFI solutions using those security hardware features cannot be applied to the embedded system as it is, and the CFI solution for the embedded system should be carefully designed to utilize limited hardware features. Second, in embedded systems, accessing the source code is challenging and typically unavailable [24]. Thus, for better compatibility, the CFI solution must be implemented at the binary level or assembly level. Last, many embedded systems usually have a real-time constraint that the system must guarantee the response within a limited time. So, the execution time overhead caused by the CFI solution should be minimized. Otherwise, the embedded system cannot fulfill the real-time constraint.

However, according to our preliminary survey, no solution considered all aforementioned challenges (more details for existing studies in section II). So, in this paper, we propose CEST, a new technique for ensuring CFI for embedded systems. Specifically, to deal with those challenges, we designed CEST as follows. First, CEST utilizes ARM TrustZone-M, a unique hardware extension for embedded systems. ARM TrustZone-M provides a trusted execution environment (TEE), so CEST can protect its CFI enforcement routine (named *edge regulator*) by running *edge regulator* in TEE. More details for CEST are described in Section V. Secondly, we propose a code instrumentation technique based on the control transfer instruction type to make CEST applicable to the program binary. The details for our instrumentation technique are explained in Section V-C. Finally, we designed a *control deliverer* to reduce the performance overhead that occurs whenever CEST performs every CFI verification operation. In particular, in previous studies, considerable performance overhead occurs due to OS kernel intervention for every CFI verification operation. However, the *control deliverer* does not need the OS kernel intervention, enabling efficient CFI enforcement of CEST (see Section V-D). Our experimental results confirm the efficiency of CEST. When performing CFI protection for a program, CEST incurs an additional performance overhead of 159.30%, whereas the existing work incurs a performance overhead of 269.00%.

In summary,

- Implement the CFI framework for low-end embedded ARM processor efficiently.
- Our approach is designed to be implemented without adding new hardware and modifying the compiler considering the characteristics of embedded systems that can be difficult to access hardware or source code.
- Ensure efficient operation compared to the existing work when entering the control flow verification routine. Our design does not occur unnecessary context switching.

This paper consists of 8 sections including an introduction section. Section II explains the research gap and comparison with our research and related existing research. Section III introduces the backgrounds, and the following section IV explains the threat model and assumptions. Section V presents our design goals (section V-A) and explains our implementation divided into three parts. Section VI analyzes the efficiency of our research based on run-time overhead and binary size. Section VII discusses the limitation of our research and future research direction. Section VIII is the last section of this paper that contains the conclusion of this research.

II. RELATED WORK

Over the past decades, CFI has been considered an effective approach to defend CHA. According to the type of control transfer instructions, there are many schemes, such as *shadow stack* [15] and *branch regulation* [25]. The shadow stack places a separate stack for storing the return address in the memory space and is a method to ensure the integrity of the backward edge using the return located in the shadow stack. In branch regulation, the integrity of the forward and backward edge is guaranteed by regulating possible operations according to the type of each branch.

To implement these CFI schemes, many researcher have proposed CFI solutions for embedded systems. These solutions can be categorized into *code instrumentation* and *hardware-based* approaches. CEST belongs to the former one since CEST also patch the binary program to enforce CFI. Table 1 summarizes those studies in the code instrumentation approach and compares them with CEST. Silhouette [8] and Kage [17] design CFI solutions for forward and backward edges. Especially, by utilizing special hardware features, i.e., unprivileged load and store instructions, they can efficiently implement CFI solutions for embedded solutions. μRAI [16] proposes an efficient CFI solution for backward edges by reserving one general-purpose register to encode the current control flow information. TZmCFI [18] enforces the backward CFI for the embedded system by using ARM TrustZone-M similar to CEST. All of these works [8], [16], [17], [18] require the source code of the target program because they are implemented by modifying the compiler.

However, since most embedded systems do not provide source code and use customized build environments or compilers, it is difficult actually to apply these solutions. Unlike them, RECFISH [19] and CFI CaRE [20] are implemented by using binary patches without the need for source code. Specifically, they insert supervisor calls (SVCs) in the target binary to safely switch to the CFI verification code while the target program is running. However, these SVCs incur a considerable performance overhead whenever they are executed, so they cannot enforce CFI efficiently. CEST is also a CFI solution based on binary patch like RECFISH and CFI CaRE. However, CEST does not utilize SVCs, so CEST can enforce forward and backward CFI in an efficient way.

On the other hand, many studies are also being conducted to attach the special hardware module to embedded systems to ensure CFI, efficiently. The HCIC [22] implements the hardware-based CFI based on encrypted hamming distance (EHD) by attaching newly designed extra hardware such as a physical unclonable function (PUF), XOR operation hardware and an encrypted hamming distance calculation circuit (EHDCC). FI-CFI [21] pointed out that the EHD used in HCIC is still course-grained cause the EHD is not a unique methodology. FI-CFI implemented more fine-grained CFI by implementing CFI based on HMAC instead of EHD. Both papers take advantage of being able to defend against both jump-oriented programming (JOP) and return-oriented programming (ROP) attacks and do not need to modify the compiler or ISA. The CRAlert [26] detect real-time detection of code reuse attacks by adding a system consisting of a two-dimensional index table called branch instruction index (BID) and a component called double-linked wait list (DWL) between CPU core and memory. Although all three papers mentioned above show low runtime overhead, they can be applied only to embedded systems where developers can change hardware. However, without additional hardware, CEST can implement CFI schemes in an efficient way VI.

III. BACKGROUND

A. ARMv8-M

In ARMv8-M, the memory protection unit (MPU) manages access privileges to memory areas. Unlike the Memory Management Unit (MMU) used in high-end processors, the MPU does not support virtual memory; it only divides the physical memory address area into a limited number of areas to manage access privileges for each area.

Some manufacturers' chipsets provide hardware-based MPU locks to prevent the MPU from being attacked. In STM32L552ZE used for the implementation of CEST proposed in this paper, the function is managed in the SYSCFG CPU Non-secure Lock Register (SYSCFG_CNSLCKR). The MPU Registers Lock bit of SYSCFG_CNSLCKR can be set at any point during execution, but once established, it is not unset until the processor is reset. In our approach, it is assumed that the non-secure kernel is safely operated through these functions.

ARMv8-M can use 16 core registers. Among them, *R13* is used as a Stack Pointer (*SP*), *R14* as a link register (*LR*), and *R15* as a Program Counter (*PC*). Except for these three registers, the others are general-purpose registers. However, in AAPCS (procedure call standard for ARM architecture), the *R12* register is used as an intra-procedure call scratch register. ARMv8-M has only two modes, privileged mode and unprivileged mode.

B. TrustZone-M

ARM TrustZone-M (TZ-M) is TrustZone [27] for ARM Cortex-M processors and a kind of physical security technology made by ARM. TZ-M is divided into two different memory spaces: secure and non-secure.

Memory and peripherals such as flash memory and SRAM are divided into secure and non-secure. Secure memory is divided into secure and non-secure callable (NSC). The NSC is an area through which a user can enter the secure area from the non-secure area using the secure gateway (*SG*) instruction in the NSC.

Memory region attribution is determined by the security attribution unit (SAU) and Implementation Defined Attribution Unit (IDAU) [28]. IDAU is fixed to the manufacturer's initial setting value and cannot be modified by the programmer, and the authority of a specific area can be designated through SAU. IDAU divides memory area into non-secure and NSC, whereas SAU divides it into non-secure and secure area. When SAU is enabled, the memory defaults to the secure attribute. It can be switched to the non-secure attribute by configuration. The final security attribution of memory is set to the attribute having the higher value among the values specified in SAU and IDAU [29], [30].

C. MEMORY LAYOUT

Figure 1 shows the memory security attributes predefined by IDAU on the STM32L552ZE. It indicates that the non-secure flash area ranges from 0x08000000 to 0x08080000, and the NSC flash area starts at 0x0C000000. However, in the ARMv8-M instruction set, the maximum memory address that can be moved by the *B* and *BL* instruction corresponding to the direct jump is $\pm$0x00FFFFFF [29]. Therefore, it is impossible to move from the non-secure area to the code located in the NSC area using the direct jump instruction. In order to overcome this in CEST, it safely moves to the NSC area using indirect jump instruction such as *LDR PC, [PC + 4]*, where the destination address is stored in the code area that the attacker cannot modify.

D. CONTROL TRANSFER INSTRUCTIONS

ARMv8-M supports several types of control transfer instructions. These can be divided into direct and indirect depending on whether the instruction contains the destination address or not, and are classified into a jump (branch) and call (branch and link) depending on whether the return address is stored in the link register. ARMv8-M does not have dedicated instructions for return. However, following instructions are used for return in a function; *BX LR* and *MOV PC, LR*.

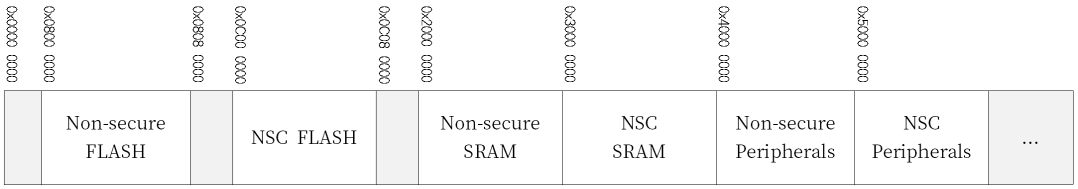


FIGURE 1. Example of STM32L552ZE memory layout defined by IDAU, NSC stands for 'Non-secure-callable'.

Therefore, in this paper, instructions are classified into five types: direct jump, direct call, indirect jump, indirect call, and return. Among them, the instruction that can be utilized for control hijacking attacks and the main protection target of our system are indirect call/jump and return. A direct call instruction is not used to control hijacking attacks, but CEST also processes the corresponding instruction to save the return address. Therefore, the instruction types patched by CEST are four types that exclude direct jump.

IV. THREAT MODEL AND ASSUMPTION

A. ASSUMPTION

- A1. All code memory are not writable
- A2. All data area are non-executable
- A3. Secure memory is isolated from non-secure memory

In a state-of-the-art embedded system, memory security policies such as W⊕X can be enforced using the MPU. The target system of our study is a system in which such a security policy exists. It is assumed that an attacker cannot modify the memory of the code area and that code execution in the data area is impossible. In addition, by using ARM TrustZone-M, the non-secure and secure areas are guaranteed to be completely separated, and it is assumed that unauthorized access to the secure area in the non-secure area is impossible. We presume that kernel and MPU are operating safely through settings such as (SYSCFG_CNSLCKR). Therefore, we aim to run the program safely when the user application is vulnerable to adversaries.

B. THREAT MODEL

- T1. Arbitrary read access to non-secure code memory
- T2. Arbitrary write access to non-secure data memory

We assume that the attacker can write access to arbitrary data areas through vulnerabilities in embedded systems written in languages vulnerable to memory corruption, such as C or C++, and hijack the control flow of normal processes. Moreover, as the attacker has read access to the code area, an attacker can collect gadgets to be used for the attack.

V. DESIGN

A. DESIGN GOALS

When we design CEST, we consider the following design goals.

- G1. Leveraging with TZ-M
  The first goal is to implement CEST on low-end embedded ARM processors without state-of-the-art technology such as BTI. Therefore, it offers the security features optimized on ARM TZ-M not the various security enforcement methodology on a high-end ARM processor. CEST guarantees reliability by locating it in a separate area from untrusted code using TZ-M.
- G2. Binary instrumentation
  In embedded systems, the source code is often not disclosed [24]. A variety of toolchains are utilized according to device vendors [19]. Hence, we designed CEST to be applied to the target software through binary instrumentation.
- G3. Low performance overhead
  Many embedded systems operate with strict real-time constraints where the task should be finished by the deadline. Thus, if CEST imposes a significant performance overhead on the system, several tasks in the system cannot be protected due to the real-time constraint. Therefore, it is crucial to minimize the performance overhead to make CEST a practical technique.

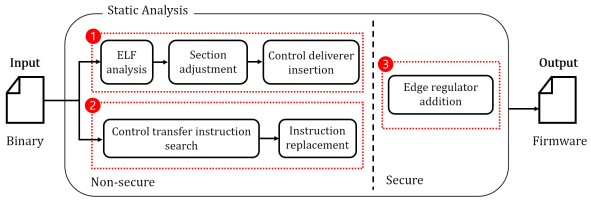


FIGURE 2. Overview of CEST.

B. DESIGN OVERVIEW

CEST enforces control flow integrity using ARM TrustZone-M to prevent control hijacking attacks in an embedded environment. CEST changes the control transfer instructions in which control flow hijacking can occur, such as indirect call, indirect jump, and return, to the instruction that can safely execute through a *control deliverer* and binary patch. When the changed instruction enters the *edge regulator* of the secure memory through the *control deliverer*, the *edge regulator* checks the control flow's legality so it can be safely changed.

In the following sections, we describe CEST by dividing it into three parts. Part ① in Figure 2 indicates the *control deliverer* in the non-secure region. It allows all control transfer instructions in the binary to move through the corresponding area to the *edge regulator* of the secure area. This is covered in Section V-D and deals with the process of creating space by adding sections and the role of *control*

deliverer region. Part ② deals with the binary patch that places the control transfer instruction in the non-secure binary into the instructions that can be safely executed. This is covered in Section V-C. Finally, part ③ represents the routines verifying forward and backward edges through *edge regulator* in run-time. This is covered in Section V-E.

C. BINARY PATCH

In the binary patch phase, all control transfer instructions except direct jump in the non-secure image are replaced with instructions to branch to the *control deliverer* through static analysis. The target instructions are direct call, indirect call/jump, and return instructions. A direct call is a function call with a fixed destination and does not belong to the control transfer instructions; it is processed to store the return address in the shadow stack for a return instruction to be executed later.

Function call instructions such as *BL* and *BLX* must store the return address in the shadow stack. Therefore, it is patched to enter the *control deliverer* using the direct call instruction *BL*. The return address is stored in the *LR* register when entering the *control deliverer* in runtime. As described earlier, the direct call instruction is not included in the control transfer instructions, so using the *BL* instruction to enter the *control deliverer* does not cause security problems.

Indirect jump instructions such as *BX*, *TBH*, and *LDR PC, R0* are also converted to *BL*, where the address at which the instruction is located (to be precise, the address of the following instruction) is contained in *LR* register so that it can be utilized in the *edge regulator*. The *LR* register is no longer used when returning a function within the non-secure region, so this method does not affect the operation of the program. The address of the following instruction is required to operate in the *edge regulator*, which will be described in detail in Section V-E. For the return instruction, neither the address of the instruction nor a return address is required. Therefore, it is patched to enter the *control deliverer* area using a direct jump.

For indirect call/jump and return, most of the instruction is 2 bytes in thumb mode. However, some instructions require 4 bytes of space after the patch. To solve this, CEST used the previous and current instructions to secure 4 to 6 bytes of space and proceed with the patch. After that, the previous instructions are executed in the *control deliverer*.

In return, there is only a 2 bytes return instruction in the basic block occasionally. Figure 3 shows an example of a case where only a 2 bytes return instruction is in the basic block. In this case, the previous instruction is located in another basic block (e.g., the instruction located at 0x0804BF14 in Figure 3). In other words, when the patch is performed in conjunction with the previous instruction, some instructions branch to the middle of the newly generated 4 bytes instruction, causing abnormal execution. This can be solved by moving both the previous basic block and the basic block that consists of the return instruction to the *control deliverer* area. This approach will allow a little more bytes to go into the *control deliverer* area, but CEST will still work

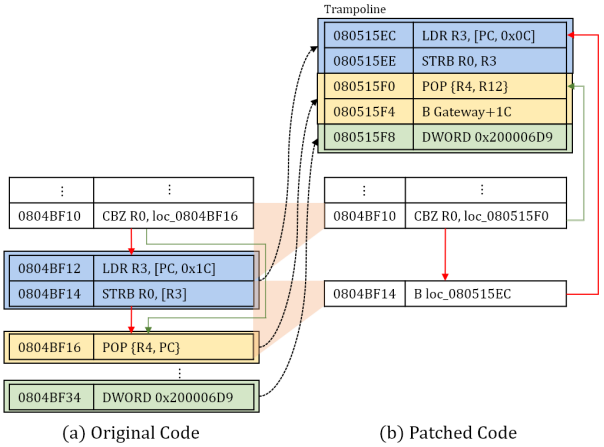


FIGURE 3. Example of patching 2 bytes return block. The left side is the original binary, and the right side is the patched binary. The instructions located at 0 × 0804*BF14* and 0 × 0804*BF16* are merged as single instruction located at 0 × 0804*BF14* in patched binary. The colored blocks in the original binary are relocated on the *trampoline* (blue, yellow, green). PC-relative instructions (e.g., 0 × 0804BF12) are updated to appropriate instruction by the situation.

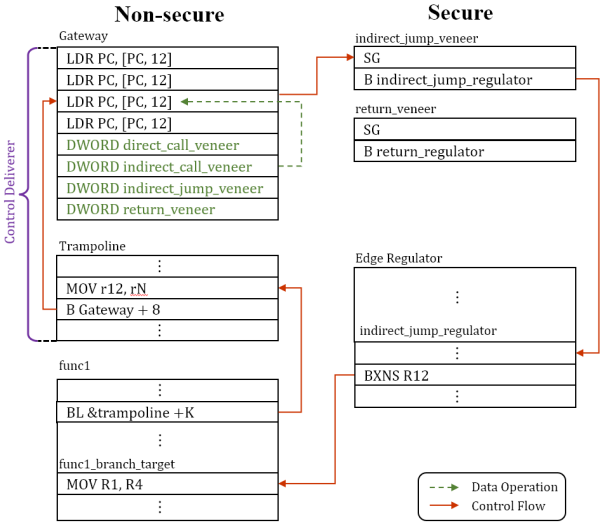


FIGURE 4. Example of CEST implemented binary. *control deliverer* consists of gateway and trampoline.

as before without issues. If there is a PC-relative instruction in the basic block to be moved, the processing of this instruction is also performed simultaneously. For example, in Figure 3, the instruction located at 0 × 0804BF12 in (a) is modified with appropriate offset at 0 × 080515EC in (b).

D. CONTROL DELIVERER

The *control deliverer* is split into two; *gateway* that exists for safe entry into the secure area, and *trampoline* that exists to set the argument to be passed to *edge regulator*. The *gateway* always has a fixed size of 32 bytes regardless of binary, and *trampoline* increases in size according to the number and type of instructions to be patched in the binary. Figure 4 shows how *trampoline* and *gateway* operate in a system to which CEST is applied through an example of indirect call. In Figure 4,

BL of func1 is a result of patching an indirect call instruction, showing that the control flow is moved to *trampoline*. Then, in *trampoline*, set the argument to be passed to *edge regulator*, move to the area corresponding to the indirect call of *gateway*, and enter the secure area.

1) TRAMPOLINE

Unlike previous studies [19], [20] that used the immediate field of *SVC* to distinguish the control transfer instruction after binary patching, CEST uses *BL* and *B* to carry out the same feature. Since no field can be used as a comment in these instructions, the original instruction cannot be distinguished by the method used in the previous paper. To mitigate this problem, CEST secures enough space for binary patching through the *trampoline* area, and adds instructions to identify the original instruction to *trampoline*. Therefore, the original instruction in the binary is fetched to move to a specific position within *trampoline* (e.g., *trampoline*+ offset). After that, instructions containing information about the original instruction to be transferred to *edge regulator* are added to the corresponding location (*trampoline*+offset).

The original instruction type to be patched and the instruction set added to the *trampoline* are described in Table 2. In the case of direct call instructions, the destination can be specified through static analysis. Therefore, in the binary patch step, the absolute address of the function to be called is calculated, and then the address is put in the *R12* register. In the direct jump/call instruction, the operation's destination is contained in the register, an operand of the instruction. However, the destination is unknown with static analysis, and the destination can only be specified at runtime. The operand is calculated within the *trampoline*, moved to the *R12* register, and entered into the *edge regulator* to process this.

The size of the *trampoline* increases linearly according to the number of control transfer instructions to be patched theoretically. However, while patching, if there are identical instructions in the non-secure area, the previously created instructions in *trampoline* are reused without adding new instructions. Therefore, the size of the finally created *trampoline* can be reduced. For example, the same function is called in different places in a non-secure area. In this case, the instructions corresponding to the function call are added in *trampoline* only once. The same function call instruction jumps to the instructions previously created in *trampoline*.

2) GATEWAY

The *gateway* refers to the first 32 bytes of the *control deliverer*. In the *gateway*, four *LDR PC, [PC, 0x0C]* instructions for entering from non-secure to secure and *edge regulator* address that is distinguished by the types of original control transfer instruction are stored. *LDR PC, [PC, 0x0C]* instruction corresponds to an indirect jump but it works as same with a direct jump. The *PC+0x0C* points to the 4 bytes address included in the code area, so attackers cannot modify it since the W⊕X policy. Therefore, the instruction can be safely used according to the previously set assumption. The reason for entering secure mode through this method is that a direct jump cannot be performed immediately due to physical limitations on the STM32L552ZE memory layout covered in Section III. In ARM, direct jump instruction has 2 or 4 bytes in length. As a result, the maximum range in which a direct jump can be performed is base address ±0x00FFFFFF, but since non-secure memory and secure memory are different by 0x04000000, *LDR PC, [PC+N]* instructions are used as a solution.

TABLE 2. List of patched instructions.

Type	Original Instruction	Instructions added into *control deliverer*
Direct Call	BL func	LDR r12, [PC, 4] B Gateway
Indirect Call	BLX rN	MOV r12, rN B Gateway + 4
Indirect jump	BX rN	MOV r12, rN B Gateway + 8
	LDR pc, [rA, rB, #N]	LDR r12, [rA, rB, #N] B Gateway + 8
	TBB [rA, rB]	LDRB r12, [rA, rB] ADD r12, LR, r12, LSL#1 B Gateway + 8
	TBH [rA, rB, LSL#1]	LDRH r12, [rA, rB, LSL#1] ADD r12, LR, r12, LSL#1 B Gateway + 8
Return	BX lr	B Gateway + 12
	MOV pc, lr	
	POP { rA-rB, pc }	POP { rA-rB , r12} B Gateway +12

Previous studies entered secure mode through the *SVC* handler, but CEST uses the *control deliverer* in the user code area to enter secure mode. This implementation method provides two main advantages. First, context switching does not occur because it is unnecessary to switch from user mode to supervisor mode. Therefore, it is possible to enter the secure area more efficiently and faster than in existing studies. This makes it possible to satisfy our goal G3, mentioned in Section V-A. Second, space for binary patching can be secured without affecting the layout of the existing code area. This enables us to satisfy our design goal G2 of implementing CEST using only binary without the need to recompile the source code.

3) SECURITY CONSIDERATION

The *R12* register transfers an argument to the *edge regulator* on implementing the CEST. According to the ARM architecture procedure call standard (AAPCS), the *R12* register is a scratch register for intra-procedure calls, especially for long branch veneers. The *R12* register is a register that does not need to be backed up in call [31]. Even if the *R12* register is used in the intra-procedure call, those instructions will be replaced by another instruction in the binary patch stage because intra-procedure call instruction is a control transfer instruction that can be exploited to control hijacking attacks. Therefore, if the original binary is a binary following AAPCS, booking the *R12* register in CEST does not affect the operation of the basic program.

Using the *control deliverer* and *R12* registers do not affect the security level of CEST. In the binary patch phase of

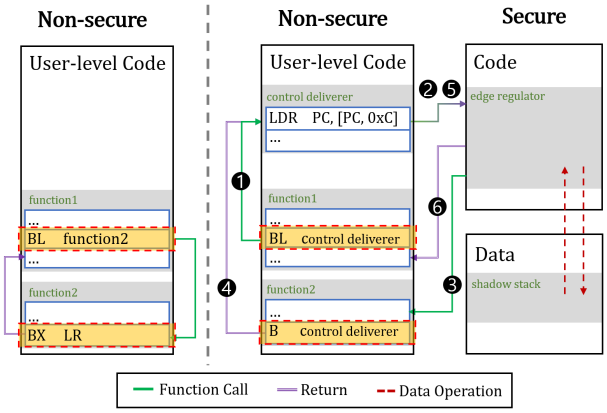


FIGURE 5. Summarized flow of direct call and return in the CEST implemented system. The left side shows the original binary, and the right side shows CEST implemented system. The highlighted instructions in function1 and function2 are patched into an instruction to enter *control deliverer*.

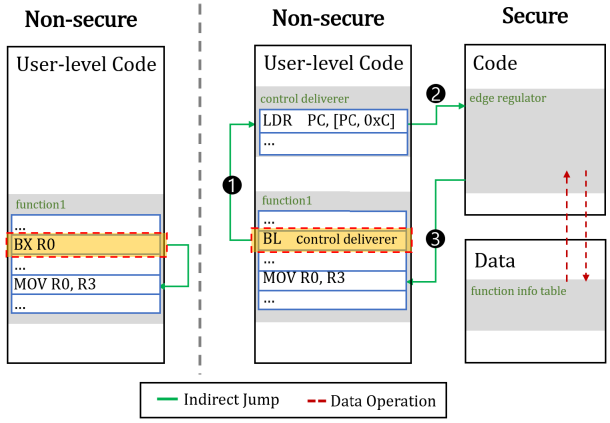


FIGURE 6. Flow of indirect jump in CEST.

CEST, all instructions using the *R12* register in the binary are changed. Even if an attacker uses an instruction that modulates the *R12* register in the *trampoline*, the attacker cannot change the control flow with manipulated *R12* register. This is because CEST enters the *edge regulator* immediately after changing the *R12* register value in the *trampoline*. To be specific, even if an attacker manipulates the *R12* register by using instructions in *trampoline* generated from indirect call/jump as a gadget, *edge regulator* can verify whether the *R12* register contains an appropriate value. A detailed description of this verification method is provided in Section V-E.

In addition, CEST entered *edge regulator* directly without going through the supervisor mode. However, the operation mechanism of CEST does not have a significant adverse effect on security compared to the supervisor mode. In order to enter the NSC area from the non-secure area, the instruction must go through the *SG* (secure gateway), so it is impossible to switch from the non-secure area to an arbitrary secure area. Therefore, the attacker can only access a limited area even if the attacker enters the secure area by manipulating the data in a non-secure area. Before entering the secure area, security can be further strengthened by adding argument sanitizing in the NSC area if required.

E. EDGE REGULATOR

The *edge regulator* consists of four types, depending on the instruction type to be processed: direct call, indirect call, indirect jump, and return. A *function info table* is required for the *edge regulator*'s operation. This table stores the entry address of functions in the non-secure user code area and the size of the function. This information is obtained from information analyzing the symbol in the binary patch step.

There is no need to verify the forward edge for the direct call because it is a function call with a fixed destination. Therefore, it is implemented only for saving the return address in the shadow stack and jumping to the function to be called. As explained in Section V-D, In *control deliverer*, CEST stores the function address into *R12* register and *edge regulator* branches to the destination using *R12* register. This process corresponds to ① → ② → ③ in Figure 5.

For indirect calls, CEST firstly checks whether the destination address is in NSC area.This is because the existing NSC call existing in the binary of the non-secure area also enters the NSC area through the *BLX* instruction. Since an NSC call can only move to the *SG* instruction, it fetches 4 bytes of the memory address and verifies whether it matches the *SG* instruction. If the destination is located in a non-secure zone, CEST must handle both the backward edge and forward edge. The backward edge is handled by storing the return address in the shadow stack same as a direct call. The forward edge is verified whether the destination is in the *function info table* because the destination must be the entry of the functions. CEST terminates the program if the target address is not in the *function info table*.

An indirect jump is mainly used to process switch-case statements and goto statements inside the function. It means that the range that can be moved through the indirect jump must be within the scope of the corresponding function. Therefore, the *edge regulator* processes the forward edge by identifying the function where the instruction is located and then examining whether the destination is within the boundaries of that function. When entering the *control deliverer*, this instruction is patched to the *BL* instruction, so that the *edge regulator* can know the address of the instruction through the *LR* register. The *edge regulator* searches the *function info table* to identify the function where this instruction is located and verify the destination stored in the *R12* register is within the boundaries of the function.

As return instruction is not for the forward edge, the *edge regulator* does not process it for the forward edge. Instead, the return address is recovered from the shadow stack that is saved at the time of the function call and returned there. This operation corresponds to ④ → ⑤ → ⑥ in Figure 5.

VI. EVALUATION

A. ENVIRONMENT SETUP

The environment setting for the testing is carried out on the STMicroelectronics Nucleo-L552ZE-Q [32] board

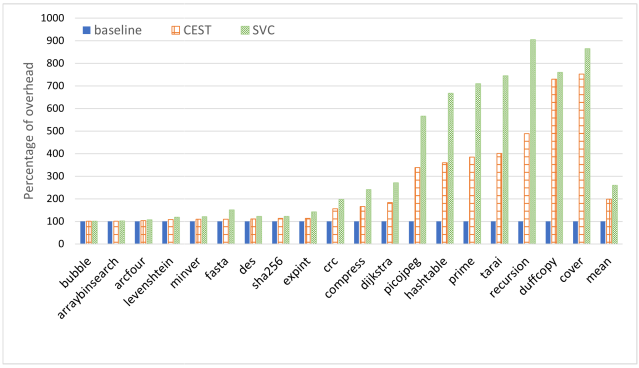


FIGURE 7. Runtime overhead graphs for 19 benchmarks. The blue solid bar graph is the baseline and execution time of the unmodified code. The hatched orange bar graph represents the runtime overhead of `CEST`. The green dotted bar graph shows the execution time overhead of the code, assuming that SVC is used. The baseline is always 100%, and the graph shows how much overhead it has compared to the baseline.

with the ARM Cortex-M33 processor supporting the ARM TrustZone-M technology and using the instruction set architecture of the ARMv8-M [29]. Zephyr OS [33] is installed on the target board to implement the `CEST`. Radare2 [34] is used as a library for the binary patch. A python library, pyelftools [35], is used to analyze the ELF and add a new area *control deliverer*. We evaluate the performance of `CEST` using BEEBS [36] the open-source benchmark software.

B. MICRO-OPERATION

The most significant difference in our implementation compared to previous papers is that we enter the *edge regulator* directly in the secure area from the non-secure user without going through the supervisor mode. In other words, assuming that the overhead of the routine verifying the integrity of each edge is similar, the part showing the most significant difference in execution time will be the time required to enter the part verifying the integrity of the edge. Therefore, before measuring the overhead of the entire system, we measured overhead by implementing the cases of entering the secure area from supervisor mode through *SVC* instruction and entering the secure area through the *control deliverer* in the non-secure user area. Entering the integrity verification routine via *SVC* (previous works) shows 145 cycles overhead while entering the integrity verification routine using the *control deliverer* (our implementation) requires only 44 cycles. This proves that the SVC-based method requires approximately 3.29 times more cycles than the trampoline-based method. This is an overhead that occurs when the *SVC* handler is not implemented, and a larger overhead will appear if the *SVC* handler is being used to implement functions such as system calls. When we assume this situation, 361 cycles in the existing implementation whereas there are only 44 cycles in our implementation, resulting in about 8.2 times more overhead in the current works.

C. RUN-TIME OVERHEAD

Two types of implementation were carried out in figure 7 to evaluate the run-time overhead. The first implementation is an implementation that checks both the integrity of the backward and forward edge which is fully functional of `CEST`. The second is implemented for comparison with the previous paper [19], [20], checking the run-time overhead when entering the branch verification routine using *SVC*. The baseline is a run-time of non-modified benchmark source implementation. Based on the micro-operation testing results, the existing approach that used the *SVC* has 100 cycles of overhead compared with ours. So we compare the overhead on run-time by adding dummy instructions to our implementation. Figure 7 shows the run-time overhead for 19 benchmarks included in BEEBS. The maximum execution time overhead for our implementation is observed as 652.27% compared to the baseline, and the minimum is observed as 0.20% tested with bubble sort benchmark. For the 19 benchmarks used in our evaluation, an average execution time overhead of 159.30% was observed compared to the baseline.

As can be seen from the trend shown in the graph, the 19 benchmark results can be analyzed by dividing them into three parts. In the case of the two benchmarks at the right end (`duffcopy` and `cover`), a substantial overhead appears compared to other benchmarks, It can be seen that the run-time overhead of the implementation using *SVC* and the implementation using `CEST` is relatively small. The `cover` benchmark is an artificially created program to test the indirect jump, and significantly more indirect jump instructions are executed compared to the general program. In these two benchmarks, many table searches occur to safely process indirect control transfer instructions, which is the reason for the high execution overhead of these benchmarks. Similarly, since the table search overhead occupies a large proportion compared to other overheads, this test case shows relatively few advantages of our implementation using *control deliverer* instead of *SVC*.

Next, the `picojpeg`, `hashtable`, `prime`, `tarai`, and `recursion` benchmarks also present large overheads. These benchmarks are cases where the overhead is large because there are many control transfer instructions compared to the program size, or the number of indirect jump and indirect calls is large. This is a typical result due to the nature of the implementation that validates the control transfer instruction and shows how efficiently our implementation using the *control deliverer* can reduce the execution time compared to the *SVC* implementation in this situation. For all the benchmarks performed, the average execution time overhead for processing one control transfer instruction is 87.25 cycles.

In the experiment conducted in this study, only the time difference between the method using *SVC* and the *control deliverer* is measured based on the results of micro-operation. Our implementation can be more effective in the real-world scenario than in experiment. This is because `CEST` does not take table searching time to process direct call instructions. In `CEST`, all necessary parameters are set before entering the *edge regulator* using the *control deliverer*; hence, there is no need for table searching to identify the destination

TABLE 3. Binary size overhead. The ratio is the size of the binary that is increased compared to the size of the non-secure area. Edge regulator has a constant size regardless of benchmark binary.

	Bytes	Ratio
Control Deliverer	207.37	1.09%
Function Info Table	1376.84	7.36%
Edge regulator	608	

address of a direct call. Considering that the table searching overhead for processing indirect call/jump is exceptionally large in the experimental result, it can be predicted that the implementation of CEST is more efficient than that shown in the experiment.

D. BINARY SIZE

The most significant part of the memory size that increases due to the implementation of CEST is the branch table that contains the entry and size of all functions located in a non-secure area. This is 1376.84 bytes on average for 19 benchmarks, accounting for 7.36% of the size of the non-secure area. In the case of the *control deliverer*, it increases according to the number of control transfer instructions. However, memory waste can be reduced by reusing the previously created set of instructions when using the same instruction. Therefore, the memory size occupied by the *control deliverer* compared to the entire non-secure image is only 1.09% on average. It is impossible to compare the binary size overhead described directly numerically in the implementation in CFI CaRE [20] with the overhead of our study. However, in the case of the branch table, as it is used to process the forward edge in the implementation of the existing research, the binary size of our study will generate a similar level of overhead. In addition, in CFI CaRE [20] and RECFISH [19], an additional table is used to specify the return address of a direct call. This value increases linearly as much as the number of control transfer instructions. Therefore, it is expected to have a similar overhead to the *control deliverer* in this study.

VII. DISCUSSION

In this study, patching is carried out and evaluated only for the non-secure user area. However, codes in the kernel area can also enter the secure area more quickly using the *control deliverer*. For complete protection of this kernel area, as implemented in other papers, it is necessary to implement the exception shadow stack to perform exception return safely. We implemented a solution to prevent control flow hijacking using shadow stack and branch regulation. However, this is an example of the implementation presented in this paper. Existing CFI solutions monitor all indirect branches in the binary, and our implementation also monitors all indirect branches in the same way, so it is theoretically possible to apply a solution other than branch regulation. It is also possible to extend this technique to other methodologies through the same approach. It is also possible to check the legitimacy of memory access by fetching further instructions (e.g., memory load/store) and modifying the *edge regulator* in the secure area instead of fetching control transfer instructions.

VIII. CONCLUSION

This paper presents CEST that enforces the CFI with low overhead utilizing the ARM TZ-M and the *control deliverer*. CEST secures the shadow stack and *function info table* by locating the assets of CEST in the ARM TZ-M. Moreover, CEST provides binary compatibility considering the characteristic of the embedded system environments that use various compile environments and tools. CEST used the *control deliverer* instead of *SVC* used in existing studies to reduce the execution time. Through efficient design, the performance improvement is approximately 40.76% compared to the implementation through *SVC*. In the *SVC* implementation, table search is required for CFI enforcement of direct call instructions, but in CEST, CFI can be enforced without table search using *control deliverer* Therefore, the performance improvement in the actual program is expected to be higher than benchmarks in this study.

REFERENCES

[1] K. Bing, L. Fu, Y. Zhuo, and L. Yanlei, "Design of an Internet of Things-based smart home system," in *Proc. 2nd Int. Conf. Intell. Control Inf. Process. (ICICIP)*, vol. 2, 2011, pp. 921–924.

[2] J.-C. Wang, C.-H. Lin, E. Siahaan, B.-W. Chen, and H.-L. Chuang, "Mixed sound event verification on wireless sensor network for home automation," *IEEE Trans. Ind. Informat.*, vol. 10, no. 1, pp. 803–812, Feb. 2014.

[3] V. Sivaraman, H. H. Gharakheili, A. Vishwanath, R. Boreli, and O. Mehani, "Network-level security and privacy control for smart-home IoT devices," in *Proc. IEEE 11th Int. Conf. Wireless Mobile Comput., Netw. Commun. (WiMob)*, Oct. 2015, pp. 163–167.

[4] B. Chen, J. Wan, L. Shu, P. Li, M. Mukherjee, and B. Yin, "Smart factory of Industry 4.0: Key technologies, application case, and challenges," *IEEE Access*, vol. 6, pp. 6505–6519, 2017.

[5] W. Rong, G. T. Vanan, and M. Phillips, "The Internet of Things (IoT) and transformation of the smart factory," in *Proc. Int. Electron. Symp. (IES)*, Sep. 2016, pp. 399–402.

[6] N. Shariatzadeh, T. Lundholm, L. Lindberg, and G. Sivard, "Integration of digital factory with smart factory based on Internet of Things," *Proc. CIRP*, vol. 50, pp. 512–517, Jun. 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2212827116305066

[7] S. Wang, D. Li, J. Wan, and C. Zhang, "Implementing smart factory of Industrie 4.0: An outlook," *Int. J. Distrib. Sensor Netw.*, vol. 12, no. 1, 2016, Art. no. 3159805, doi: 10.1155/2016/3159805.

[8] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *Proc. 29th USENIX Secur. Symp. (USENIX Secur.)*, Aug. 2020, pp. 1219–1236. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-jie

[9] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 48–62.

[10] *Pax Team*. Accessed: Dec. 19, 2022. [Online]. Available: http://pax.grsecurity.net

[11] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted code randomization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 461–477.

[12] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling address space layout randomization for SGX programs," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[13] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 763–780.

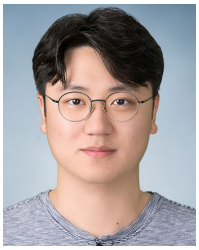
[14] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *Proc. 12th USENIX Symp. Operating Syst. Design Implement. (OSDI)*. Savannah, GA, USA: USENIX Association, Nov. 2016, pp. 367–382. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/williams-king

[15] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 1–40, Oct. 2009, doi: 10.1145/1609956.1609960.
[16] N. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Return address integrity for embedded systems," Sandia Nat. Lab. (SNL-NM), Albuquerque, NM, USA, Tech. Rep. SAND2020-0869C, 2020.
[17] 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 *Proc. 31st USENIX Secur. Symp. (USENIX Secur.)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2281–2298. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/du
[18] T. Kawada, S. Honda, Y. Matsubara, and H. Takada, "TZmCFI: RTOS-aware control-flow integrity using TrustZone for Armv8-M," *Int. J. Parallel Program.*, vol. 49, no. 2, pp. 216–236, Apr. 2021.
[19] R. J. Walls, N. F. Brown, T. L. Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-flow integrity for real-time embedded systems," in *Proc. 31st Euromicro Conf. Real-Time Syst. (ECRTS)*, vol. 133, S. Quinton, Ed. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 2:1–2:24. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10739
[20] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*. Springer, 2017, pp. 259–284.
[21] A. Fu, W. Ding, B. Kuang, Q. Li, W. Susilo, and Y. Zhang, "FH-CFI: Fine-grained hardware-assisted control flow integrity for ARM-based IoT devices," *Comput. Secur.*, vol. 116, May 2022, Art. no. 102666, doi: 10.1016/j.cose.2022.102666.
[22] J. Zhang, B. Qi, Z. Qin, and G. Qu, "HCIC: Hardware-assisted control-flow integrity checking," *IEEE Internet Things J.*, vol. 6, no. 1, pp. 458–471, Feb. 2019.
[23] *Branch Target Identification*. Accessed: Dec. 19, 2022. [Online]. Available: https://developer.arm.com/documentation/ddi0596/2020-12/Base-Instructions/BTI–Branch-Target-Identification
[24] P. Kiaei, C.-B. Breunesse, M. Ahmadi, P. Schaumont, and J. Van Woudenberg, "Rewrite to reinforce: Rewriting the binary to apply countermeasures against fault injection," in *Proc. 58th ACM/IEEE Design Automat. Conf. (DAC)*. IEEE, 2021, pp. 319–324.
[25] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," *SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 94–105, Jun. 2012, doi: 10.1145/2366231.2337171.
[26] W. Wang, G. Hu, X. Xu, and J. Zhang, "CRAlert: Hardware-assisted code reuse attack detection," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 3, pp. 1607–1611, Mar. 2022.
[27] P. Guo, Y. Yan, C. Zhu, and J. Wang, "Research on arm TrustZone and understanding the security vulnerability in its cache architecture," in *Proc. Int. Conf. Secur., Privacy Anonymity Comput., Commun. Storage*, 2020, pp. 200–213.
[28] *Trustzone Technology for ARMV8-M Architecture*. Accessed: Dec. 19, 2022. [Online]. Available: https://developer.arm.com/documentation/100690/0201/
[29] *ARMV8-M Architecture Reference Manual, 284–286, 584–586*. Accessed: Dec. 19, 2022. [Online]. Available: https://developer.arm.com/documentation/ddi0553/latest
[30] *STM32l552XX Datasheet, Ultra-Low-Power Arm Cortex-M33 32-Bit MCU+Trustzone+FPU, 165 DMIPS, Up to 512 KB Flash Memory, 256kb SRAM, SMPS*. Accessed: Dec. 19, 2022. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32l552cc.pdf
[31] *On the AAPCS, With an Application to Efficient Parameter Passing*. Accessed: Dec. 19, 2022. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/on-the-aapcs-with-an-application-to-efficient-parameter-passing
[32] STMicroelectronics. *STM32 NUCLEO-144 Development Board With STM32l552ZE MCU, SMPS, Supports Arduino, ST ZIO and Morpho Connectivity*. Accessed: Dec. 19, 2022. [Online]. Available: https://www.st.com/en/evaluation-tools/nucleo-l552ze-q.html
[33] *Zephyr OS*. Accessed: Dec. 19, 2022. [Online]. Available: https://www.zephyrproject.org/
[34] *Radare2: The Libre Unix-Like Reverse Engineering Framework*. Accessed: Dec. 19, 2022. [Online]. Available: https://github.com/radareorg/radare2
[35] *PyelfTools, a Pure-Python Library for Parsing and Analyzing Elf Files and Dwarf Debugging Information*. Accessed: Dec. 19, 2022. [Online]. Available: https://github.com/eliben/pyelftools
[36] J. Pallister, S. J. Hollis, and J. Bennett, "BEEBS: Open benchmarks for energy measurements on embedded platforms," *CoRR*, vol. abs/1308.5174, Dec. 2013.

GISU YEO is currently pursuing the bachelor's degree in computer science and engineering with the Department of Electrical and Computer Engineering, Pusan National University, Busan, Republic of Korea. His research interests include cyber-physical systems and embedded system security.

YERYEONG KIM is currently pursuing the bachelor's degree in computer science and engineering with the Department of Electrical and Computer Engineering, Pusan National University, Busan, Republic of Korea. Her research interests include robotics, compiler, and operating systems.

SUHYEON SONG received the B.S. degree in computing systems from the Unitec Institution of Technology, Auckland, New Zealand, in 2021. His research interests include computer system security and ARM TrustZone.



DONGHYUN KWON received the B.S. and Ph.D. degrees in electrical and computer engineering from Seoul National University, South Korea, in 2012 and 2019, respectively. He is currently a Professor with the School of Computer Science and Engineering, Pusan National University, South Korea. His research interest includes system security against various types of threats.