

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

XCFI: Comprehensive Control-Flow Integrity for Arm TrustZone-M

Yunju Gu, Jaeyeol Park, and Donghyun Kwon, *Pusan National University*

https://www.usenix.org/conference/usenixsecurity26/presentation/gu-yunju

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

XCFI: Comprehensive Control-Flow Integrity for Arm TrustZone-M

Yunju Gu
Pusan National University
ventus07@pusan.ac.kr

Jaeyeol Park
Pusan National University
woduf0628@pusan.ac.kr

Donghyun Kwon*
Pusan National University
kwondh@pusan.ac.kr

Abstract

Arm TrustZone-M (TZ-M) provides hardware-based isolation for tiny embedded systems by partitioning execution into *secure* and *non-secure* states. However, despite this isolation, memory vulnerabilities in software running within either state can still be exploited to perform control-flow hijacking attacks. To mitigate these threats, numerous *control-flow integrity* (CFI) studies have been proposed for embedded systems, but they have several limitations: many neglect the secure state, fail to protect control-flow events during TZ-M security state transitions, or incur prohibitive performance overhead.

In this paper, we present XCFI, a comprehensive CFI mechanism for TZ-M. XCFI employs a 32-bit *control-flow identifier* (CID) based protection to uniformly enforce fine-grained CFI across both secure and non-secure states, covering all control-flow events, including indirect branches, returns, and exceptions. Crucially, XCFI extends CFI enforcement to *cross-state control-flow events* in TZ-M, which have not been protected by prior work. Overall, XCFI provides comprehensive control-flow protection across all TZ-M execution contexts while incurring only modest runtime overhead.

1 Introduction

As tiny embedded systems increasingly process sensitive data, Arm introduced TrustZone-M (TZ-M) [6], a lightweight trusted execution environment designed for the resource-constrained Armv8-M environment. Similar to its application-processor counterpart, TrustZone-A (TZ-A) [5], TZ-M enforces hardware-level isolation by partitioning execution into *secure* (S) and *non-secure* (NS) states. However, to minimize latency and system complexity, TZ-M adopts a markedly different design: while TZ-A relies on a dedicated monitor mode to mediate security state transitions in software, TZ-M enables direct hardware-assisted transitions via dedicated call and return instructions, allowing the processor to switch states automatically without an intermediate privilege mode.

Despite this isolation, software in both states is predominantly written in memory-unsafe languages such as C and C++, leaving TZ-M systems vulnerable to memory corruption errors that can be exploited to mount control-flow hijacking attacks. *Control-flow integrity* (CFI) [1] has thus emerged as a fundamental defense, and numerous CFI mechanisms [3, 15, 17, 27, 41, 48, 57, 59, 62] have been proposed for microcontrollers (MCUs) to protect *control-flow events*, including indirect branches (i.e., forward-edge), returns (i.e., backward-edge), and exceptions.

However, existing CFI mechanisms for MCUs suffer from several limitations. First, prior works do not adequately account for control-flow events unique to TZ-M, particularly those arising during transitions between the S and NS states. As a result, cross-state indirect branches, returns, and exceptions remain unprotected, and recent attacks such as `ret2ns` [37,38] have demonstrated that these events can be exploited to subvert intended CFI enforcement. In such attacks, execution is redirected to attacker-controlled NS code by corrupting return contexts or function targets involved in S/NS transitions. Second, many existing works [27, 41, 48, 57, 59] fail to provide comprehensive protection for the S state itself, instead treating it primarily as a trusted computing base (TCB) for storing CFI metadata (e.g., shadow stack) used to protect the NS state. Consequently, software executing within the S state remains vulnerable to intra-state control-flow hijacking attacks. Third, several proposals adopt coarse-grained CFI mechanisms that limit their security effectiveness; for example, Silhouette [62], Kage [17], and SuM [15] provide coarse-grained protection for forward-edge control-flow transfers. Finally, approaches [48, 57] that rely on tracing hardware often incur prohibitive performance overhead; for example, InsectACIDE [57] reports an overhead of 149%.

To address these limitations, we present XCFI, an efficient and comprehensive CFI mechanism for TZ-M-enabled MCUs. XCFI leverages the pointer authentication and branch target identification (PACBTI) extensions of the Armv8.1-M to pro-

*Corresponding Author.

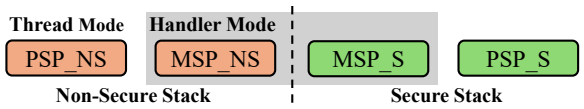


Figure 1: Active stack configuration across security states and execution modes in TZ-M.

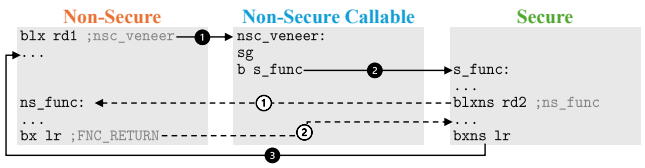


Figure 2: Security state transitions. Solid lines indicate NS-S-NS transitions, while dashed lines indicate S-NS-S transitions.

vide uniform and efficient control-flow protection across NS and S states. To overcome the limitations of naive PAC- or BTI-based designs [4, 31, 32, 60], XCFI introduces a unified *control-flow identifier* (CID) based CFI mechanism that enables fine-grained enforcement over all control-flow events. Furthermore, by analyzing TZ-M security state transition procedure and coordinating architectural features in Armv8.1-M, XCFI extends CFI enforcement to *cross-state control-flow events*, which are not addressed by existing approaches.

We implement a prototype of XCFI on the Armv8.1-M architecture with TZ-M and demonstrate its performance and applicability through evaluation on a set of workloads. The contributions of this paper are summarized as follows:

- We design XCFI to provide a fine-grained and unified CFI mechanism based on CID, protecting indirect branches, returns, and exceptions in practical embedded systems.
- We extend a cross-state CFI design for TZ-M environment that protects both intra-state control flows and control-flow transfers between the S and NS states, including security state transitions and exception handling.
- We demonstrate the practicality of XCFI through an implementation on a TZ-M-enabled MCU and an evaluation on representative workloads, including BEEBS, CoreMark-Pro, and Zephyr-based Trusted Firmware-M (TF-M) workloads, showing its applicability to realistic TZ-M environments with practical performance and flash memory overhead.

2 Background

Execution Modes and Exception Handling. Arm Cortex-M processors support two execution modes: thread mode and handler mode. Application code typically executes in thread mode, while interrupts and exceptions are handled in handler mode. Thread mode may execute with either privileged or unprivileged access, whereas handler mode always executes with privileged access. The active stack pointer (`SP`) depends on the current execution mode. In thread mode, the process stack pointer (`PSP`) is used, while in handler mode, the main stack pointer (`MSP`) is used.

When an interrupt or exception occurs, the hardware saves the current execution context to the stack that was active at the time of the exception before transferring control to the appropriate handler. The saved context includes the program counter (`PC`), the link register (`LR`) holding the return address, and the program status register (`xPSR`). The hardware also writes a special return value, `EXC_RETURN`, to `LR`. This value encodes information such as the execution mode and security state at the time the exception occurred. When `EXC_RETURN` is loaded into the `PC`, the hardware restores the saved state and resumes execution from the point at which the exception occurred.

Security States. TZ-M partitions the system memory into S and NS memory, enforced by hardware. Based on this partitioning, the processor operates in one of two corresponding security states, depending on where the currently executing code resides. Code executing from NS memory runs in the NS state, while code executing from S memory runs in the S state. Security state also determines memory access permissions: the NS state can access only NS memory, whereas the S state can access both S and NS memory.

Security state defines how stacks and registers are managed. As shown in Figure 1, `SP` is banked for each security state, allowing the S and NS states to maintain separate stacks. S state uses `PSP_S` and `MSP_S` for thread and handler modes, respectively, while NS state uses `PSP_NS` and `MSP_NS`. Conversely, general-purpose registers (`R0-R12` and `LR`) are shared between states and are not hardware-banked. Therefore, to prevent inference of the S execution context in the NS state, Arm recommends clearing registers except for function arguments when transitioning from the S state to the NS state [8].

Security State Transitions. Security state transitions are strictly controlled by hardware-defined mechanisms, as illustrated in Figure 2. Transitions from the NS state to the S state are permitted only through designated non-secure callable (NSC) entry functions in S memory, referred to as NSC veneers. Each NSC veneer must begin with a secure gateway instruction (`SG`). When execution in the NS state branches to an NSC veneer and the `SG` is executed (❶), the hardware transitions the processor to the S state. Control is then transferred to the target S function (❷). After the S function completes, control returns to the original NS caller (❸).

In contrast, transitions from the S state to the NS state are performed using interstate branch instructions such as `BLXNS` or `BXNS` (①). When an NS function is invoked from

the S state, the hardware saves the return address and execution context on the S stack and sets a special return value, FNC_RETURN, in the LR. When FNC_RETURN is loaded onto the PC, the hardware restores the saved context from the S stack and resumes execution in the S state (②).

Pointer Authentication. Pointer authentication (PA) is a hardware mechanism that protects the integrity of register contents by generating and verifying a pointer authentication code (PAC) using the PAC and AUT instructions. PAC computes a PAC from a secret key and two modifiers, and embeds the resulting PAC into the destination register (e.g., PAC R12, SP, LR). AUT computes a PAC using the provided key and modifiers and compares it against the PAC stored in the destination register, raising a fault on a mismatch (e.g., AUT R12, SP, LR). By default, PAC/AUT use a fixed register combination, R12 as the destination register and SP and LR as modifiers, whereas the extended PACG/AUTG pair allows arbitrary registers combinations. PAC key is supplied via a system register and varies across security states and privilege levels. This mechanism naturally supports backward-edge CFI by signing return addresses at function entry and authenticating them at return.

Branch Target Identification. Branch target identification (BTI) enforces forward-edge CFI by restricting indirect branches to designated landing pads. Indirect branches set the B bit in the execution program status register (EPSR), part of the xPSR, and execution is permitted only if the target begins with a BTI instruction or, in the TZ-M environment, an SG instruction. Executing any other instruction while the EPSR.B bit is set triggers a fault. This constraint is preserved across exceptions and security state transitions, as the EPSR.B bit is automatically saved and restored and is not banked per security state. When BTI is enabled, the SG instruction also clears the EPSR.B bit, serving as a valid landing pad while performing a security state transition. Consequently, NSC veneers do not require an additional BTI instruction at their entry points.

Memory Protection Unit. The memory protection unit (MPU) enforces access control by partitioning the address space into configurable regions with defined read, write, and execute permissions. In TZ-M, S and NS MPU regions are configured independently, and memory accesses are checked against the MPU configuration of the current security state: S-state accesses follow the S MPU, while NS-state accesses follow the NS MPU. When the MPU is enabled, all accesses are enforced by these policies and violations raise a fault. Memory addresses not covered by any MPU region may be accessible in privileged mode according to the system address map, while unprivileged accesses are prohibited.

Test Target Instructions. Starting from Armv8-M, the architecture provides test target (TT) instructions that allow querying the memory attributes of a given address without performing an actual access. The result is written to the destination register as a TT_RESP value, which encodes MPU-based access permissions, security-related properties, and the matching MPU region information (e.g., validity and region number). When executed in NS state, TT instructions do not reference the S MPU and do not return security-related properties. This design prevents the NS state from inferring the layout or protection policies of S memory. Among the TT instructions, the TTT (Test Target, Unprivileged) instruction reports the access permissions that would apply to a given address under unprivileged access.

3 Threat Model and Assumptions

XCFI targets TZ-M-enabled MCUs based on the Armv8.1-M architecture, where architectural security features such as PACBTI and MPU are enabled. We assume a standard secure boot process that establishes initial system integrity and generates a fresh PAC key at each system boot. We adopt a typical TZ-M execution model in which the main application and RTOS kernel execute in the NS state, while S state provides secure services to handle requests from the NS state [53]. As commonly assumed in MCU-based RTOS [12, 20, 49], all software components—including applications, libraries, and the kernel—execute with privilege in a single address space. Following the Armv8-M Security Extension [10], we assume that S exceptions are configured with the highest priority. In line with prior work [3, 41, 48], we assume an execution environment in which executable code resides in immutable flash memory, including the exception vector table, while all other memory is configured as non-executable, enforced by an MPU-based W⊕X policy. NS and S memory regions are protected by their respective MPUs. Architectural mechanisms such as Debug Watchpoint and Trace (DWT) are assumed, following prior work [15, 46, 47], to protect critical hardware security configurations, including MPU settings and the vector table offset register.

We assume a powerful software attacker who can exploit software vulnerabilities that enable memory corruption in both the S and NS states to induce arbitrary writes to S or NS SRAM. Through such vulnerabilities, the attacker may corrupt control-flow-critical data, such as return addresses, function pointers, or saved execution contexts, and hijack control flow within the S or NS state, as well as across security state transitions. Accordingly, our threat model encompasses control-flow hijacking attacks, including return-oriented programming (ROP) [45] and jump-oriented programming (JOP) [13]. Non-control-data attacks [14, 23], such as attacks on ordinary data objects, are outside the scope of XCFI. Protecting the confidentiality and integrity of such data, or providing finer-grained compartmentaliza-

tion within a security state, is orthogonal to XCFI and can be addressed by complementary defenses such as compartmentalization [28–30] or MPU-based isolation [17, 21, 63]. DMA-based attacks [39] and side-channel attacks [44] are also considered out of scope, as this work focuses exclusively on control-flow hijacking.

4 Design

This section describes the design of XCFI. § 4.1 introduces the CID used throughout XCFI for CFI enforcement, while § 4.2 and § 4.3 explain how XCFI enforces CFI for intra-state and cross-state control-flow events, respectively. § 4.4 presents the handling of S state re-entry with an existing upper-level S context.

4.1 Control-Flow Identifier

XCFI builds on PACBTI to enforce CFI. However, PA is vulnerable to reuse attacks that replay previously computed PACs [31, 32, 60], and BTI provides only coarse-grained protection [4], as it permits branches to any marked landing pad. To address these limitations, XCFI introduces a CID.

A CID is a 32-bit identifier uniquely assigned to each function. To enable fine-grained CFI across forward-edge, backward-edge, and exception handling, each CID is composed of two 16-bit fields:

- **Unique Identifier (UID)**: uniquely identifies each function and occupies the upper 16 bits of the CID.
- **Signature Identifier (SID)**: represents the set of valid targets for indirect calls and serves as the primary identifier for forward-edge CFI. It occupies the lower 16 bits of the CID.

This 16-bit partitioning is motivated by architectural constraints of the Arm 32-bit architecture, which does not support moving a full 32-bit immediate value into a register using a single instruction. Instead, Arm provides the `movw` and `movt` instructions to move the lower and upper 16 bits separately, enabling efficient initialization and manipulation of CIDs. This design supports up to 2^{16} distinct functions and 2^{16} indirect call target sets, which is sufficient for typical MCUs.

Each function's CID is embedded in the code memory at a fixed offset, located 4 bytes before the function entry point. For NSC veneers, however, the CID is placed immediately after the instruction that transfers control to the actual S function. Since code memory is immutable at runtime, these embedded CID values cannot be tampered with by an attacker. To securely reference CIDs at runtime, XCFI reserves the `R10` to hold the CID of the currently executing function.

```
1 ; disable interrupts
2 movw    r10, #expected_sid ; Load expected SID
3 ldr     rd2, [rd1, #-5]    ; Load target CID
4 uxth.w  rd2, rd2
5 subs.w  rd2, rd2, r10      ; Verify SID
6 bne.w   XCFI_fault         ; Fault on mismatch
7 ; enable interrupts
8 blx     rd1
```

Listing 1: Caller-side check for indirect calls. Since the indirect branch target pointer (`rd1`) includes the Thumb bit (LSB=1), the CID stored at offset −4 from the function entry is accessed using `[rd1, #-5]`.

4.2 Intra-State CFI

This section explains how XCFI enforces intra-state CFI using the CID defined in § 4.1, covering forward-edge, backward-edge, and exception handling within the same security state.

4.2.1 Forward-Edge CFI

Forward-edge CFI covers both indirect calls and indirect jumps. However, indirect jumps typically arise from compiler-generated jump tables and are generally confined to intra-function targets. Consequently, prior works [4, 26] have excluded indirect jumps from primary attack considerations. Accordingly, XCFI disables jump table generation using the `-fno-jump-tables` option and focuses on protecting indirect calls.

XCFI verifies indirect calls based on SID, which encodes the set of targets permitted by the CFI policy. Such verification can be performed at different points along a control-flow transfer and is categorized into *caller-side check* [1], and *callee-side check* [4, 22]. In a caller-side check, the caller verifies the transfer before executing an indirect call by comparing the SID of the runtime branch target against the expected target's SID; the transfer is allowed only if the two identifiers match. In contrast, a callee-side check performs verification after control transfer by comparing the caller-propagated SID with the SID of the authorized caller.

XCFI adopts a caller-side check for forward-edge CFI. This is because enforcing a callee-side check would require SID verification at callee entry points, which would either introduce SID checks along direct call paths or require separate entry points to bypass such checks for direct calls.

Listing 1 illustrates XCFI's forward-edge CFI with a caller-side check for an indirect call. The expected SID is loaded into the `R10` (ln 2). Using the branch target address held in the register, the corresponding CID is loaded from the address at an offset of −4 bytes (ln 3). The lower 16-bit SID is extracted from this CID and compared against the expected SID (ln 4–5). If the comparison fails, a fault is raised to signal a control-flow violation (ln 6). Otherwise, the indirect call is executed (ln 8). To ensure atomicity of the verification, XCFI

```
1  .word   0xdeadbeef          ; embedded CID value
2  bti
3  ldr.w   r10, [pc, #-0x8]    ; Load function CID
4  pacg    rd1, r10, lr        ; PAC for return address
5  push    {rd1, lr}
6  ...
7  pop     {rd1, lr}
8  movw    r10, #expected_sid ; Reload CID
9  movt    r10, #expected_uid
10 autg    rd1, r10, lr        ; Authenticate return
11 bx      lr
```

Listing 2: `PACG/AUTG` inserted into the function prologue and epilogue for backward-edge CFI.

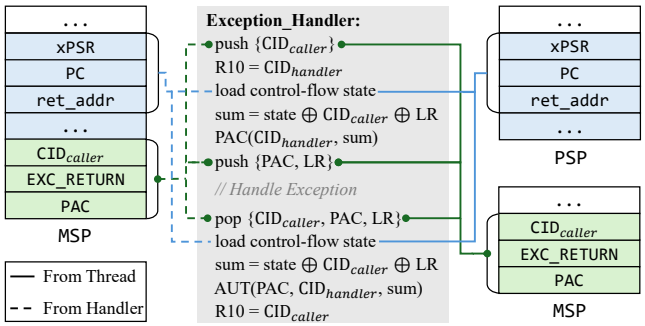


Figure 3: Exception handling protection in XCFI. The blue region denotes the exception stack, and the green region (marked with a circular arrow) denotes the exception handler stack.

temporarily disables interrupts during the check, preventing interleaving by asynchronous events (ln 1, 7).

4.2.2 Backward-Edge CFI

As described in § 2, conventional PAC-based backward-edge CFI signs return addresses using `SP`. This approach relies on `SP` to distinguish calling contexts. However, `SP` may take the same value across different call sites or functions when stack frames have identical sizes and alignment. As a result, a return address paired with its PAC in one function may authenticate in another function that uses the same `SP` value, enabling PAC reuse attacks. To prevent such reuse, XCFI uses the CID—rather than `SP`—as the PAC modifier. Because each CID embeds a function-unique UID, the resulting PAC is bound to a specific function and cannot be replayed without the matching CID.

Listing 2 shows the code used by XCFI to enforce backward-edge CFI. At function entry, the CID is loaded into `R10` (ln 3). A PAC for the return address is computed using `R10` and `LR` as modifiers and saved on the stack (ln 4–5). Upon function return, the return address and its associated PAC are restored from the stack (ln 7), and `AUT` is performed using the `R10` and `LR` (ln 10). If verification succeeds, the function returns normally (ln 11). Otherwise, verification fails, indicating tampering with the return address, the PAC, or the CID. When nested function calls occur, the CID stored in `R10` may be overwritten by callees. In this case, the current function CID is reloaded into `R10` immediately before performing `AUT` (ln 8–9). By doing so, XCFI avoids restoring the current function CID in `R10` after every nested call; instead, it reloads the CID immediately before `AUT`, reducing overhead while ensuring correct PAC-based authentication.

4.2.3 Exception Handling

Exceptions, including interrupts, may occur either synchronously or asynchronously during program execution. When an exception is taken, the hardware saves the exception context on the stack that was active at the time of the exception (referred to as the *exception stack*) and transfers control to the corresponding exception handler. The saved context includes control-flow state, such as the return address and program status. If this control-flow state is corrupted, execution may be diverted from the intended control-flow, leading to control-flow hijacking attacks.

To protect such control-flow state during exception handling, XCFI must first identify the exception stack. This is determined by examining the `EXC_RETURN` value stored in `LR`, which encodes the execution mode at the time the exception was taken. Based on this information, XCFI selects `PSP` for exceptions taken from thread mode and `MSP` for those taken from handler mode. As illustrated in Figure 3, when an exception is taken from thread mode, the exception stack resides in the processor stack, while the exception handler executes on a separate stack, referred to as the *handler stack*, which is allocated in the main stack. In contrast, when an exception is taken from handler mode, both the exception stack and the handler stack reside in the main stack.

Using the identified exception stack, XCFI applies PAC-based authentication to the exception control-flow state, as illustrated in Figure 3. Upon exception entry, the caller CID stored in `R10` is first saved on the handler stack, since the handler overwrites `R10` with its own CID during execution. XCFI then retrieves the control-flow state from the exception stack (`PSP` or `MSP`) and combines it with the preserved caller CID and the exception handler `LR` (`EXC_RETURN`) using an XOR operation to form a summary value. PAC is computed over this summary using the handler CID as the modifier and is stored together with `EXC_RETURN` on the handler stack. On exception return, XCFI recomputes the summary value and performs authentication using `AUT`. Any modification to the control-flow state, the caller CID, or `EXC_RETURN` results in authentication failure, as a single `AUT` verifies all these values after they are folded into a summary via XOR. Upon successful verification, the caller CID is restored to `R10`, ensuring that CFI is preserved across exception handling.

To ensure atomicity, XCFI briefly disables interrupts while

```
1  NSC_veneer:
2    sg
3    ; disable interrupts
4    ldr     rd1, [pc, #16] ; Load veneer cid
5    uxth.w  rd1, rd1
6    subs.w  rd1, rd1, r10  ; Verify SID
7    bne.w   XCFI_fault
8    ; enable interrupts
9    b.w     secure_func
10   .word   0xdeadbeef     ; Embedded cid value
```

Listing 3: Callee-side check inserted into the NSC veneer.

executing the exception protection procedure. Notably, interrupts are disabled only during the prologue and epilogue of the exception handler and are re-enabled during the handler body, allowing nested exceptions to occur.

4.3 Cross-State CFI

Unlike intra-state CFI, cross-state CFI must be designed under the constraints imposed by TZ-M. TZ-M enforces strict confidentiality of the S state; accordingly, code executing in the NS state must not be able to observe or infer S state code, execution context, CFI metadata, or return addresses. Under these constraints, this section describes how XCFI extends the intra-state CFI mechanisms introduced in § 4.2 to enforce CFI across security state boundaries.

4.3.1 Forward-edge CFI

Unlike intra-state forward-edge CFI, XCFI employs both caller-side and callee-side checks for cross-state forward-edge CFI. This is because cross-state function calls are accompanied by security state transitions, and the control-flow path differs depending on the direction of the transition.

When the S state calls an NS function, the indirect call is protected using the same caller-side check as intra-state forward-edge CFI. Before the indirect call is executed, XCFI loads the SID of the runtime branch target and verifies it against the expected SID. In contrast, *when the NS state calls an S function*, the call path differs fundamentally from that of an intra-state call. In TZ-M, an NS-to-S call does not branch directly to the target S function; instead, it targets an NSC veneer that mediates entry into the S state. Consequently, enforcing a caller-side check would require the NS caller to verify the SID of the target NSC veneer before indirect call. However, because NSC veneers reside in S memory, NS state cannot access their CID, rendering such verification fundamentally infeasible.

To address this, XCFI performs a callee-side check specifically for NS-to-S indirect calls. After control transfers from the NS state to the NSC veneer, the veneer verifies whether the calling NS function is authorized. Listing 3 illustrates the callee-side check performed within the NSC veneer. Prior to

Memory Map	NS Code	···	S Code
NS MPU Config.	NS MPU (RX)		
S MPU Config.			S MPU (RX)
TTT Resp. (NS) (SR/AP/MR)	0/non-zero/non-zero		0/0/0

Figure 4: Behavior of `TTT`. The `TTT` response fields indicate security-related (`SR`), access-permission (`AP`), and MPU-region (`MR`) attributes. under different states and MPU configurations.

```
1  ; disable interrupts
2  movw    r10, #expected_sid ; Load expected SID
3  ttt     rd2, rd1
4  cbz     secure             ; TTT==0 -> NS-to-S
5  ldr     rd2, [rd1, #-5]    ; Caller-side check
6  uxth.w  rd2, rd2
7  subs.w  rd2, rd2, r10
8  bne.w   XCFI_fault
9  secure:
10 ; enable interrupts
11   blx     rd1              ; Branch to NSC veneer
```

Listing 4: Determining the security state of a branch target in the NS state using the `TTT`.

the indirect call, the NS caller places the expected NSC veneer SID into `R10` and then executes the indirect call. After execution enters the S state via `SG` (ln 2), the NSC veneer loads its own CID and derives the corresponding SID (ln 4–5). The derived SID is compared against the expected SID propagated in `R10` (ln 6). A mismatch is treated as a control-flow violation and results in a fault (ln 7). If the verification succeeds, the actual S function is invoked (ln 9). To ensure atomicity of this verification, interrupts are temporarily disabled during the check (ln 3, 8).

Since callee-side checks are required only for NS-to-S indirect calls, XCFI must distinguish NS-to-S calls from NS-to-NS calls at each NS indirect call site. To this end, XCFI leverages the `TTT` together with the MPU-enforced access permissions. XCFI identifies NS-to-S indirect calls by checking whether `TT_RESP` is zero. As shown in Figure 4, NS code is mapped by the NS MPU, whereas S code—including NSC veneers—is not. When `TTT` is executed in the NS state, security-related attributes are always reported as zero, regardless of the queried address. For NS code addresses, however, the returned `TT_RESP` still contains non-zero MPU-related fields reflecting NS MPU configuration. In contrast, queries on S code match no MPU region, causing all MPU-related fields to be zero and yielding an all-zero `TT_RESP`.

Listing 4 illustrates the NS-side procedure used to distinguish NS-to-S indirect calls from NS-to-NS calls. Prior to the indirect call, the caller loads the expected target SID into `R10` (ln 2). `TTT` is then executed on the runtime branch target (ln 3). If `TTT` result is zero, the call is classified as an NS-to-S.

In this case, CID verification is skipped and the indirect call is executed (ln 9–11). Otherwise, the target is classified as NS function, and the standard caller-side check is applied before the branch (ln 5–8). On the S side, however, standard caller-side check (i.e., Listing 1) is sufficient, as branch target CIDs are directly accessible and no target classification is needed.

4.3.2 Backward-edge CFI

Unlike forward-edge CFI, backward-edge CFI verifies return addresses locally at the return point. Specifically, the return address is stored on the callee's stack and authenticated using PAC-based authentication immediately before control is transferred back to the caller, ensuring that it has not been tampered with during execution. Because this verification is performed entirely within the callee, backward-edge CFI does not require propagating verification metadata across security state boundaries and can be applied to cross-state returns. However, because cross-state returns involve hardware-managed security state transitions under asymmetric memory access permissions between the S and NS states, the mechanism used to verify return addresses depends on the direction of the security state transition.

When an S function returns to the NS state, it is handled in the same manner as an intra-state return: the callee preserves the caller's return address and transfers control directly to it. Accordingly, XCFI applies the same PAC-based authentication used for intra-state backward-edge CFI by verifying the return address before leaving the S state. In contrast, for returns from the NS state to the S state, the NS callee does not retain the actual return address to the S state in `LR` during NS execution, in order to preserve S state confidentiality. Instead, `LR` is set to a special value, `FNC_RETURN`, while the actual return address and execution context are stored exclusively on the S stack and restored by hardware upon return. As a result, the return address and execution context cannot be accessed or tampered with by the NS state. In this case, XCFI relies on these hardware guarantees and protects the `LR` value used in the NS state (i.e., `FNC_RETURN`) using PAC-based authentication.

4.3.3 Exception Handling

With TZ-M, exceptions may target a security state different from the one currently executing, and exception handling supports nesting across security states. Since S exceptions have the highest priority (§ 3), cross-state exceptions can arise in two scenarios: (i) when an S exception occurs during NS execution, and (ii) when an NS exception occurs while executing in S thread mode.

When an S exception occurs during NS execution, the hardware saves the exception context on the NS stack at exception entry and transitions execution to the S state. Because this context remains accessible from the S state, the S exception handler can directly access control-flow state saved on the NS stack. As a result, XCFI can apply the same protection mechanism as in intra-state exception handling. XCFI uses the `EXC_RETURN` value to identify the exception stack (`PSP_NS` or `MSP_NS`), collects the control-flow state from the exception context, and protects it using PAC-based authentication.

In contrast, *when an NS exception occurs while executing in S thread mode*, the NS exception handler cannot access the exception context saved on the S thread stack. Consequently, the S execution context and return address stored on the S stack are never exposed to NS code. In this case, the only control-flow state that must be protected is the `EXC_RETURN` value used to resume execution in the S state, which is protected by PAC-based authentication applied to the `LR`.

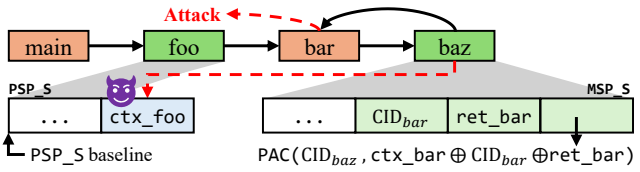


Figure 5: S state re-entry attack scenario. The red dashed path indicates an attacker-controlled control-flow sequence that exploits S state re-entry.

4.4 Secure State Re-entry with an Existing Upper-Level Secure Context

As discussed earlier, when control-flow transfers from the S state to the NS state, access to S memory is prohibited in the NS state. In a single S-to-NS transition—such as returning from the S state to the NS state or handling an NS exception during S execution—the S execution context remains inaccessible throughout NS execution. As a result, XCFI does not apply additional protection to the S execution context in these cases. However, this assumption does not hold when execution re-enters the S state before a previously invoked S function has returned. In such S state re-entry with an existing upper-level S context scenarios, the S thread stack that was isolated during NS execution becomes accessible again, exposing the previously saved S execution context.

Figure 5 illustrates a representative scenario that leads to S-state re-entry. When the NS function `main` invokes an S function `foo`, execution enters the S state, with `PSP_S` set to the baseline value established after boot. If an NS exception `bar` occurs while `foo` is executing, the execution context of `foo` (`ctx_foo`) is saved on the S thread stack, and execution transitions from the S state to the NS state. At this point, the saved context remains isolated, as the NS state cannot access S memory. However, if an S exception `baz` subsequently occurs during the handling of `bar`, execution re-enters the

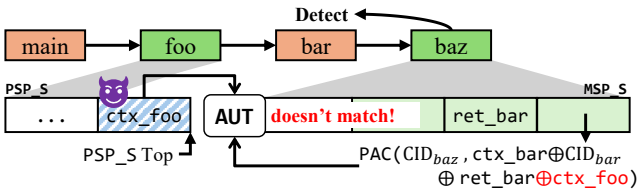


Figure 6: S state re-entry protection. The shaded `ctx_foo` indicates a tampered execution context modified by an attacker, which is detected by the PAC-based authentication.

S state while `ctx_foo` is still present on the S thread stack. As illustrated by the red dashed path in Figure 5, this S state re-entry causes the S stack to become accessible again during execution of `baz`, exposing the `ctx_foo` and allowing it to be tampered with, potentially leading to control-flow hijacking attacks.

Under the XCFI execution model (§ 3), this scenario can be generalized into a limited set of control-flow sequences that permit S state re-entry:

- NS thread → S thread → **NS handler** → **S handler**
- NS thread → S thread → **NS thread** → **S handler**
- NS thread → S thread → **NS thread** → **S thread**

Because the S state functions as a secure service provider, execution in the S state always originates from the NS state. Furthermore, since S exceptions have higher priority than NS exceptions, S state re-entry can occur only when the upper-level S execution is an S thread.

Based on these observations, the presence of S state re-entry can be determined by examining the state of the S thread stack relative to a `PSP_S` baseline. The definition of this baseline depends on the execution model of the S state. With a single S thread stack, the `PSP_S` value immediately after boot defines the baseline, whereas in partitioned S execution environments [53], where each secure partition maintains its own S thread stack, the initial `PSP_S` of each partition serves as baseline.

Under this baseline definition, when re-entering the S state after an S-to-NS transition, the execution context of the previous S execution remains on the S thread stack, causing `PSP_S` to point to a lower address than the baseline. XCFI leverages this deviation to detect S state re-entry. This detection procedure is applied only at S entry points reachable from the NS state: S function entries via NSC veneers and S exception handlers. Because all NS-to-S control-flow transfers occur only at this limited and statically identifiable set of entry points, it is sufficient to restrict detection and protection to these sites.

When a S state re-entry is detected, XCFI protects the execution context of the upper-level S function using a PAC-based authentication. Specifically, XCFI extends the current summary value by incorporating the upper-level S execution context, and computes a PAC over the summary. The resulting PAC is authenticated before control returns to the S execution. As illustrated in Figure 6, when function `baz` is identified as a S state re-entry point, XCFI constructs the summary by XOR-combining `ctx_foo`. Because the upper-level S context resides at the top of the S thread stack, immediately below the current `PSP_S`, the protection targets can be accessed directly without stack traversal. XCFI then computes a PAC over this summary and stores it on the `baz` stack. Any modification to `ctx_foo` during the execution of `baz` causes PAC authentication to fail before returning to `bar`, thereby preventing control-flow hijacking attacks.

In contrast, NS state re-entry with existing upper-level NS context (e.g., the `main`→`foo`→`bar` shown in Figure 6) does not require additional protection, as the execution context of the upper-level NS function is already protected by the S function. Even if these values are modified during subsequent NS execution, any tampering is detected when control returns to the S state, where the protected NS control-flow state is verified upon S function return.

5 Implementation

We implemented a prototype of XCFI on an MPS3 board (AN555 FPGA image) featuring an Armv8.1-M–based Cortex-M85 core. The prototype is built using LLVM 21 [33], Zephyr v4.3 [49], and TF-M v2.2 [53] to configure the TZ-M environment. The core protection mechanisms of XCFI are implemented via compile-time instrumentation. We extend the LLVM backend with a custom `MachineFunction` pass that inserts caller-side CID check before indirect calls and instruments `PACG/AUTG` instructions in function prologues and epilogues. In addition, we modify the LLD linker to emit a callee-side CID check when generating `SG` veneers for cross-state transitions. We use the modified toolchain to compile both S and NS software, including the runtime libraries and the RTOS kernel. Also, XCFI identifies NS-to-S entry points at compile time. In TZ-M systems, these entry points correspond to S functions annotated with `__attribute__((cmse_nonsecure_entry))` [9] and S exception handlers.

XCFI adopts the SID generation scheme of FineIBT [22], enabling support for fine-grained forward-edge CFI policies such as arity-based [54], type-based [52], and MLTA-based [34] policies. In the current prototype, SIDs are generated at the LLVM IR level by hashing function signatures derived from the number and types of arguments, as well as the return type. Additionally, UIDs are computed as hash values of function names. Furthermore, CID values whose bit patterns coincide with the encodings of landing-pad instructions (i.e., `BTI` or `SG`) are regenerated to prevent unintended instruction interpretation.

6 Evaluation

In this section, we evaluate the execution time and flash memory overhead of XCFI. We measure the cost of XCFI's protection primitives in isolation and evaluate interrupt latency using Zephyr's `latency_measure` benchmark [50]. We use the BEEBS [42] and CoreMark-Pro [18] benchmarks and break down the overhead into its forward- and backward-edge CFI components. We further compare XCFI against existing hardware-based CFI mechanisms, including PACBTI and Silhouette [62]. We also evaluate the real-time impact of XCFI using TACLeBench's Papabench workload [19] and Open SyringePump [40], both integrated into Zephyr. Finally, we demonstrate the applicability of XCFI in a TZ-M environment using Zephyr's `tfm_integration` examples [51].

6.1 Experimental Setup

We evaluate XCFI with two comparison targets: PACBTI and Silhouette [62]. PACBTI is enabled using `-mbranch-protection=bti+pac-ret`, which provides BTI-based forward-edge CFI and PAC-based backward-edge CFI with `SP` and `LR` as modifiers. Silhouette is selected because it is the most recent open-source compiler-based CFI mechanism for embedded systems and operates independently of any specific RTOS.

For a fair comparison, we port Silhouette from its original bare-metal implementation to the same Zephyr environment as XCFI, place the shadow stack at a fixed 2MB offset in SRAM, and apply protection uniformly to both application and library code. Execution-time overhead is measured in processor cycles using the DWT cycle count [8], and flash memory overhead is measured using `llvm-size` [33]. Following prior work [48,62], we evaluate the 20 longest-running BEEBS benchmarks to reduce sensitivity to compiler optimizations.

We disable interrupts during benchmark execution because the benchmarks run for short durations and involve minimal interrupt handling. In addition, the benchmark examples do not involve security state transitions; therefore, we focus exclusively on evaluating intra-state CFI. In contrast, the real-time evaluation enables interrupts to capture interrupt-driven execution under timing constraints, while the TF-M evaluation includes security state transitions and exception handling. This configuration allows us to evaluate intra-state timing behavior and cross-state CFI enforcement under practical conditions.

6.2 Microbenchmarks

6.2.1 Primitive Costs

Table 1 reports the cycle cost of XCFI's protection primitives measured in isolation. Overall, each primitive costs only a few tens of cycles, as XCFI's checks are implemented using short instruction sequences for CID checks and PAC-based authentication.

Primitive	Cycles
`SG`-based transition	17
Indirect call path in S state	17
Indirect call path in NS state (NS-to-S)	23
Indirect call path in NS state (NS-to-NS)	18
Return protection (pro. / epi.)	15 / 5
Exception protection (pro. / epi.)	28 / 30
Exception security-state identification (pro. / epi.)	10 / 28

Table 1: Cycle cost of XCFI protection primitives.

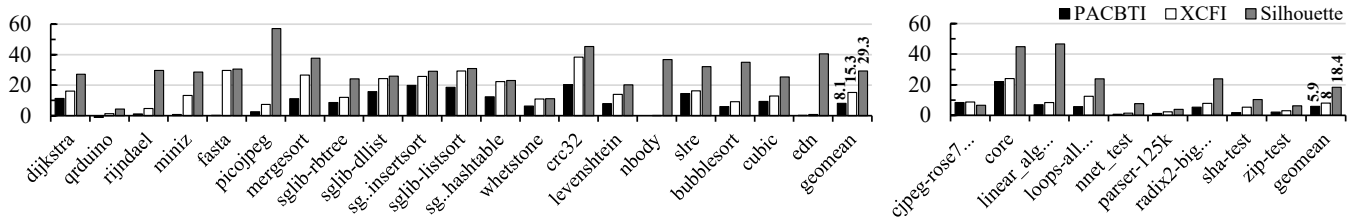
The `SG`-based transition cost measures the NSC-veneer path after `SG`, including callee-side CID check. Indirect call paths are separated because the measured path differs depending on the security states of the caller and callee. The S-state path performs the standard caller-side CID check, while the NS-state path perform `TTT`-based target identification before either branching to an NSC veneer for NS-to-S calls or performing a caller-side CID check for NS-to-NS calls. For returns and exceptions, prologue and epilogue costs are separated because protection is split between entry-time PAC computation and exit-time authentication. The exception security-state identification cost is incurred only for cross-state exception handling, where XCFI identifies whether the saved exception context is associated with the S or NS state.

6.2.2 Interrupt Latency

XCFI briefly disables interrupts to ensure the atomicity of inserted protection code, such as forward-edge CID checks and exception-handler prologue/epilogue instrumentation. As shown in Table 1, interrupt masking is confined to primitive-level protection sequences rather than the entire handler execution. The impact of this interrupt masking is measured as the latency from interrupt handling to thread resumption. With XCFI, the return-to-original-thread case increases from 634 to 669 cycles, while the return-to-different-thread case increases from 863 to 1003 cycles. These results indicate that the interrupt-latency impact of XCFI remains limited.

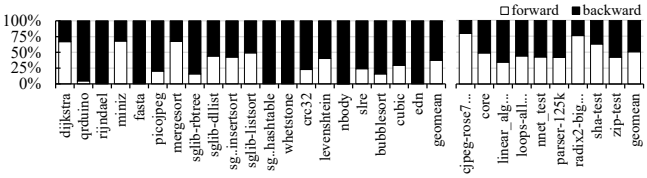
6.3 Performance Overhead

We evaluate the performance overhead of the CFI mechanism introduced by XCFI. Figure 7 shows execution-time results for BEEBS and CoreMark-Pro. To analyze the components of execution-time overhead, Figure 8 decomposes XCFI's overhead into forward-edge and backward-edge components. This breakdown shows that backward-edge CFI dominates overall overhead because PAC-based authentication is applied at nearly every function call, while CID checks are triggered only on indirect calls. Across BEEBS and CoreMark-Pro, indi-



(a) BEEBS (b) CoreMark-Pro

Figure 7: Performance overhead.



(a) BEEBS (b) CoreMark-Pro

Figure 8: Performance overhead breakdown.

rect calls account for 0%–83% of all function calls, depending on the workload. Therefore, the overhead breakdown should be interpreted together with event frequency: workloads with frequent function calls tend to be dominated by backward-edge CFI, whereas workloads with a high indirect-call ratio exhibit a larger forward-edge contribution.

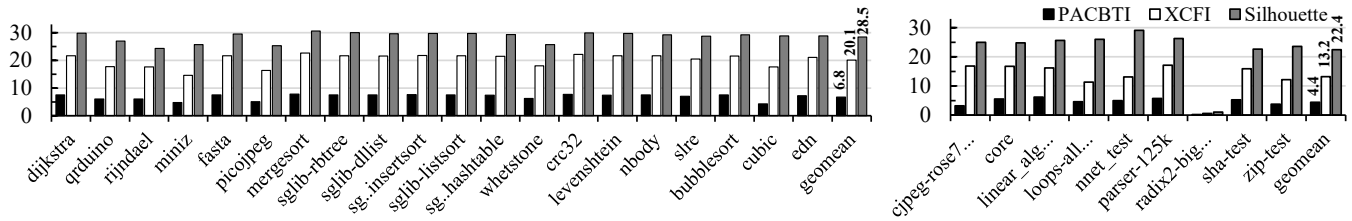
In the BEEBS benchmark suite, XCFI exhibits the highest execution-time overhead in `crc32`, reaching 38.4%. This overhead stems from the large number of function calls in `crc32` (2,049 per execution), which significantly increases the frequency of return address authentication. Compared to `levenshtein`, which has a similar execution time, `crc32` executes approximately 100× more function calls. Because the core computation of `crc32` itself is relatively short, the fixed cost introduced by XCFI's CFI instrumentation accounts for a larger fraction of the total runtime. `mergesort` incurs an overhead of 26.7% and shows the highest relative contribution from forward-edge CFI, as 82.7% of its function calls are indirect and frequently trigger CID check. Similar behavior is observed in CoreMark-Pro: `core` incurs a 24% overhead due to a large number of function calls, while in `cjpeg-rose7-preset`, forward-edge CFI contributes more significantly due to the high fraction of indirect calls (54.2%).

We next compare XCFI with PACBTI and Silhouette. Across both benchmarks, execution-time overhead generally increases in the order of PACBTI, XCFI, and Silhouette. Since XCFI builds on PACBTI, the additional overhead introduced by XCFI is modest, amounting to 7.2% on BEEBS and 2.1% on CoreMark-Pro. Silhouette incurs higher overhead than reported in its original evaluation because protection is applied to both application and library code, and Zephyr's kernel replaces bare-metal implementations with kernel- and library-mediated execution paths that involve frequent internal function calls (e.g., *malloc*), amplifying the overhead of shadow stack updates and store instrumentation.

Although XCFI outperforms Silhouette overall, Silhouette exhibits slightly lower overhead in workloads dominated by indirect calls (e.g., `cjpeg-rose7-preset`). This is because Silhouette's coarse-grained label-based checks incur lower overhead than XCFI, which performs additional CID loading and verification to provide fine-grained protection. Conversely, XCFI incurs lower backward-edge overhead by relying on lightweight PAC-based authentication, whereas Silhouette incurs additional costs from shadow stack maintenance and store hardening. This contrast is evident in `nbody`. XCFI incurs negligible overhead because most calls neither create stack frames nor involve indirect calls, allowing most instrumentation to be bypassed. In contrast, Silhouette exhibits substantially higher overhead due to store hardening, which expands single store-multiple instructions into multiple unprivileged store operations.

6.4 Flash Memory Overhead

In memory-constrained MCUs, flash memory overhead is as important as execution time. Figure 9 presents the flash memory overhead for BEEBS and CoreMark-Pro. In XCFI, most of the flash overhead originates from backward-edge CFI, as `PACG/AUTG` instructions are inserted at nearly every function. Compared to PACBTI, the additional code introduced by XCFI results in flash memory overheads of 13.7% on BEEBS and 8.8% on CoreMark-Pro. In contrast, XCFI exhibits lower geometric-mean flash memory overhead than Silhouette, by 8.0% on BEEBS and 9.2% on CoreMark-Pro. Across PACBTI, XCFI, and Silhouette, `radix2-big-64k` consistently exhibits notably low flash overhead, as shown in Figure 9b, because a large portion of its flash footprint consists of read-only data that is unaffected by instrumentation. Overall, XCFI provides effective control-flow protection with flash memory overheads that are comparable to or smaller than those of the evaluated baselines, making it suitable for memory-constrained MCUs.



(a) BEEBS (b) CoreMark-Pro

Figure 9: Flash memory overhead.

App	Task	Metric	Baseline	XCFI	Deadline
Papabench	Stabil	AVG	35 μs	38 μs	50 ms
		WCET	39 μs	42 μs	50 ms
	Report	AVG	21 μs	34 μs	100 ms
		WCET	61 μs	88 μs	100 ms
	Navi	AVG	28 μs	37 μs	250 ms
		WCET	47 μs	53 μs	250 ms
SyringePump	Injec.	Runtime	499 ms	507 ms	2000 ms

Table 2: Real-time impact analysis. Papabench reports per-task average and WCET, while SyringePump reports end-to-end injection runtime.

	Non-secure (NS)			Secure (S)			Transition / Exception		
	TOT	IC	RET	TOT	IC	RET	SG	SysTick	PendSV
attestation	12347	9197	8731	98	14	62	3	7	3
crypto	13236	10079	9451	609	106	393	18	1	18
certificate	10428	8522	7767	611	106	395	18	7	18
ipc	3583	2749	1741	56	7	33	3	0	1
secure_partition	1903	1281	1041	498	36	276	18	1	18

Table 3: Instruction counts for TF-M workloads, grouped by execution domain. `SG` denotes NS-to-S transitions, while `SysTick` and `PendSV` denote exceptions from the NS and S states, respectively.

6.5 Real-time Impact

To evaluate the impact of XCFI on real-time workloads, we evaluate whether the instrumented workloads satisfy their timing constraints. Papabench models embedded control software for a UAV and consists of periodic tasks such as stabilisation, reporting, and navigation. Since these tasks have different periods and deadlines, they are suitable for evaluating task-level execution time and WCET. SyringePump represents an end-to-end medicine injection workload and has been used in prior works [2, 56].

Table 2 shows the real-time impact analysis. In Papabench, XCFI slightly increases both average execution time and WCET, while all tasks remain far below their deadlines. In SyringePump, the end-to-end injection runtime also remains below the deadline after applying XCFI. Across the evaluated workloads, the measured execution-time increase ranges from 3 μs to 8 ms, which remains small relative to the corresponding deadlines. These results show that XCFI does not introduce deadline violations in the evaluated real-time workloads.

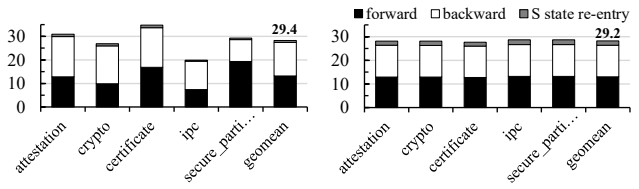
6.6 Applicability in TrustZone-M

To evaluate XCFI in realistic TZ-M environments, we use Zephyr's `tfm_integration` examples: `psa_crypto`, `tfm_ipc`, and `tfm_secure_partition`. `psa_crypto` exercises attestation, cryptographic operations, and certificate handling in the S state via the PSA Crypto API, while `tfm_ipc` and `tfm_secure_partition` evaluate IPC-based secure API invocations and user-defined secure partition execution.

Table 3 summarizes the control-flow events observed in these workloads, including indirect calls, returns, security state transitions, and exceptions. The TF-M examples generate frequent security state transitions and exceptions, exercising both thread and handler modes. They also involve numerous intra-state indirect calls and returns. Notably, all NS-to-S transitions occur via indirect calls through NSC entries.

Figure 10a presents the runtime overhead for the TF-M examples. Compared to the results in § 6.3, TF-M examples exhibit a larger proportion of overhead from forward-edge CFI due to the high fraction of indirect calls, which account for approximately 75% of all function calls. The highest overhead is observed in `psa_crypto (certificate)` (34.8%). In this example, approximately 73.9% of functions create stack frames, triggering PAC-based authentication, while 78.2% of function calls are indirect calls, resulting in substantial CID checks. The combination of frequent returns and indirect calls leads to significant overhead from both CFI components.

The overhead associated with S state re-entry handling is small and has a negligible impact on overall performance. S state re-entry detection procedure incurs 1,046 cycles per execution, which translates to a geometric mean overhead of approximately 0.9% across the evaluated TF-M examples. In our experiments, none of the TF-M examples triggered actual S state re-entry; therefore, the measured overhead reflects only the cost of detection. When S state re-entry does occur, protecting the upper-level S execution context introduces an additional cost of 188 cycles in both the prologue and the



(a) Performance overhead. (b) Flash memory overhead.

Figure 10: TF-M evaluation results.

epilogue, which is small compared to the detection overhead and is unlikely to significantly affect overall performance.

Figure 10b shows the flash memory overhead introduced by XCFI for the TF-M examples. The overhead from forward-edge and backward-edge CFI follows trends similar to those in § 6.4. Additional code for S state re-entry handling is inserted only at S entry points reachable from the NS state, namely NSC veneers and S exception handlers. As a result, the flash memory overhead attributable to S state re-entry handling remains small, at 1.8%.

Overall, these results demonstrate that XCFI operates effectively in TZ-M environments with diverse control-flow events across the S and NS states, while maintaining practical performance and flash memory overhead.

7 Security Evaluation

This section evaluates the security of XCFI from two perspectives. First, § 7.1 presents an end-to-end evaluation using a `ret2ns` attack [37, 38], showing that XCFI detects practical cross-state control-flow hijacking attacks against TZ-M security state transitions. Second, § 7.2 analyzes residual attack attempts against XCFI's CFI enforcement, including CID manipulation, abuse of target classification, BTI bypass attempts, and PAC bypass attempts.

7.1 End-to-End Attack Evaluation

To evaluate whether XCFI can defend against real cross-state control-flow hijacking attacks, we port the public `ret2ns` attack artifact [36] to our evaluation environment and perform an end-to-end attack evaluation. `ret2ns` exploits control-flow-critical state saved during TZ-M security state transitions to divert execution from the normal cross-state return or call path to attacker-controlled code in the NS state. Although `ret2ns` is typically discussed in privilege-separated settings, its core primitive is cross-state control-flow hijacking through corrupted return or indirect-call targets, which XCFI blocks independently of the privilege model.

The ported `ret2ns` attacks are classified into two types. The first is a `BXNS`-based backward-edge attack, which corrupts the saved return address and execution context used when an S function returns to the NS state, causing execution to resume at an attacker-controlled NS function. The second is a `BLXNS`-based forward-edge attack, which corrupts a function pointer used inside an S function so that a subsequent S-to-NS indirect call invokes an attacker-controlled NS function. The artifact also includes privileged and unprivileged NS variants. Since both variants share the same control-flow hijacking primitives, and the unprivileged variant invokes the same S function through `SVC`, the evaluation uses the privileged execution environment assumed by XCFI.

In the `BXNS`-based backward-edge attack, after `_driver_LCD_print()` is executed inside the S function, a buffer overflow corrupts the saved return context so that the S function returns to the NS `attacker_controlled()` function. If the attack succeeds, control is transferred to attacker-controlled code instead of returning to the intended NS caller. However, XCFI applies PAC-based backward-edge CFI to S-to-NS returns. The corrupted return context cannot be authenticated with a valid PAC, causing the `AUT` verification before return to fail and raise a fault. As a result, control is not transferred to the attacker-controlled NS function.

In the `BLXNS`-based forward-edge attack, a function pointer `check_fp`, stored as a stack object inside the S function, is overwritten by a buffer overflow with the address of the NS `attacker_controlled()` function. When the S function later invokes an NS function through this function pointer, the attacker attempts to execute the attacker-controlled function instead of the intended NS function. XCFI inserts a forward-edge SID check before executing the `BLXNS` instruction, which compares the SID of the runtime branch target with the expected SID at the callsite. Since the SID of the corrupted target does not match the expected SID, XCFI detects the mismatch as a control-flow violation. Consequently, the forward-edge `ret2ns` attack is blocked before control is transferred to the attacker-controlled function.

These results demonstrate that XCFI detects real cross-state control-flow hijacking attacks implemented in the `ret2ns` artifact in an end-to-end setting. Both representative `ret2ns` attacks are detected before control is transferred to attacker-controlled NS code, showing that XCFI remains effective against practical attacks that exploit TZ-M security state transitions.

7.2 Security Analysis

This section describes attack scenarios that may be attempted even in the presence of XCFI's CFI enforcement, and analyzes how CFI is preserved under each scenario.

7.2.1 Integrity of CID

CID Tampering Attacks. An attacker who can read the code memory may learn the CID values assigned to functions and attempt to supply forged or mismatched CID values dur-

ing CID checks or PAC-based authentication. We consider the following attack vectors and explain why they are prevented in XCFI:

- **Register-Level CID Manipulation.** An attacker may attempt to bypass CID checks or PAC-based authentication by corrupting the CID value supplied through operand registers. However, such attacks are ineffective in XCFI. The CID used for verification and authentication is always conveyed via a dedicated register (`R10`), which is reserved exclusively for this purpose. Moreover, the CID is loaded into `R10` only via immediate operands or PC-relative loads from immutable code memory, leaving no execution path through which attacker-controlled data can influence the CID.
- **Exception-Induced CID Tampering.** An attacker may attempt to overwrite the caller CID temporarily saved during exception handling. In XCFI, the saved CID is part of a summarized control-flow state protected by PAC-based authentication; any modification to the saved CID causes authentication failure during `AUT` and is detected before control-flow resumes.

Unintended Instruction Execution Attacks. An attacker may attempt to redirect an indirect branch to a code address containing a CID value, aiming to have it decoded and executed as an instruction. Such attacks are blocked in XCFI at two stages. First, the redirected branch is rejected by the CID check, as the CID value fetched from the unintended target does not match the expected CID for the call site. Even if this check were bypassed by coincidence, the attack would still fail due to BTI enforcement. As described in § 5, CID values are never encoded to match landing pad instructions. Because all indirect branches in XCFI must target valid landing pad instructions, any redirection to a CID triggers a BTI violation when the processor attempts to execute a non-landing-pad instruction.

7.2.2 Robustness of XCFI Enforcement

Attempts to Abuse the `TTT`. XCFI distinguishes NS-to-S indirect calls from NS-to-NS by interpreting an all-zero `TT_RESP` returned by `TTT` as indicating a call to an NSC entry. For such calls, callee-side checks are performed within the NSC entry. An attacker may attempt to exploit this behavior by redirecting an indirect call to a non-NSC target that nevertheless yields an all-zero `TT_RESP`. This could cause the call to be misclassified as an NSC entry, thereby bypassing the caller-side check. Since the target is not an actual NSC entry, no callee-side check would be executed at the destination. However, such attempts are prevented as follows:

- **Branches to non-NSC S Memory.** An attacker may attempt to redirect an indirect branch to non-NSC S memory, since executing `TTT` on such addresses from the NS state yields an all-zero `TT_RESP`. This may cause the transfer to be misclassified as an NS-to-S transition and skip the caller-side check. However, such redirection cannot result in a valid NS-to-S transition. In TZ-M, transitions from the NS state to the S state are permitted only via `SG` located in NSC regions. An `SG` located elsewhere in S memory—whether in code or SRAM—cannot trigger a state transition when executed from the NS state. Consequently, an indirect branch targeting S memory outside the NSC region attempts execution while remaining in the NS state and immediately triggers a security violation, preventing any bypass of CID checks.
- **Branches to NS Memory Not Covered by the NS MPU.** An attacker may attempt to redirect an indirect branch to NS memory that is not covered by any NS MPU regions, since executing the `TTT` on such addresses may yield an all-zero `TT_RESP`. However, such memory is not executable. Only NS code memory is explicitly marked as executable and mapped by the NS MPU region, while all other NS memories—including SRAM and read-only data—are non-executable. Consequently, any indirect branch targeting NS memory outside executable code memory results in a fault and is blocked before any instruction can be executed.

Attempts to Bypass BTI. An attacker may attempt to redirect an indirect branch to a target that is not a valid `BTI` landing pad and evade the resulting BTI violation by exploiting interrupt timing. Specifically, the attacker may redirect an indirect branch to a non-`BTI` target and exploit a narrow edge case in which the caller-side CID check coincidentally succeeds. Before the first instruction at the target executes, the attacker then deliberately triggers an interrupt to clear the `EPSR.B` bit during exception handling. In XCFI, however, the `EPSR.B` bit is treated as part of the control-flow state and is protected during exception handling using PAC-based authentication. Any attempt to tamper with this bit via interrupt-induced interference is therefore detected during the `AUT` upon exception return, before control flow can resume.

Attempts to Bypass PAC. An attacker may attempt to bypass backward-edge CFI by triggering interrupts to interfere with PAC computation or authentication, for example, by corrupting the PAC modifiers `R10` or `LR`. Although XCFI does not disable interrupts during PAC computation and authentication, such attacks are ineffective. As described in § 4.2.3, these registers are included in the protected summary value, and any interrupt-induced modification is detected by `AUT` upon return. Consequently, backward-edge CFI remains correct even in the presence of interrupts.

8 Related Works

8.1 Control-Flow Integrity on Microcontrollers

Numerous CFI mechanisms have been proposed for MCUs [3, 15, 17, 27, 41, 48, 55, 57, 59, 62]. However, existing approaches suffer from fundamental limitations: they protect only a subset of control-flow events, incur substantial runtime overhead, or rely on the S state as part of the TCB.

Several prior works [15, 17, 27, 55, 62] primarily protect backward-edges using shadow stacks that store and validate return addresses in protected memory, while offering limited support for forward-edge control flow. μRAI [3] follows an alternative backward-edge protection approach by resolving returns via code-embedded direct jump targets selected at runtime based on an encoded execution context. While effective for backward-edge protection, both approaches provide no fine-grained guarantees for forward-edge control-flow transfers.

Other works [48, 57] achieve fine-grained protection for both forward- and backward-edges by recording control-flow events using the Memory Trace Buffer (MTB) and validating them against a predefined Control-Flow Graph (CFG). However, this design incurs high runtime overhead due to frequent security state transitions.

Several TZ-M–based mechanisms [27, 41, 48, 57, 59] further assume the S state as part of the TCB. CFI CaRE [41], TZmCFI [27], and CEST [59] place shadow stacks in the S state, while Sherloc [48] and InsectACIDE [57] store and verify control-flow information within S memory, leaving S-state execution outside the protection scope.

In contrast, XCFI provides comprehensive protection for forward-edge, backward-edge, and exception handling with modest runtime overhead. Moreover, XCFI enforces CFI in both the NS and S states, as well as cross-state control-flow transfers, addressing S state execution and security state transitions that are largely unprotected in prior studies.

8.2 Hardware-based Control-Flow Protection

Prior to XCFI, many studies leveraged hardware support for control-flow protection, including PAC, BTI, and related hardware features, primarily targeting general-purpose processors and focusing on user applications or operating system kernels.

Several works [4, 25, 31, 32, 58, 60, 61] employ PAC and BTI, introduced in the Armv8.5-A architecture [7], to protect control flow. Both mechanisms rely on execution context to distinguish valid control-flow transfers, and insufficient context differentiation can lead to PAC reuse or coarse-grained protection. To mitigate this, PAC-based approaches [25, 31, 32, 58, 60] incorporate function identities or call-path information into the PAC modifier, such as chaining caller context [32] or hashing function names [60]. BTI-based approaches similarly introduce context for forward-edge CFI; for example, CFA+ [4] assigns equivalence classes and identifiers to indirect calls. Architecturally distinct from Arm's BTI, FineIBT [22] leverages Intel's Indirect Branch Tracking [24] and CFG-based SIDs to achieve fine-grained forward-edge CFI. In XCFI, both PAC and BTI are leveraged, and a single 32-bit immediate identifier, CID, is introduced as a unified control-flow context for both forward- and backward-edge CFI.

In parallel, PAC has also been applied to Armv8.1-M architecture-based systems [11, 16]. Unlike CFI, these works focus on control-flow attestation, using PAC to generate post-execution evidence. Consequently, violations are detected only after execution, making immediate detection at the point of occurrence impossible, and TZ-M is used primarily to manage attestation data rather than to protect S state code or cross-state control-flow transfers. In contrast, XCFI enables immediate detection of control-flow violations at runtime and treats the TZ-M environment itself as a protection target rather than part of the TCB.

8.3 TrustZone-M Protections

Several studies have proposed defenses against attacks in TZ-M environments [30, 35, 38]. MICROFT [37, 38] analyzes vulnerabilities arising during the security state transitions and introduces the `ret2ns` attack. This attack exploits a legitimate `SG`-based entry into the S state from NS handler mode and tampers with the return address to resume execution in the NS state with escalated privileges. To mitigate this attack, MICROFT proposes MPU-based address validation and Software Fault Isolation (SFI) to validate the range of return addresses during S-to-NS transitions. In contrast, XCFI verifies the integrity of the execution context using PAC-based authentication during S-to-NS returns (§ 4.3.2), providing direct protection against return address tampering.

Other works focus primarily on compartmentalization or address space layout randomization (ASLR) rather than enforcing CFI. TZ-DATASHIELD [30] enforces compartmentalization in TZ-M to reduce the attack surface of the shared MCU address space and protect the confidentiality and integrity of sensitive data. While it additionally employs CFI for shared data and peripheral accesses, the provided protection is coarse-grained. iASLR [35] applies ASLR to increase the difficulty of attack within the S state, without enforcing CFI. Overall, existing TZ-M defenses emphasize randomization or compartmentalization, whereas XCFI directly enforces CFI across security states.

9 Discussion

Non-local Control Flow and Corner Cases. Non-local control-flow transfers, such as `setjmp/longjmp` and task-level context switching, redirect execution by saving and later

restoring execution context rather than following conventional call/return semantics. For `setjmp/longjmp`, the transfer target is determined by the context saved in `jmp_buf`, not by a return address on the call stack. Similarly, task-level context switching resumes execution by restoring scheduler-managed contexts, such as task control blocks or context frames. Although these mechanisms modify the execution flow, they do not involve security state transitions, and the preserved context is stored and managed in separate data structures outside the call stack. Protecting these contexts is orthogonal to XCFI and can be achieved using existing mechanisms, such as MPU-based isolation [17, 21, 63] or DWT-based monitoring [15, 43, 46, 47], to prevent unintended modification by other tasks.

Other corner cases depend on how the compiler and runtime implement the control flow. Stack unwinding is mainly associated with C++ exception handling, which is not used in the evaluated workloads. Tail-call optimization does not create an additional return edge when lowered as a branch; direct tail calls transfer control to statically determined targets, while indirect tail calls are protected by XCFI's forward-edge CFI as ordinary indirect branches.

Extending XCFI to Privilege-Separated Systems. XCFI assumes privileged-only execution, consistent with execution models commonly adopted in MCU-based RTOSs [12, 20, 49]. Nevertheless, its core design can be applied with minimal changes in environments that employ privilege separation. While the `TTT` instruction is unavailable in unprivileged execution, this limitation can be addressed by exploiting the fact that S and NS code reside in disjoint address ranges and replacing `TTT`-based checks with simple range checks on branch targets. This allows XCFI to be naturally extended to privilege-separated systems.

When NS exceptions can preempt S exceptions. The current prototype of XCFI assumes that S exceptions have the highest priority. If NS exceptions are configured with equal or higher priority, an NS exception may preempt an S handler, resulting in S state re-entry of the form S handler $\rightarrow$ NS handler $\rightarrow$ S handler. Although this execution model is not directly supported, XCFI can be extended to handle it. The `EXC_RETURN` value in `LR` indicates whether the current S exception originated from the NS state, and the presence of an unreturned upper-level S context can be detected by checking whether the S main stack pointer (`MSP_S`) has moved beyond its initial base address. Upon detecting such a S state re-entry, XCFI applies PAC-based authentication to the upper-level S context as described in § 4.4.

Deployment Considerations. XCFI is implemented in the compiler and linker and requires recompilation of both S and NS software, but does not require source-code changes. The current prototype is based on LLVM; supporting GCC would require corresponding compiler and linker support. XCFI is not tied to a specific RTOS or TF-M implementation, as it relies on architectural mechanisms standardized in Armv8-M, including TZ-M, MPU, and `TT` instructions. While Armv8.1-M PACBTI provides an efficient implementation substrate, the protection scheme of XCFI is not inherently tied to PACBTI. On platforms without PACBTI, the same enforcement strategy can be adapted in software by replacing PAC-based authentication and BTI-based landing-pad enforcement with XOR-based authentication [4] and label-based forward-edge CFI [15, 62], respectively, at the cost of additional overhead.

Applicability to Cortex-A. The Cortex-A architecture [7] also provides PAC and BTI, making XCFI applicable to TZ-A systems for intra-state control-flow protection. In this setting, CID-based verification can be combined with PACBTI following the same principles as in TZ-M. However, TZ-A handles security state transitions through secure monitor calls rather than direct cross-state function calls. As a result, the cross-state CFI mechanisms proposed in XCFI, which rely on explicit call and return paths across security states, cannot be directly applied in TZ-A environments.

10 Conclusion

In this paper, we presented XCFI, a comprehensive CFI mechanism for Arm TZ-M–based MCUs. XCFI leverages the PACBTI features of Armv8.1-M and introduces a unified CID to enforce fine-grained CFI across both S and NS states. This design provides uniform protection for indirect calls, function returns, exceptions, and cross-state control-flow events that are not addressed by prior work. By extending CFI enforcement to security state transitions, including S state re-entry, XCFI fills a critical gap in existing embedded CFI approaches. Our implementation and evaluation on a TZ-M–enabled MCU demonstrate that XCFI achieves comprehensive control-flow protection with practical performance overhead. Furthermore, evaluation using Zephyr's TF-M examples shows that XCFI maintains stable and efficient execution under realistic examples involving frequent security state transitions and exceptions.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and valuable comments that helped improve this paper. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2026-25484187).

Ethical Considerations

This work focuses on defensive control-flow integrity mechanisms for embedded systems. The study does not involve human subjects, personal data, or production systems. All experiments are conducted on open-source software and publicly available development platforms. The techniques presented in this paper are defensive in nature and do not include exploit code. We therefore believe that this work does not raise ethical concerns.

Open Science

We comply with the open science policy by releasing the XCFI prototype, including a modified LLVM compiler, Zephyr RTOS, and TF-M. The artifacts are released as open source and can be accessed at: https://doi.org/10.5281/zenodo.20195744

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[2] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 743–754, 2016.

[3] Naif Saleh Almakhdhub, Abraham A Clements, Saurabh Bagchi, and Mathias Payer. μrai: Securing embedded systems with return address integrity. In *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[4] Mahmoud Ammar, Ahmed Abdelraoof, and Silviu Vlasceanu. On bridging the gap between control flow integrity and attestation schemes. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6633–6650, 2024.

[5] ARM. Trustzone for cortex-a. https://www.arm.com/technologies/trustzone-for-cortex-a. Accessed: 2026-01-14.

[6] ARM. Trustzone for cortex-m. https://www.arm.com/technologies/trustzone-for-cortex-m. Accessed: 2026-01-14.

[7] ARM. Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. [Online], 2019. Available: https://developer.arm.com/documentation/ddi0487/fc.

[8] ARM. ARMv8-M Architecture Reference Manual. [Online], 2024. Available: https://developer.arm.com/documentation/ddi0553/latest/.

[9] Arm. Arm c language extensions (acle). https://arm-software.github.io/acle/, 2025. Accessed: 2026-02-02.

[10] Arm. *Armv8-M Security Extension User Guide*, 107655/100 edition, 2025. Arm Developer Documentation. Retrieved from https://developer.arm.com/documentation/107655/100.

[11] Md Armanuzzaman, Engin Kirda, and Ziming Zhao. Enola: Efficient control-flow attestation for embedded systems. *arXiv preprint arXiv:2501.11207*, 2025.

[12] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and Matthias Wählisch. Riot: An open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, 5(6):4428–4440, 2018.

[13] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security*, pages 30–40, 2011.

[14] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX security symposium*, volume 5, page 146, 2005.

[15] Wonwoo Choi, Minjae Seo, Seongman Lee, and Brent Byunghoon Kang. Sum: Efficient shadow stack protection on arm cortex-m. *Computers & Security*, 136:103568, 2024.

[16] André Cirne, Patrícia R Sousa, João S Resende, and Luís Antunes. Runpba–runtime attestation for microcontrollers with pacbti. *arXiv preprint arXiv:2512.12729*, 2025.

[17] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J Walls, and John Criswell. Holistic {Control-Flow} protection on {Real-Time} embedded systems with kage. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2281–2298, 2022.

[18] EEMBC. Coremark-pro. https://www.eembc.org/coremark-pro/, 2026. Accessed: 2026-02-02.

[19] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. Taclebench: A benchmark collection

to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

[20] FreeRTOS. Freertos/freertos — classic freertos distribution. https://github.com/FreeRTOS/FreeRTOS, 2026. Accessed: 2026-02-02.

[21] FreeRTOS. Freertos-mpu: The memory protection unit (mpu). https://www.freertos.org/Security/04-FreeRTOS-MPU-memory-protection-unit, n.d. Accessed: 2025-09-10.

[22] Alexander J Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P Kemerlis. Fineibt: Fine-grain control-flow enforcement with indirect branch tracking. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 527–546, 2023.

[23] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[24] Intel Corporation. Control-flow enforcement technology specification. Technical report, Intel Corporation, 2019.

[25] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. Tightly seal your sensitive pointers with {PACTight}. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3717–3734, 2022.

[26] Sashidhar Jakkamsetti, Youngil Kim, Andrew Searles, and Gene Tsudik. Eilid: Execution integrity for low-end iot devices. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7. IEEE, 2025.

[27] Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. Tzmcfi: Rtos-aware control-flow integrity using trustzone for armv8-m. *International Journal of Parallel Programming*, 49(2):216–236, 2021.

[28] Arslan Khan, Dongyan Xu, and Dave Jing Tian. Ec: Embedded systems compartmentalization via intra-kernel isolation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2990–3007. IEEE, 2023.

[29] Arslan Khan, Dongyan Xu, and Dave Jing Tian. Low-cost privilege separation with compile time compartmentalization for embedded systems. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3008–3025. IEEE, 2023.

[30] Zelun Kong, Minkyung Park, Le Guan, Ning Zhang, and Chung Hwan Kim. Tz-datashield: Automated data protection for embedded systems via data-flow-based compartmentalization. *Network*, 15:16, 2025.

[31] Hans Liljestrand, Thomas Nyman, Jan-Erik Ekberg, and N Asokan. Authenticated call stack. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–2, 2019.

[32] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, 2019.

[33] LLVM. Llvm 21.x. https://github.com/llvm/llvm-project/tree/release/21.x, 2025. Accessed on: January 28, 2026.

[34] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[35] Lan Luo, Yue Zhang, Clayton White, Brandon Keating, Bryan Pearson, Xinhui Shao, Zhen Ling, Haofei Yu, Cliff Zou, and Xinwen Fu. On security of trustzone-m-based iot systems. *IEEE Internet of Things Journal*, 9(12):9683–9699, 2022.

[36] Zheyuan Ma, Xi Tan, Lukasz Ziarek, Ning Zhang, Hongxin Hu, and Ziming Zhao. ret2ns-Cortex-M-TrustZone: Return-to-non-secure vulnerabilities on arm cortex-m trustzone. https://github.com/CactiLab/ret2ns-Cortex-M-TrustZone, 2023. Accessed: 2026-05-21.

[37] Zheyuan Ma, Xi Tan, Lukasz Ziarek, Ning Zhang, Hongxin Hu, and Ziming Zhao. Return-to-non-secure vulnerabilities on arm cortex-m trustzone: Attack and defense. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[38] Zheyuan Ma, Xi Tan, Lukasz Ziarek, Ning Zhang, Shambhu Upadhyaya, Hongxin Hu, and Ziming Zhao. Microft: Exploring and mitigating cross-state control-flow hijacking attacks on arm cortex-m trustzone. *IEEE Transactions on Information Forensics and Security*, 2025.

[39] A Theodore Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon W Moore, and Robert NM Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. 2019.

[40] Max Mino. Open syringe pump. https://github.com/manimino/OpenSyringePump, 2020. Accessed: 2026-05-21.

[41] Thomas Nyman, Jan-Erik Ekberg, Lucas 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*, pages 259–284. Springer, 2017.

[42] James Pallister, Simon Hollis, and Jeremy Bennett. Beebs: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint arXiv:1308.5174*, 2013.

[43] Jaeyeol Park, Yunju Gu, and Donghyun Kwon. Smore: Practical redzone-based stack memory error detection mechanism for embedded systems. In *2025 IEEE Annual Computer Security Applications Conference (ACSAC)*, pages 184–197. IEEE, 2025.

[44] Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto. Busted!!! microarchitectural side-channel attacks on the mcu bus interconnect. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3679–3696. IEEE, 2024.

[45] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[46] Zhuojia Shen, Komail Dharsee, and John Criswell. Fast execute-only memory for embedded systems. In *2020 IEEE Secure Development (SecDev)*, pages 7–14. IEEE, 2020.

[47] Zhuojia Shen, Komail Dharsee, and John Criswell. Randezvous: Making randomization effective on mcus. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 28–41, 2022.

[48] Xi Tan and Ziming Zhao. Sherloc: Secure and holistic control-flow violation detection on embedded systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1332–1346, 2023.

[49] The Zephyr Project. Zephyr RTOS 4.3.x. https://github.com/zephyrproject-rtos/zephyr/tree/v4.3-branch, 2025. Accessed on: January 28, 2026.

[50] The Zephyr Project. Zephyr latency_measure benchmark. https://github.com/zephyrproject-rtos/zephyr/tree/main/tests/benchmarks/latency_measure, 2026. Accessed: 2026-05-21.

[51] The Zephyr Project. Zephyr TF-M integration samples. https://github.com/zephyrproject-rtos/zephyr/tree/main/samples/tfm_integration, 2026. Accessed: 2026-05-21.

[52] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing {Forward-Edge}{Control-Flow} integrity in {GCC} & {LLVM}. In *23rd USENIX security symposium (USENIX security 14)*, pages 941–955, 2014.

[53] TrustedFirmware.org. Trusted firmware-m (TF-M) 2.2.x. https://github.com/TrustedFirmware-M/trusted-firmware-m/tree/release/2.2.x, 2024. Accessed on: January 28, 2026.

[54] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953. IEEE, 2016.

[55] Robert J Walls, Nicholas F Brown, Thomas Le Baron, Craig A Shue, Hamed Okhravi, and Bryan C Ward. Control-flow integrity for real-time embedded systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, pages 2–1. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[56] Jinwen Wang, Yujie Wang, Ao Li, Yang Xiao, Ruide Zhang, Wenjing Lou, Y Thomas Hou, and Ning Zhang. {ARI}: Attestation of real-time mission execution integrity. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2761–2778, 2023.

[57] Yujie Wang, C Mack, Xi Tan, Ning Zhang, Ziming Zhao, Sanjoy Baruah, and B Ward. Insectacide: Debugger-based holistic asynchronous cfi for embedded systems. In *Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'24)*. IEEE Computer Society Press., 2024.

[58] Yutian Yang, Songbo Zhu, Wenbo Shen, Yajin Zhou, Jiadong Sun, and Kui Ren. Arm pointer authentication based forward-edge and backward-edge control flow integrity for kernels. *arXiv preprint arXiv:1912.10666*, 2019.

[59] Gisu Yeo, Yeryeong Kim, Suhyeon Song, and Donghyun Kwon. Efficient cfi enforcement for embedded systems using arm trustzone-m. *IEEE Access*, 10:132675–132684, 2022.

[60] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. {In-Kernel}{Control-Flow} integrity on commodity {OSes} using {ARM} pointer authentication. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 89–106, 2022.

[61] Tai Yue, Kai Lu, Zhenyu Ning, Pengfei Wang, Lei Zhou, Xu Zhou, Yaohua Wang, Fengwei Zhang, and Gen Zhang. Efficient forward-edge control-flow integrity for cots binaries via arm bti. *IEEE Transactions on Information Forensics and Security*, 2025.

[62] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J Walls. Silhouette: Efficient protected shadow stacks for embedded systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1219–1236, 2020.

[63] Xia Zhou, Yujie Bu, Meng Xu, Yajin Zhou, and Lei Wu. ubox: A lightweight and hardware-assisted sandbox for multicore embedded systems. *IEEE Transactions on Dependable and Secure Computing*, 2024.