Computers & Security 136 (2024) 103568

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

SuM: Efficient shadow stack protection on ARM Cortex-M

Wonwoo Choi, Minjae Seo, Seongman Lee, Brent Byunghoon Kang *

Graduate School of Information Security, KAIST, South Korea

ARTICLE INFO

Keywords:
ARM
Software vulnerability
Exploit mitigation
Shadow stack
Control-flow integrity
Compiler

ABSTRACT

System software written in unsafe languages such as C/C++ is susceptible to various types of security vulnerabilities. Historically, backward-edges such as return addresses have been an attractive target for control-flow hijacking attacks due to the severity and ease of exploitation. Although various backward-edge control-flow integrity schemes have been proposed over the years, most of them mainly focus on protecting desktop/server-class systems, leaving embedded systems unprotected. Even worse, bringing their defense mechanisms into resource-constrained embedded systems is undesirable because they were originally designed for high-end computing systems and thus are not directly applicable to embedded systems without compromising performance and real-time constraints.

In this paper, we propose *Shadow under the Mask* (SUM), an efficient and robust backward-edge control flow protection that is applicable to ARM Cortex-M processors. Specifically, SUM realizes a non-bypassable shadow stack mechanism and safeguards its structural integrity in a novel combination of an MPU and `FaultMask`—an overlooked hardware feature in Cortex-M processors. To be more specific, SUM restricts all access to the shadow stack through MPU, ensuring its integrity; and temporarily disables its MPU protection through `FaultMask` during the execution of safe instructions, guaranteeing that only authorized instructions can modify the shadow stack. In our empirical evaluation, SUM incurs minimal runtime overhead of 2.77% and 2.63%, respectively, on the BEEBS and CoreMark benchmark suites. These results underscore the viability of our proposed approach as a practical and potent solution to address the highlighted cybersecurity challenge.

1. Introduction

Embedded systems, predominantly built upon microcontrollers (MCUs), are designed to execute specific tasks with stringent real-time and energy consumption constraints. They are pervasive, and their ubiquity extends across a wide range of devices, including but not limited to smart home appliances, implantable medical devices, and Unmanned Vehicles (UV). The advent of the Internet of Things (IoT) has augmented these systems with formidable networking capabilities. However, this newfound interconnectedness has also enlarged their attack surface, thereby elevating the importance of security in embedded systems to a paramount concern in both academic and industry spheres.

Regrettably, a majority of contemporary embedded systems are largely built using low-level, unsafe languages such as C/C++. While these languages offer performance benefits, their indiscriminate and careless utilization can pose significant security risks. Among these, memory errors constitute a critical vulnerability that can be exploited to launch control-flow hijacking attacks. These types of attacks can subvert the execution flow of a program, thereby allowing adversaries to carry out malicious activities. The situation is further compounded in the context of embedded systems, where system software typically operates at the same privilege level in a monolithic form.

A promising mitigation technique against such attacks involves enforcing control-flow integrity (CFI) (Abadi et al., 2005). However, despite its security benefit, the original CFI implementation (Abadi et al., 2005) with the protection of the shadow stack incurs non-negligible overhead. To alleviate the observed performance overhead, quite a few subsequent CFI researches have predominantly focused on implementing forward-edge CFI (e.g., protection for call and jump instructions) while comparatively neglecting backward-edges (i.e., return address). This regressive tendency leads to an erosion of system security, as recent studies (Carlini and Wagner, 2014; Göktas et al., 2014; Göktaş et al., 2014; Davi et al., 2014; Carlini et al., 2015) have demonstrated that both coarse-grained (Zhang and Sekar, 2013; Zhang et al., 2013; Pappas

* Corresponding author.
E-mail address: brentkang@kaist.ac.kr (B.B. Kang).

https://doi.org/10.1016/j.cose.2023.103568
Received 18 July 2023; Received in revised form 26 September 2023; Accepted 26 October 2023
Available online 31 October 2023

et al., 2013) and fine-grained (Abadi et al., 2005; Tice et al., 2014; Niu and Tan, 2014, 2015; Ding et al., 2017; Hu et al., 2018) solutions fall short of delivering their complete guarantee *without* the incorporation of a shadow stack.

Given the importance of the shadow stack, numerous variants of shadow stack techniques have been proposed over the years (Kuznetsov et al., 2014; Lu et al., 2015; Dang et al., 2015; Zieris and Horsch, 2018). Nevertheless, as outlined in recent studies (Abbasi et al., 2019; Yu et al., 2022), the majority of these defenses are geared toward desktop computers and servers. Compounding this issue, the integration of existing defense mechanisms into embedded systems presents formidable challenges. Namely, previous work commonly relies on (i) software instrumentation, (ii) randomization, or (iii) hardware isolation primitives—none of which can be effortlessly and seamlessly incorporated into embedded systems without sacrificing performance or security assurances.

More specifically, the first approach (Abadi et al., 2005) write-protects the shadow stack through software fault isolation (SFI) (Wahbe et al., 1993; Sehr et al., 2010), inevitably accruing significant overhead due to the heavy instrumentation of all memory write instructions. The second approach (Shacham et al., 2004; Kuznetsov et al., 2014; Lu et al., 2015; Zieris and Horsch, 2018) relies on obscuring its shadow stack through randomization rather than providing robust isolation, thereby rendering itself vulnerable to information leakage attacks (Evans et al., 2015; Gawlik et al., 2016; Oikonomopoulos et al., 2016). Furthermore, randomization is not practical for embedded systems as MCUs do not architecturally support virtual memory or abundant physical memory. The final approach (Cho et al., 2017; Koning et al., 2017; Pomonis et al., 2017; Frassetto et al., 2018; Vahldiek-Oberwagner et al., 2019; Hedayati et al., 2019; Burow et al., 2019; Ismail et al., 2021; Wang et al., 2020; Gravani et al., 2021; Xie et al., 2022) ensures the integrity of the shadow stack by actively employing hardware isolation primitives such as Intel MPK (Programming Guide, 2011). However, there is no functionally equivalent or comparable hardware security feature available in MCUs.

To address this gap, this paper proposes *shadow under the Mask* (SuM), an efficient and robust shadow stack designed for Cortex-M processors. The overarching goal of using SuM is to guarantee the integrity of the shadow stack through a novel intra-address space isolation called *selective masking*. This primitive is realized through a unique combination of the memory protection unit (MPU) and `FaultMask`—an execution priority control flag in ARM MCUs. Specifically, `FaultMask` enables a particular instruction sequence to temporarily escalate its privilege and be not subject to the restriction of the MPU. Through this, SuM restricts unauthorized access via the MPU and only allows authorized access by temporarily disabling the MPU with the help of `FaultMask`.

We develop a full prototype of SuM utilizing the LLVM infrastructure (Lattner and Adve, 2004). Subsequently, we conduct a comprehensive set of experiments on our prototype using the BEEBS and CoreMarks benchmark suites, focusing on its impact on runtime performance and flash memory overhead. Our comprehensive evaluation reveals that SuM, when tested on the two benchmark suites, incurs runtime overhead of 2.77% and 2.63%. Concurrently, the observed flash memory overhead amounts to 8.83% and 6.59%. It is important to note that the practical implications of the flash memory overhead are empirically determined to be considerably less pronounced than the recorded values.

In summary, we make the following contributions:

- We devise a selective masking, a novel and efficient primitive for intra-address space isolation. With this primitive, only authorized instructions are allowed to modify protected regions.
- We design SuM, a novel lightweight backward-edge control flow protection based on selective masking. Also, we implement a prototype of SuM on top of the LLVM compiler.

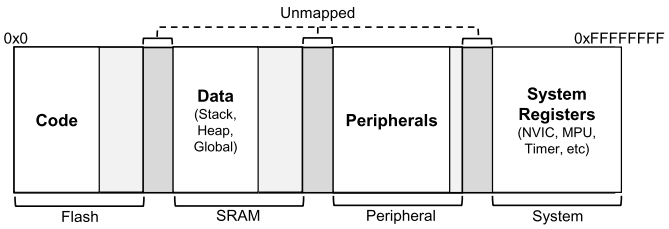

Fig. 1. The memory layout of ARM MCUs.

- We empirically evaluate the SuM implementation on the BEEBS and CoreMark benchmark suites. Our experimental results show that, on the two benchmark suites, the SuM implementation imposes 2.77% and 2.63% performance overhead and 8.83% and 6.59% for flash memory overhead.

2. Background

In this section, we describe the background information regarding ARM MCU architectures that is essential to understanding our study.

2.1. Memory layout of MCUs

As depicted in Fig. 1, the memory layout of ARM MCUs is generally divided into four regions as follows: `System`, `Peripheral`, `Flash`, and `SRAM` regions. The `System` region maps system control registers (e.g., interrupt controller (NVIC), MPU, timer), which are essential to configure the system behavior. Attached peripherals and controllers are memory-mapped in the `Peripheral` region. Therefore, software components can communicate with peripherals through memory-mapped I/O (MMIO). The `Flash` and `SRAM` regions are commonly used for the code region and for maintaining runtime data structures (e.g., stack and heap), respectively. Since the memory resources and peripherals are limited in MCUs, typically, there are unmapped memory regions between each of the aforementioned regions.

2.2. Memory protection unit

Contrary to general-purpose processors, MCUs lack virtual memory support provided by the memory management unit (MMU). Therefore, all execution contexts of embedded systems based on MCUs share the global address space, as depicted in Fig. 1. Instead of deploying an MMU, ARM MCUs typically implement a memory protection unit (MPU). The MPU enables a developer to configure access permission over the physical memory regions. While the maximum number of configurable memory regions varies from device to device, at least eight regions are supported in general. The configuration of MPU is done via the MPU control registers mapped in the `System` region.

2.3. Exception and exception priority

In ARM architectures, exceptions refer to any events that could divert the normal execution flow. It includes peripheral interrupts, software faults, and system calls. The handler of each exception needs to be defined and declared in a vector table located at the specified location of the read-only flash of an MCU.

To consider the difference in the criticality of events, ARM MCUs support exception priorities and allow an exception with a higher priority to preempt lower ones. The priority of an exception is configurable by setting the priority value, where a numerically lower value denotes a higher priority. While the priority value of the majority of exceptions can be defined by developers (with the lowest value being 0), Hard Fault and Non-Maskable Interrupts (NMI) have fixed priority values of -1 and -2, respectively.

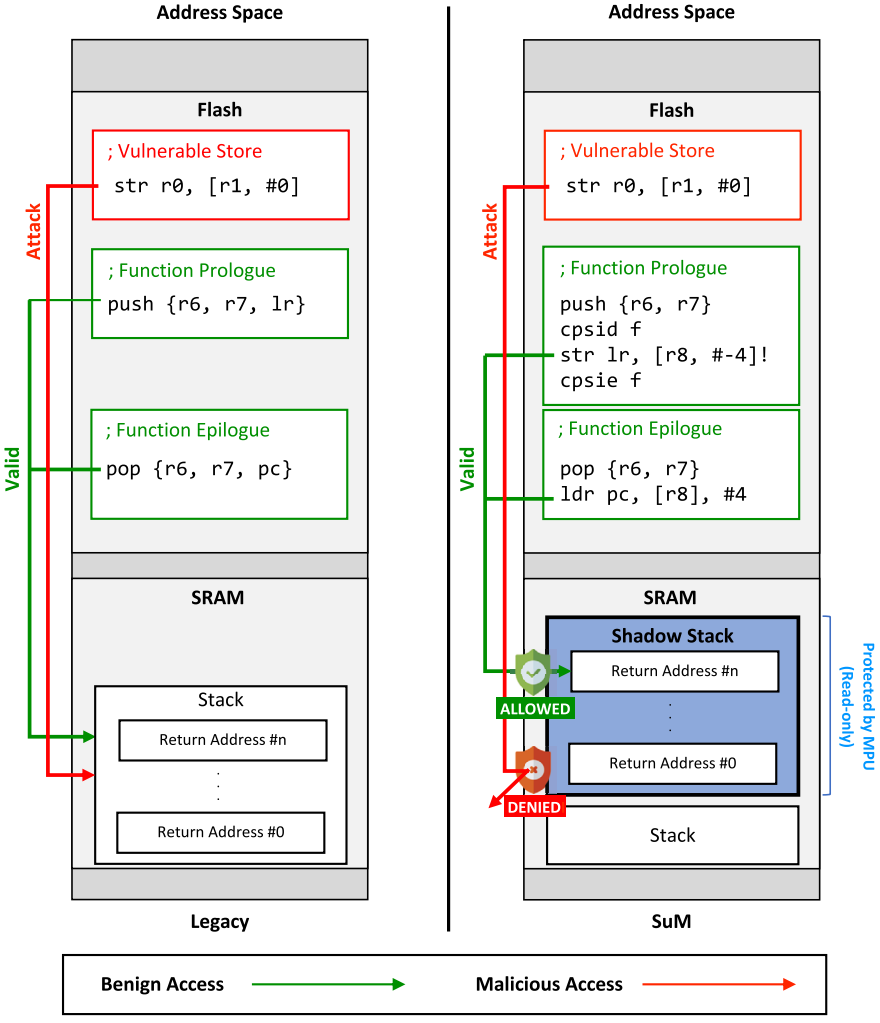

Fig. 2. The shadow stack protection of SuM.

3. Threat model and assumption

We assume a powerful adversary model that gains an arbitrary read/write primitive by exploiting memory corruption vulnerabilities in software. With this primitive, adversaries would attempt to overwrite return addresses to obtain arbitrary code execution. To prevent such an attack, SuM aims to prevent backward-edge control-flow hijacking. However, we assume that the adversary cannot launch a code modification or code injection attack due to the W⊕X. Additionally, non-control data attacks (Chen et al., 2005) are out of the scope of this work.

We mainly focus on bare-metal embedded systems that consist of a single task thread with a super-loop. However, it is worth noting that our design principle can be extended to OS-based embedded systems (the requirements for such an extension are explained in detail in Section 8). Furthermore, we assume that all software components operate at the privileged level, which is a typical choice for embedded systems to avoid mode-switching runtime overhead and thus satisfy the real-time constraints (Clements et al., 2017; Kim et al., 2018).

4. Design

SuM is a backward-edge control flow protection constructed upon the mechanism of a shadow stack. The integrity of the shadow stack is protected through a novel memory isolation primitive referred to as selective masking. In this section, we describe the inner workings of SuM. We initially present the design choices for deploying a shadow stack suitable for MCUs. Thereafter, we expound upon the capability and mechanism of selective masking. Additionally, we illustrate the interplay between the shadow stack and selective masking in SuM. Finally, we introduce supplementary security measures employed by SuM to protect against potential threats beyond trivial return address and shadow stack corruption.

4.1. Shadow stack

While there exist several designs for a shadow stack (Burow et al., 2019), identifying a suitable design is a critical endeavor, particularly for execution environments bound by various constraints. Notably, MCUs are subject to stringent constraints in runtime, memory, and flash memory. Given these constraints, the design of a shadow stack should be chosen based on two primary criteria: *Mapping* and *Encoding*. The mapping of a shadow stack determines whether the return address in the shadow stack is compact or parallel. Additionally, the encoding of a shadow stack governs the tracking of the shadow stack's top pointer.

Contrary to a compact shadow stack, a parallel shadow stack does not sequentially store return addresses. Rather, a parallel shadow stack stores return addresses at a constant offset from the original location in the regular stack, resulting in unused regions between return addresses. This characteristic renders a parallel stack out-performing in terms of runtime overhead since it obviates the need for tracking the shadow stack pointer. However, it inevitably incurs substantial mem-

ory overhead, as it requires the system to accommodate a shadow stack equivalent in size to the regular stack. Consequently, a compact shadow stack is a more suitable choice for MCUs.

Furthermore, for MCUs, two possible encoding schemes for shadow stacks are present: *Global Variable* and *Register*. The former encodes the shadow stack pointer in a global variable, while the latter does so in a dedicated register. Among these, SuM adopts the register encoding scheme, designating `r8` as the dedicated register. This is due to the fact that the global variable encoding scheme not only typically incurs higher runtime overhead but also incurs greater flash memory overhead due to the increased length of instructions required for shadow stack operations (Burow et al., 2019).

4.2. Selective masking

Selective masking is an integral component of SuM, serving a vital purpose in thwarting memory corruption attacks that target the shadow stack. It is a primitive that enables the restriction of access to a protected region exclusively to authorized store instructions, while preventing unauthorized store instructions from accessing it. This technique leverages the general functionality of the MPU to assist in mitigating unauthorized access attempts to the protected region. The approach employed to grant access exclusively to authorized instructions relies on specific features inherent in ARM MCUs:

- `MPU_CTRL.HFNMIENA`: A flag used to control the behavior of the MPU while handling exceptions with a priority value higher than -1. If this flag is set, the MPU is disabled. The affected exceptions include the `HardFault` exception, which is used to handle critical errors that are unlikely to be recovered without a reset (ARM, 2006, 2016).
- `FaultMask`: One of the flags that can be used to elevate the current execution priority value. When this flag is set via the `CPSID f` instruction, the current execution priority value is boosted to the `HardFault` exception's priority value (ARM, 2006, 2016). This register has been used by fault handlers, such as `BusFault`, to rectify errors while remaining independent of other faults.

Thus, selective masking's memory protection first works by setting the `MPU_CTRL.HFNMIENA` flag and configuring the MPU to the protected region as read-only. Next, selective masking wraps the `FaultMask` flag set (i.e., `CPSID f`) and clear instructions (i.e., `CPSIE f`) around the authorized store instruction. Subsequently, this configuration protects the MPU-configured memory region from all store instructions except those that are authorized.

4.3. Protected shadow stack

4.3.1. Concept

Once the shadow stack is deployed, the fundamental principle of SuM's integrity protection against memory corruption attacks rests on placing the shadow stack within a protected region of selective masking. Then, SuM authorizes shadow stack push instructions —wrapped with `FaultMask` flag instructions—to modify the shadow stack. The overview of this approach is depicted in Fig. 2.

Moreover, to ensure comprehensive protection, it is necessary to impose a limit on the size of the shadow stack, preventing potential overflow beyond the boundaries of the protected region. While embedding a limit check prior to each shadow stack push instruction can meet this requirement, it can also introduce unnecessary runtime overhead. As such, SuM strategically leverages the unmapped regions of the ARM MCU memory map. Specifically, the shadow stack is placed at the end of the SRAM region, adjacent to the unmapped region. Consequently, any attempts to overflow the shadow stack would trigger a `BusFault` exception, signifying a violation of the memory protection mechanism.

Algorithm 1: SuM Instrumentation.

Input: FL = $f_1, f_2, ..., f_n$ a set of Functions to Process

```
1  Procedure Initialization():
2      Initialize shadow stack pointer register (r8)
3      Initialize MPU_CTRL.HFNMIENA
4      Configure MPU to protected region as read-only
5  Procedure SuM_Instrumentation(FL):
6      for function f in FL do
7          if SpillLR(f) then
8              RetAddr ← GetSpillInstr(f)
9              ShdwStkPush ← ConvertShdwStkPush(RetAddr)
10             Restore ← GetSpillRestoreInstr(f)
11             ConvertShdwStkPop(Restore)
12             Wrap(CPSID, ShdwStkPush, CPSIE)
```

4.3.2. Instrumentation

As shown in Algorithm 1, we delineate the comprehensive workflow of SuM to deploy and protect the shadow stack.

Initialization Procedure. To initialize, the following steps are undertaken. Initially, at lines 2–3, the initialization procedure initializes the shadow stack pointer register (`r8`). Furthermore, for selective masking, `MPU_CTRL.HFNMIENA` is initialized to disable the MPU during HardFault. Subsequently, in line 4, the MPU is configured to establish a protected region with read-only permissions.

SuM *Instrumentation Procedure.* The primary instrumentation procedure is described as follows. It iterates over each set of functions within the function list (FL) in Line 6. For each function, it first determines whether a function needs to be protected by the SuM-enforced shadow stack. Specifically, a check is undertaken to ascertain whether the function spills the return address (i.e., `lr`) in Line 7. This spilling may transpire either because it is a non-leaf function or due to a consequence of high register pressure. If so, it proceeds to locate the instruction responsible for spilling the return address, along with the callee save registers, onto the stack (within the function *prologue*). Once identified, in Line 9, the return address spill operation is transformed into a shadow stack push operation. This modification involves changing the original instruction, such as replacing "`push r6, r7, lr`" with "`push r6, r7 + str lr, [r8, #-4]!`" (as depicted in Fig. 2). Similarly, in Line 10, the function identifies the instruction responsible for restoring the return address and callee save registers from the stack (within the function *epilogue*). Subsequently, in Line 11, the return address restore operation is modified into a shadow stack pop operation. For example, "`pop r6, r7, pc`" is transformed into "`pop r6, r7 + ldr pc, [r8], #4`" (as illustrated in Fig. 2). As a result, in Line 12, the function encapsulates the shadow stack push instruction.

4.4. SuM-aware attack prevention

While SuM's shadow stack protection can thwart straightforward attempts to corrupt the shadow stack, adversaries may gain an understanding of SuM's mechanisms. Consequently, there could be attempts to disable or bypass the shadow stack protection provided by SuM. In such instances, SuM needs to guarantee that return instructions cannot be exploited to hijack the control flow to locations other than the original call sites. The subsequent sections, in conjunction with Fig. 3, detail potential efforts to disable or bypass SuM's shadow stack protection and the corresponding mitigation.

4.4.1. W⊕X (write XOR execute) policy

Enforcing the W⊕X policy (Microsoft, 2018) is the foremost requirement for security measures based on code instrumentation. Otherwise, adversaries can overwrite the code region to achieve arbitrary code execution. For example, the adversaries can forge a selectively masked store in the code region to corrupt the shadow stack and hijack backward-edge control flow. In this case, SuM effectively enforces

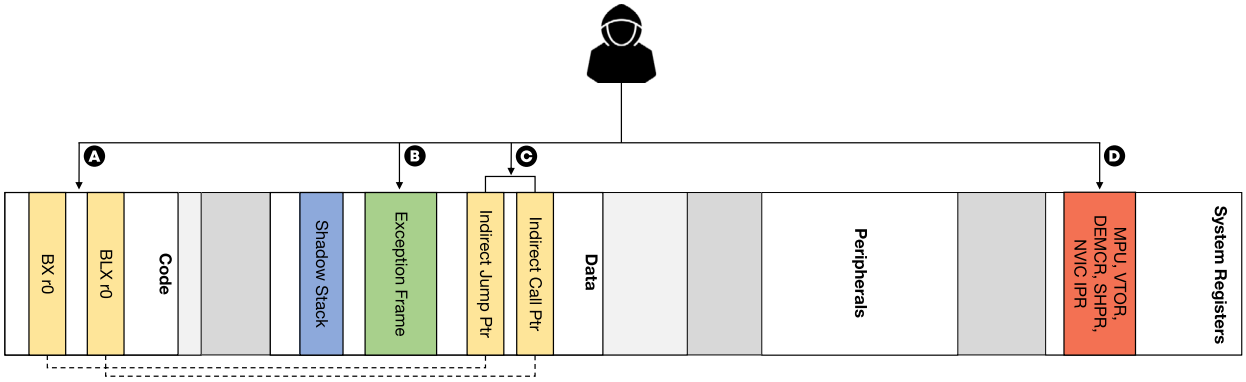

Fig. 3. SuM attack vectors. A Code Corruption, B Exception Frame Corruption, C Indirect Call/Jump Pointer Corruption, D System Register Corruption.

the policy using the MPU and ensures that write-able regions are non-executable.

4.4.2. Exception return protection

Exception return is a special type of backward-edge control flow for exception handlers; it requires extra measures to protect. Upon the arrival of an exception, an ARM MCU preempts the currently executing instruction and makes a data structure (i.e., an exception frame) before executing the corresponding exception handler. An exception frame consists of the data required to perform an exception return, and it includes the preempted program counter and return address register (`lr`) value. Thus, the corruption of an exception frame can result in backward-edge control flow hijacking.

To cope with this issue, SuM protects the security-critical exception frame data, such as a preempted program counter, using the shadow stack in a manner similar to how it handles return addresses. To this end, SuM replaces the exception vector table, which is referred to for locating the associated handler when an exception arrives, with another table full of addresses for the SuM-generated exception dispatcher. Upon an exception's arrival, the exception dispatcher stores the security-critical exception frame data in the shadow stack and makes a call to the original handler. After the control flow returns from the original handler, the exception dispatcher then restores the saved exception frame data before performing an exception return.

4.4.3. Forward-edge control flow protection

While SuM primarily focuses on protecting backward-edge control flow, which covers a wide range of arbitrary code execution attacks, it is essential to defend against potential abuse of forward-edge attacks; otherwise, this could undermine our security guarantees. Several threats might arise from the following possible attacks: (1) abuse of unintended instructions and (2) jump to a function prologue.

(1) Abuse of Unintended Instructions. ARM Cortex-M supports a variable-length Thumb-2 instruction set composed of intermixed 32-bit or 16-bit instructions. Hence, adversaries may exploit the occurrence of unintended (unaligned) 16-bit instructions located in the middle of an intended 32-bit instruction. For instance, a `FaultMask` set instruction, `CPSID f`, could accidentally occur in the middle of an intended 32-bit instruction; as a result, adversaries can abuse it as a means to turn off the protection of the shadow stack.

(2) Jump to the Prologue with Manipulated `lr`. Beyond exploiting unintended instructions, adversaries can corrupt the shadow stack by using only intended instructions. As a concrete example, adversaries with control of both (1) the target of an indirect jump (not a call) and (2) the value of the link register (`lr`) can jump to a function prologue with a maliciously crafted the `lr` register, which keeps the `lr` register intact (whereas the call instruction does not), thereby gaining the ability to push an arbitrary value onto the shadow stack.

Table 1
Security-sensitive system control registers.

Register	Description
MPU	MPU configuration registers
VTOR	Exception vector table relocation register
DEMCR	Debug feature (e.g., DWT) activation register
DWT	DWT configuration registers
SHPR	System handler exception priority register
NVIC_IPR	Peripheral exception priority register

To resolve these issues, SuM employs a coarse-grained CFI protection mechanism to prevent the aforementioned potential abuse of indirect calls and jumps. Specifically, in the case of the indirect calls, it applies label-based CFI that inserts a unique label into every function entry, which is similar to the technique proposed in Du et al. (2022). The label is specially chosen so that it cannot be produced by the compiler according to the specification of ARM instruction encoding (ARM, 2006). This restricts the potential branch targets of indirect calls exclusively to function prologues. In the case of indirect jumps, SuM converts all indirect jumps into jump-table-based branches (e.g., using the `tbb` instruction) with index masking. This instrumentation restricts the indirect jumps to only valid jump targets; they are not allowed to jump to unintended instructions or function prologues.

4.4.4. System register protection

SuM requires the protection of security-sensitive system control registers, which are summarized in Table 1. Allowing adversaries to manipulate these registers results in the subversion of its security guarantees. A challenge in protecting the registers is that the MPU cannot enforce access control on System Control Space (SCS) (ARM, 2006, 2016), which is a memory region mapping the system control registers.

Therefore, to ensure the integrity of system control registers, we leverage a watchpoint-based protection technique, which employs the watchpoint unit found in ARM MCUs, namely, Data Watchpoint Trace (DWT), similar to that used in the previous work (Shen et al., 2020). To be more precise, the DWT has the ability of monitoring and raising the `DebugMonitor` exception upon detecting access to monitored memory. Contrary to the MPU, the DWT has the ability to monitor the SCS region, thus enabling SuM to closely monitor all memory access to security-critical system control registers. In the event of an attack attempt, the `DebugMonitor` exception handler can be activated to initiate appropriate responses (e.g., reset the system). It is pertinent to note that the `DebugMonitor` exception is deactivated when handling an exception of equivalent or higher priority (ARM, 2006, 2016). Therefore, it is incumbent upon developers to reserve the highest exception priority for the `DebugMonitor` exception, ensuring a robust line of defense.

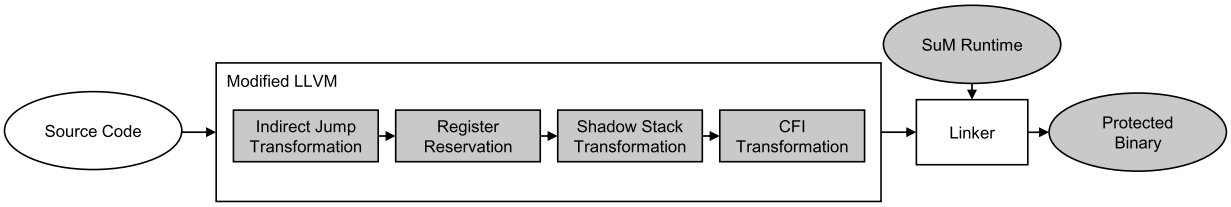

Fig. 4. SuM deployment process.

Table 2
The permission configuration of the MPU and DWT units.

Unit	Regions Covered	Perm.
MPU_0	Code in Flash	r_x
MPU_1	Others in Flash (e.g., data)	rw_
MPU_2	Shadow Stack	r__
MPU_3	Others in SRAM (e.g., heap)	rw_
DWT_0	MPU, VTOR, DEMCR, SHPR, NVIC_IPR	r__
DWT_1	DWT	r__

5. Implementation

We built a prototype of SuM on top of the LLVM 9.0 compiler framework (Lattner and Adve, 2004). The LLVM compiler was modified to apply our compile-time instrumentation on embedded system source code. Additionally, we developed a static runtime library to provide supplementary code and data necessary for the functionality of SuM. The subsequent sections will delve into the specific modifications made to the LLVM compiler and the inner workings of the runtime static library.

5.1. LLVM modification

As seen in Fig. 4, the modified version of LLVM encompasses four fundamental components: the indirect jump transformation, register reservation, shadow stack transformation, and CFI transformation. The indirect jump transformation is integrated into the LLVM middle-end, which is designed to process the LLVM intermediate representation (IR). On the other hand, the remaining components are implemented within the LLVM backend responsible for handling machine-specific instructions.

The indirect jump transformation involves converting indirect jumps into switch statements, effectively preventing the abuse of indirect jumps (Section 4.4.3). This conversion is implemented using the `IndirectBrExpandPass` pass, a built-in middle-end pass in LLVM. To ensure that no modifications are inadvertently reversed, the pass is positioned at the end of the LLVM middle-end. Additionally, the register reservation ensures that the `r8` register is not allocated during the register allocation stage, and it is facilitated by LLVM's `markSuperRegs` function. The shadow stack transformation pass deploys and safeguards a shadow stack, following the guidelines outlined in Algorithm 1. Lastly, the CFI transformation pass focuses on enforcing label-based CFI (Section 4.4.3) by inserting a unique label at function entry points and incorporating label check instructions for indirect calls.

5.2. Runtime library support

The SuM runtime library comprises the initialization code, exception dispatcher, and redirection exception vector table. The initialization code is responsible for configuring system control registers, such as the DWT for system register protection (Section 4.4.4) and the MPU for selective masking (Section 4.2). The exception dispatcher and redirection exception vector table serve as core components in exception return protection (Section 4.4.2).

The detailed configuration is summarized in Table 2. Regarding the MPU configuration, three MPU units, `MPU_0`, `MPU_1`, and `MPU_3`, are designated to map the code (`rw`), data (`rw`), and heap (`rw`) regions, respectively. The remaining MPU unit, MPU_2, is used for our purpose to configure the shadow stack as read-only. Regarding the DWT configuration, both of the two DWT units are configured to ensure the integrity of security-sensitive system control registers. In detail, while `DWT_0` is used to protect MPU, VTOR, DEMCR, SHPR, and NVIC_IPR as they are adjacent to each other. The DWT itself, which is far apart from the preceding registers, is protected by `DWT_1`.

6. Evaluation

In this section, we evaluate SuM with respect to both runtime and flash memory overhead. Our evaluation leverages BEEBS (Pallister et al., 2013) and CoreMark (EEMBC), open-source benchmark suites that have been utilized extensively in previous studies as a measure of mitigation overhead (Zhou et al., 2020; Almakhdhub et al., 2020; Kwon et al., 2019). Regarding BEEBS, it encompasses multiple workloads with durations that may be deemed impractically brief. Consequently, we selected 29 workloads that have been identified as having a more extended lifespan (Zhou et al., 2020).

As a baseline, we compiled our test program using the unmodified LLVM 9.0, which is the same version upon which our modified compiler is based. We employed the -O3 compiler optimization and link-time optimization (`-flto`) for both the baseline and SuM compilations. Furthermore, we utilized picolibc (Packard Picolibc, 2018) as the standard C library for both the baseline and SuM, given its capacity to facilitate convenient scripts for compiling using LLVM. Our evaluation results derived from our prototype, SuM, are described with more details in Appendix A.

6.1. Runtime overhead

We conducted all experiments on the NUCLEO-F401RE development board equipped with a Cortex-M4 MCU, 512 KB flash memory, and 96 KB SRAM. The MCU was configured to run at its default frequency (84 MHz). The execution time was ascertained utilizing the `DWT` timer, characterized by its cycle-level granularity. Given that this measurement approach necessitates adjustments to the DWT registers, we temporarily suspended the system protection mechanisms pertinent to these registers exclusively for the purpose of the evaluation. Each workload underwent execution across 10,240 iterations.

The primary components of SuM that could influence an application's runtime include the shadow stack, selective masking, CFI, and exception return protection (ERP). The shadow stack incurs its overhead mainly due to the insertion of shadow stack push and pop instructions, along with the change in register allocation caused by the shadow stack register reservation. The selective masking component also contributes to the overhead through the insertion of `cps` instructions. The CFI overhead predominantly arises from the label check instructions. Lastly, the ERP imposes overhead due to the backup and restoration of contents within the exception frame, though its impact is negligible as it is only activated during exception handling.

Fig. 5 and Fig. 6(a) present the evaluation outcomes. SuM imposes geomean runtime overhead of 2.77% and 2.63% on BEEBS and CoreMark, respectively. To discern the origins of this overhead, we executed

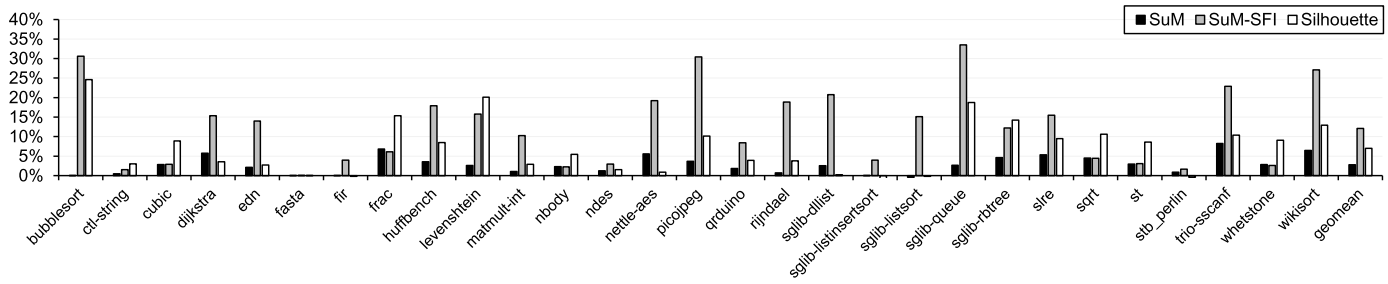

Fig. 5. Runtime overhead on BEEBS.

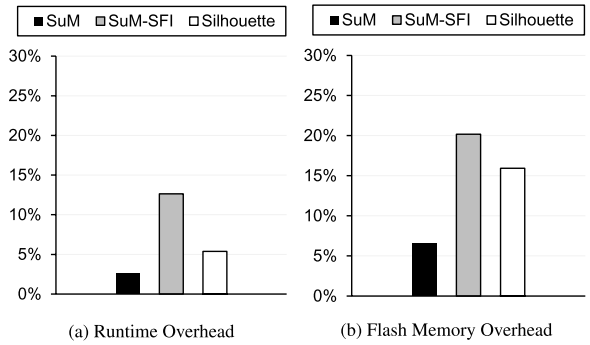

Fig. 6. Runtime and flash memory overhead on CoreMark.

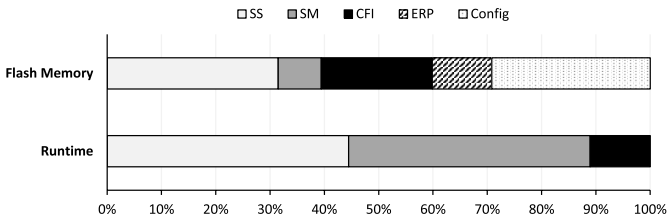

Fig. 7. Constitution of the overhead on BEEBS. *SS* for Shadow Stack, *SM* for Selective Masking, *ERP* for Exception Return Protection, and *Config* for Configuration.

further analyses on the individual components of SuM within the context of BEEBS. As shown in Fig. 7, the predominant contributors to this overhead are the incorporation of the shadow stack and the implementation of selective masking, which are responsible for 1.43% and 0.96% of the overall runtime overhead, respectively. Conversely, exception return protection imposes negligible runtime overhead, owing to its activation solely during the exception's arrival, which is an infrequent occurrence.

6.2. Flash memory overhead

In many embedded systems, the overhead of flash memory can be a crucial factor due to the limited resources of flash memory. Thus, we calculated the flash memory overhead of SuM for each workload by dividing the flash memory consumption of the corresponding binary with that of its corresponding baseline. The size information of each binary was collected using the `size` command.

In addition to the components mentioned in Section 6.1, the configuration code of SuM might influence the flash memory usage of an application. For both the shadow stack and selective masking, the primary causes of flash memory overhead are similar to those of runtime overhead, which are shadow stack instructions and CPS instructions in the `.text` section. For CFI, the label affixed to each indirect call target, along with the label-checking instructions, leads to an increase in the `.text` section. For ERP, the exception frame backup and restoration instructions contribute to overhead in the `.text` section, while the redirection exception vector table contributes to the overhead in the `.rodata` section. Lastly, the configuration code adds instructions to configure the MPU and the DWT into the `.text` section. It is worth noting that the overhead incurred by ERP and configuration code is constant (464 bytes and 174 bytes, respectively).

Fig. 8 and Fig. 6(b) illustrate the flash memory overhead of SuM. The geomean flash memory overhead for SuM on BEEBS and CoreMark is 8.83% and 6.59%, respectively. However, based on the further studies on the overhead constitution of SuM using BEEBS, our analysis reveals that a significant portion of the overhead can be attributed to the constant overhead, specifically ERP (0.96%) and configuration code (2.56%). Concurrently, the shadow stack, selective masking, and CFI contribute to flash memory overhead of 2.76%, 0.69%, and 1.79%, respectively. These findings imply that in an embedded system characterized by an extensive code base, the code size overhead will closely mirror that emanating from the deployment of the shadow stack, selective masking, and CFI. With the formula defined as:

$$\frac{(1+NCO)*BaselineSize+CO}{BaselineSize}$$

we can forecast the trend of SuM's flash memory overhead according to the baseline flash memory consumption, where `NCO` represents the non-constant overhead ratio and `CO` represents the constant overhead in bytes. For example, for BEEBS, with an `NCO` of 5.25% (i.e., the sum of shadow stack, selective masking, and CFI) and a `CO` of 638 bytes (i.e., the sum of ERP and configuration code), the result of the formula —upon integrating a `BaselineSize` of 2 MB— indicates that the practical flash memory overhead of SuM is approximately 5.28% for MCUs equipped with a flash memory capacity of 2 MB or beyond.

6.3. Comparison with SFI

In the absence of an effective intra-address space isolation primitive, a purely software-based technique known as software fault isolation (SFI) might be utilized to protect the shadow stack. SFI inserts bit masking instructions before each store instruction to prohibit access to the protected region. However, it requires the heavy code instrumentation on instructions that do not directly constitute shadow stack operations. Such code instrumentation incurs additional runtime overhead and flash memory overhead.

To evaluate the efficiency of shadow stack protection methods, we devised re-implemented SuM based on SFI and conducted evaluation with BEEBS. SuM-SFI inserts ORR instruction, which sets the specified bit location of a register, before each store instruction to prevent the store from accessing the shadow stack region. Specifically, our board incorporates a 96 KB SRAM. Within this configuration, SuM-SFI designates the shadow stack to reside in the initial 64 KB segment of the SRAM. Subsequent to this allocation, ORR instructions are inserted to set the 16th bits of the destination address of store instructions. This mechanism restricts any store operations directed towards the initial 64 KB segment of the SRAM, effectively protecting the shadow

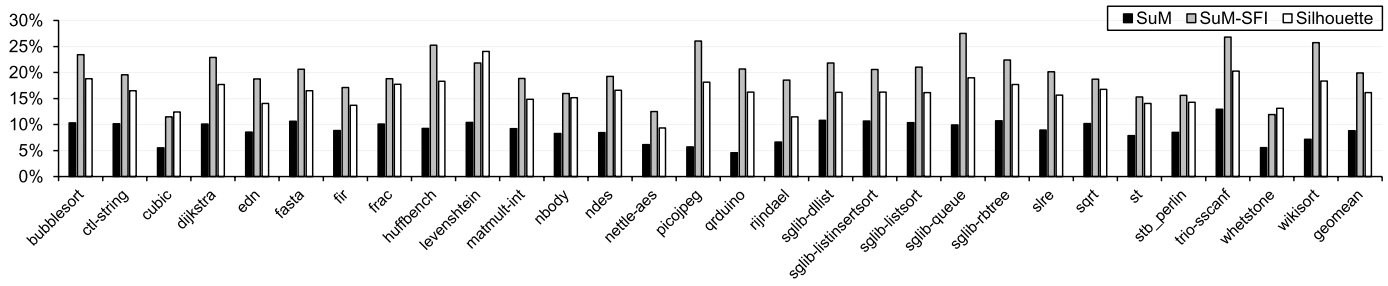

Fig. 8. Flash memory overhead on BEEBS.

stack. Additionally, to ensure normal operations, SuM-SFI must be configured to accommodate store operations that interface with non-SRAM regions, encompassing peripheral and system regions. To this end, SuM-SFI discerns these specific instructions through the utilization of backward-slicing techniques (Clements et al., 2017). Following their identification, these instructions are manipulated with ORR operations to permit access solely to the designated regions and concurrently exclude the broader SRAM region (i.e.,0x20000000-0x3FFFFFFF). For example, for instructions accessing the system region (i.e.,0xE0000000-0xFFFFFFFF), setting the 30th bit of the destination address should suffice the requirement. It is imperative to recognize that this does not epitomize an optimal SFI scheme in the context of memory utilization — it underutilizes more than half of the available SRAM. This design was chosen specifically to facilitate the empirical estimation of runtime and flash memory overhead intrinsic to SFI-based shadow stack protection mechanisms.

The findings are graphically illustrated in Fig. 5 and Fig. 8. SuM-SFI manifests pronounced overhead, both in terms of runtime and flash memory, with respective overheads of 12.12% and 19.9% for BEEBS and 12.63% and 20.17% for CoreMark. For comparative clarity, when isolating from the constant overhead components, such as ERP and configuration code, SuM-SFI imposes flash memory overhead of 15.92% and 17.55% for BEEBS and CoreMark, respectively.

6.4. Comparison with silhouette

To overcome the inefficient SFI-based isolation, Silhouette (Zhou et al., 2020) proposed its own intra-address space isolation primitive for ARM MCUs, namely store hardening. Store hardening replaces insecure store instructions with unprivileged store instruction (e.g., `STRT`) and configures the MPU to restrict a protected region to be privilege-accessible. By doing so, only unconverted secure store instructions are permitted to access the protected region. With store hardening, Silhouette implements a secure shadow stack by only allowing shadow stack push instructions to manipulate the shadow stack.

Despite the notable reduction in overhead compared to SFI, the store hardening technique still imposes considerable runtime and flash memory overhead. While the store hardening technique requires the conversion of all insecure store instructions, there are various edge cases where the conversion to an unprivileged store is not obvious. For example, unprivileged store instructions cannot manage negative immediate offsets as well as multiple stores. In such cases, additional fix-up instructions must be inserted to preserve the original logical flow with the cost of runtime and flash memory. Moreover, the innate 4-byte size of all unprivileged store instructions, contrasted with the predominant 2-byte configuration of regular store instructions, amplifies the flash memory overhead of Silhouette.

To compare SuM with Silhouette, we acquired the LLVM passes of Silhouette (Zhou et al., 2021) and ported the passes to our compiler. Fig. 5 and Fig. 6a exhibit the runtime overhead associated with Silhouette, while Fig. 8 and Fig. 6b provide insight into its flash memory overhead. In this comparative analysis, Silhouette manifests runtime overhead of 6.97% and 5.37% for BEEBS and CoreMark, respectively, compared to 2.77% and 2.63% for SuM. Additionally, when considering flash memory overhead, Silhouette notably exceeds SuM with metrics of 16.15% and 14.42% for BEEBS and CoreMark, respectively. Excluding the constant overhead attributed to configuration code, Silhouette's flash memory overhead stands at 15.57% and 13.57% across the two benchmark suites. It is imperative to note that, in the pursuit of a fair comparison, our empirical methodologies diverge from the original experiments of Silhouette. Specifically, we incorporate the code instrumentation on standard and HAL libraries, aligning with SuM's threat model, which assumes potential vulnerabilities to memory corruption across any part of the code.

7. Security analysis

In this section, we demonstrate how SuM ensures the security of the backward-edge control flow against memory corruption attacks. Specifically, we elucidate how the shadow stack protection and the SuM-aware attack mitigation mechanisms obstruct adversaries from manipulating the control flow of return instructions.

7.1. Return address corruption

Adversaries, particularly those unaware of the system's protective mechanisms or only conscious of the existence of the shadow stack, may attempt to corrupt return addresses in either the stack or the shadow stack region. SuM thwarts both of these attempts. Conventional buffer overflow attacks, which typically overwrite return addresses, fail due to the presence of the shadow stack deployed by SuM. Furthermore, more sophisticated attempts to directly corrupt the shadow stack also fail, as none of the store instructions, except for those in the function prologues, are selectively masked. Thus, any attempt at corruption will result in a memory permission violation fault, thereby triggering detection.

7.2. Disabling selective masking

The activation of selective masking depends on the configuration of several security-sensitive system control registers, as summarized in Table 1. Consequently, in order to disable the selective masking protection, adversaries must corrupt these critical system control registers. SuM prevents such attempts by configuring the DWT's debug registers, ensuring that any attempt to disable selective masking fails, even when faced with arbitrary read-write primitives.

7.3. Bypassing selective masking

Without the capability of deactivating the selective masking, the only approach for adversaries to divert the backward-edge control flow under SuM is to bypass the protection. This bypassing could be achieved either by relocating the shadow stack with `r8` manipulation or by exploiting the existing selectively masked store instructions—whether they were intended or not—to corrupt the protected shadow stack.

The relocation of the shadow stack requires gadgets to manipulate `r8`. However, since `r8` is reserved during the compilation process, any intended instructions manipulating r8 are not produced except the instructions that we explicitly added for the shadow stack. Further, any unintended instructions manipulating r8 cannot be reached due to the SUM's protection applied to indirect calls and indirect jumps (Section 4.4.3). Also, note that the recurrent utilization of epilogues to pop return addresses not initially generated by prologues is proscribed. This restriction is imposed because function prologues invariably dominate all intra-function codes, encompassing the function epilogues. Additionally, SUM's indirect jump protection inherently precludes any indirect jumps that target locations outside the intended functions.

Regardless of the approach, to exploit existing selectively masked store instructions, adversaries must 1) divert the control flow towards the store instruction and 2) gain control of the register that the store instruction employs, which is `lr` for intended ones. Thwarting either of these conditions can prevent such bypass attempts. However, any attempt to exploit unintended selectively masked store instructions is destined to fail because the first prerequisite condition cannot be fulfilled. While the only remaining control data for control flow diversion are exception frames and indirect call/jump pointers, the exception return protection of SUM effectively prevents its misuse through the backup-and-restoring mechanism. Additionally, in the case of the abuse of indirect call/jump pointers misuse, the label-based CFI and table-based jump conversion ensure that the execution flow could only branch to function prologues or valid jump targets within the function, but not to any unintended instructions.

Moreover, attempts to corrupt the shadow stack based on intended selectively masked stores are also unlikely to succeed due to both the unattainable first and second conditions. First, indirect jumps cannot be abused due to the first condition, as the selectively masked store instructions only exist within function prologues, which are not valid jump targets. Nevertheless, although it is exceedingly complex, the first condition could potentially be met during an attempt to abuse the exception frame. This is because exceptions can occur asynchronously at any point in the program, including function prologues. However, fulfilling the second condition is impossible because the `lr` register within the exception frame is safeguarded with SUM's exception return protection. Also, attempt to misuse indirect calls fail due to the second condition, as the indirect call itself replaces the `lr` register with the address of call-site.

8. Discussion

While the primary focus of this study pertains to bare-metal embedded systems, it is worth noting that the strategies and methods outlined herein could be suitably extended to accommodate OS-based embedded systems. These systems would require only modest adaptations in terms of the SUM runtime and the relevant operating system (OS). The following delineates the necessary modifications:

Extended Exception Return Protection. In contrast to bare-metal embedded systems, OS-based embedded systems deploy a separate stack for handling exceptions, which facilitates the implementation of context switching. Essentially, two distinct stacks are utilized: one for exception-mode stack and another for thread-mode stack, with the choice between them contingent on the current execution mode. However, this arrangement presents a challenge in locating the exception frames warranting protection. The issue arises due to the nested exceptions supported by ARM architecture, whereby an exception frame can be engendered either on the exception handler stack or the normal stack, depending on the point of preemption. As a result, a minor adaptation to the exception dispatcher is necessary to identify and locate the previous exception frame accurately.

Shadow Stack Switching. In OS-based embedded systems, wherein multiple tasks coexist, each task needs to deploy its own shadow stack. Therefore, the operating system requires a mechanism similar to the regular thread-mode stack switching conducted during context switching. This necessitates the appropriate adaption of the context switching routine. A crucial aspect to consider is that the current shadow stack during context switching contains return addresses that remain to be consumed. Consequently, the operation of switching the actual shadow stack pointer must be deferred until the exception returns to the context that underwent the context switch.

Shadow Stack Pointer Storage. Given the utilization of multiple shadow stacks, it is an inevitable consequence that inactive shadow stack pointers will be stored in memory during shadow stack switching. To ensure the integrity of these pointers, a robust storage mechanism is essential. In this context, selective masking can be deployed to protect the shadow stack pointers, utilizing a strategy similar to SUM's protection of the shadow stack. In other words, SUM could be extended to define an additional protected region for shadow stack pointers and authorize shadow stack switching instructions.

Further Securing System Registers. In contrast to bare-metal embedded systems, where concurrency of multiple tasks can lead to potential synchronization issues, there is a requisite need for implementing exception masking as a safeguard against race conditions. However, without any consideration on the protection of SUM, DWT exceptions might also be blocked, thus allowing attackers to manipulate security-sensitive system registers (e.g., MPU). Thus, instructions to completely mask all exceptions (i.e., `CPSID I`) must be converted to partial exception masking instructions (i.e., `MSR BASEPRI, reg`) to prevent DWT exceptions from being suppressed under any circumstance unless the instructions to be executed are verified carefully to not break the protection of SUM.

Controlling MPU. While it may not be a common case for current practice, there could be security-sensitive OS-based embedded systems requiring MPU-based compartmentalization of tasks. Considering such cases, extending SUM to support MPU switching emerges as a pertinent avenue for future exploration. For that, exception masking may be used to temporarily disable DWT and allow manipulation of MPU. However, it is imperative to verify the manipulation code to ensure that it does not alter the MPU regions under SUM's management.

9. Related works

A variety of shadow stack mechanisms has been proposed to ensure the integrity of return addresses with their respective runtime integrity guarantee for the security metadata (e.g., shadow stack). In the following, we first elaborate on closely related work on embedded systems and its limitations, paying particular attention to how to guarantee the integrity of its security metadata. Furthermore, we briefly outline other related works for high-end processors, such as x86-64 and ARM Cortex-A.

9.1. Backward-edge protection for MCUs

Information Hiding. EPOXY (Clements et al., 2017) realizes a safe stack mechanism (Kuznetsov et al., 2014) that divides the traditional stack into the safe stack and unsafe stack. Specifically, it separates stack objects into two groups: safe objects (e.g., benign local variables, return address), which are placed in the safe stack; and unsafe objects (e.g., address-taken variables, buffers), which are placed in the unsafe stack.

To raise the bar against memory errors, EPOXY randomizes its location within the global data segment (placed in the SRAM area). However, it still leaves the safe stack open to memory corruption attacks because such a randomized address is determined at the compile-time; and moreover, the randomization relies on a few kilobytes of the SRAM, which is insufficient to thwart information leakage attacks (Evans et al., 2015; Oikonomopoulos et al., 2016; Gawlik et al., 2016), much less brute force attacks (Shacham et al., 2004).

Coarse-grained CFI. BackFlow (Bresch et al., 2020) suggests a bitmap-based, coarse-grained CFI technique to enforce that all backward edges return only to call-preceded instructions. Specifically, BackFlow creates a bitmap that is used to mark the bits representing the prior call-sites, and uses it to check at runtime whether the destination of return addresses is preceded by the call instruction. However, this approach provides no protection for bitmap against memory corruption attacks. Also, it is too coarse-grained to prevent CFI bypass attacks (Carlini and Wagner, 2014; Göktaş et al., 2014; Davi et al., 2014), e.g., stitching an ROP chain with call-preceded gadgets.

ARM TrustZone-M. Since ARMv8 Cortex-M, ARM introduced TrustZone-M (Yiu, 2015), a lightweight version of TrustZone, which was previously only supported on high-end Cortex-A processors. Notably, it has an extremely fast world-switching mechanism, which takes around 4 cycles. Based on this finding, TZmCFI (Kawada et al., 2021) and CFI CaRE (Nyman et al., 2017) implement and protect a shadow stack based on TrustZone-M. In other words, the shadow stack is placed in the secure world, which is separated from the non-secure world where normal applications run. The TrustZone-based isolation protects the shadow stack against all types of unauthorized access from the non-secure world and hence achieves strong backward-edge control flow protection. However, only the ARMv8-M architectures support TrustZone-M, whereas the `FaultMask` hardware feature is supported by both ARMv7-M and ARMv8-M MCUs.

Unprivileged Instruction. Closest related to our work, Silhouette (Zhou et al., 2020) and Kage (Du et al., 2022) introduce an efficient intra-address space isolation technique through the store hardening technique and make use of it to protect the shadow stack. The store hardening technique is based on unprivileged store instructions, which perform memory write operations regardless of the current privilege level as if it were unprivileged. Specifically, they transform all store instructions except shadow stack instructions into unprivileged store instructions (e.g., `strt`), while configuring the shadow stack to be only accessible in the privileged mode using the MPU. In the end, since the attack is forced to launch memory corruption attacks against the shadow stack with unprivileged store instructions, the integrity of the shadow stack mapped as privileged is guaranteed. However, both of them rely on heavy code instrumentation, thus incurring non-trivial performance overhead.

Register Encoding. μRAI (Almakhdhub et al., 2020) achieves a backward-edge control flow protection using a state register, which encodes the current execution path. μRAI encodes the path information for each function call in the `lr` register. Then, for each return, the return target is identified based on the path information in the `lr` register using Function Lookup Tables (FLTs), which consist of return addresses indexed with the encoded path information. While the approach is safe from memory corruption, given the `lr` register is not spilled, it incurs non-negligible code size overhead for storing FLTs.

9.2. Backward-edge protection for general-purpose CPUs

Contrary to MCUs, general-purpose processors are commonly equipped with various kinds of hardware features that can be employed or repurposed for security. As a result, numerous hardware-assisted researches have been proposed to design and implement efficient and effective back-edge protection. Here, we discuss these existing works on general-purpose systems.

Information Hiding. Since the memory management unit (MMU) on general-purpose CPUs supports the huge virtual address space, randomization could be a practical defense against memory corruption attacks. CPI (Kuznetsov et al., 2014) proposes a control flow hijacking mitigation by protecting the integrity of code pointers. In the course of that, CPI employs a safe stack, a lightweight variant of shadow stack, and protects it with randomization. However, recent information disclosure attacks (Göktaş et al., 2016; Oikonomopoulos et al., 2016; Gawlik et al., 2016) demonstrate that randomization-based approaches do not offer sufficient protection. To remedy this problem, Zieris and Horsch (2018) propose a leakage-resilient safe stack to counteract these information disclosure attacks. Such attacks drastically reduce the randomization entropy available for the shadow stack through thread/stack spraying (Göktaş et al., 2016) and memory oracles (Oikonomopoulos et al., 2016). To prevent such entropy reduction, the authors analyzed the behavior of these attacks and the design flaws of existing techniques (e.g., structural flaws, oversized stack, etc.). Based on this analysis, they provided a set of countermeasures, such as a dedicated memory pool and a minimized shadow stack.

Domain-based Memory Isolation. Modern processors generally provide domain-based memory protection primitives such as Intel MPK (Programming Guide, 2011) and ARM DACR (ARM, 2021)). ERIM (Vahldiek-Oberwagner et al., 2019) implements an in-process isolation framework using Intel MPK. It shows how sensitive user data can be protected effectively based on its fast domain switching. Moreover, it presents the usage of such a technique for an integrity-protected shadow stack implementation. Similar to ERIM, IskiOS (Gravani et al., 2021) retrofits the ERIM's protection into the kernel; it adapts Intel MPK, which is originally designed only for user use, to be leveraged in the kernel space by configuring kernel pages as user-mode pages. As in ERIM, it showcases the shadow stack integrity guarantee based on it.

User Page Protection. To mitigate ret2usr attacks (Kemerlis et al., 2014), most contemporary processors provide user page access prevention primitives (e.g., Intel SMAP (Intel, 2016) and ARM PAN (ARM, 2021)). ILDI (Cho et al., 2017) builds a kernel space memory isolation primitive based on ARM PAN and Load and Store Unprivileged (LSU) instructions. Based on this primitive, it demonstrates its usefulness by building an integrity-protected shadow stack in the kernel space. In a similar manner, SEIMI (Wang et al., 2020) proposes an in-process memory isolation scheme using Intel SMAP. To be more precise, it first escalates user processes to the privileged mode and configures sensitive memory as user-mode pages, thus preventing any access to the user-mode pages without turning off the SMAP protection. Moreover, the authors suggested shadow stack protection as one of its best use.

10. Conclusion

This paper introduces SuM, which offers an efficient and secure backward-edge control flow protection technique for ARM Cortex-M processors. SuM deploys a non-bypassable shadow stack and efficiently protects its integrity via selective masking—a novel intra-address space isolation primitive that uses a unique combination of an MPU and `FaultMask`. We implemented SuM based on the LLVM compiler framework and evaluated the prototype of SuM on the BEEBS and CoreMark

benchmark suites. The evaluation shows that SUM manifests low-runtime overhead of less than 3% on both of the benchmark suites. Additionally, even though SUM incurs 8.83% and 6.59% code size overhead on BEEBS and CoreMark, respectively, this overhead is anticipated to be much lower in practice. Consequently, SUM is anticipated to serve as a pragmatic method for backward-edge control flow protection in embedded systems.

CRediT authorship contribution statement

Wonwoo Choi: Conceptualization, Software, Writing – original draft. **Minjae Seo:** Investigation, Writing – review & editing. **Seongman Lee:** Investigation, Writing – review & editing. **Brent Byunghoon Kang:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2020R1A2C2101134).

Appendix A. Evaluation results

This section consists of evaluation results with detailed measurements (see Tables A.1-A.5). For Table A.5, the values represent the increment in the flash memory consumption from the corresponding component, whereas the values in Table A.4 and Table A.1(b) show the total flash memory consumption with the corresponding protection applied.

Table A.1
Runtime and flash memory overhead of SUM, SUM-SFI, and Silhouette on the CoreMark benchmark suite.

Workload	Baseline (ms)	SuM (ms)		Silhouette (ms)		SFI (ms)	
CoreMark	54884.48	56326.51	2.63%	57833.41	5.37%	61813.81	12.63%

(a) Runtime Overhead

Workload	Baseline (bytes)	SuM (bytes)		Silhouette (bytes)		Silhouette-without-const (bytes)		SFI (bytes)		SFI-without-const (bytes)	
CoreMark	26508	28254	6.59%	30330	14.42%	30104	13.57%	31854	20.17%	31160	17.55%

(b) Flash Memory Overhead

Table A.2
Runtime overhead of SUM, SUM-SFI, and Silhouette on the BEEBS benchmark suite.

Workload	Baseline (ms)	SuM (ms)		Silhouette (ms)		SFI (ms)	
bubblesort	5902.53	5902.74	0.00%	7354.87	24.61%	7707.79	30.58%
ctl-string	1195.02	1200.80	0.48%	1230.92	3.00%	1213.52	1.55%
cubic	58638.90	60293.60	2.82%	63838.17	8.87%	60348.30	2.92%
dijkstra	75488.38	79840.89	5.77%	78172.04	3.56%	87081.29	15.36%
edn	3617.72	3694.37	2.12%	3715.44	2.70%	4123.39	13.98%
fasta	35077.26	35086.36	0.03%	35081.26	0.01%	35092.55	0.04%
fir	23975.98	23987.36	0.05%	23932.86	-0.18%	24928.13	3.97%
frac	11444.64	12225.53	6.82%	13202.32	15.36%	12140.56	6.08%
huffbench	33988.26	35190.06	3.54%	36874.03	8.49%	40077.80	17.92%
levenshtein	6122.72	6280.78	2.58%	7355.09	20.13%	7088.24	15.77%
matmult-int	4948.82	4999.86	1.03%	5092.17	2.90%	5456.74	10.26%
nbody	231309.90	236593.11	2.28%	243944.65	5.46%	236557.81	2.27%
ndes	3346.92	3388.29	1.24%	3398.58	1.54%	3446.27	2.97%
nettle-aes	7931.83	8374.45	5.58%	8001.22	0.87%	9456.42	19.22%
picojpeg	73217.46	75890.91	3.65%	80628.35	10.12%	95497.64	30.43%
qrduino	78783.96	80205.18	1.80%	81849.46	3.89%	85413.61	8.41%
rijndael	73759.36	74287.88	0.72%	76548.22	3.78%	87680.80	18.87%
sglib-dllist	2669.85	2737.83	2.55%	2676.32	0.24%	3223.43	20.73%
sglib-listinsertsort	2795.66	2796.64	0.03%	2769.99	-0.92%	2906.54	3.97%
sglib-listsort	2231.70	2222.32	-0.42%	2228.00	-0.17%	2568.90	15.11%
sglib-queue	1974.20	2026.47	2.65%	2344.05	18.73%	2635.57	33.50%
sglib-rbtree	8968.64	9385.30	4.65%	10245.81	14.24%	10064.17	12.22%
slre	4250.94	4476.39	5.30%	4654.65	9.50%	4908.56	15.47%
sqrt	64131.32	67015.80	4.50%	70938.34	10.61%	66992.79	4.46%
st	38996.77	40161.01	2.99%	42336.03	8.56%	40192.91	3.07%
stb_perlin	5497.29	5545.72	0.88%	5474.60	-0.41%	5589.44	1.68%
trio-sscanf	1233.80	1335.67	8.26%	1361.72	10.37%	1516.36	22.90%
whetstone	222884.98	229147.73	2.81%	243126.81	9.08%	228731.19	2.62%
wikisort	197747.08	210506.50	6.45%	223322.53	12.93%	251378.03	27.12%
GEOMEAN			2.77%		6.97%		12.12%
MIN			-0.42%		-0.92%		0.04%
MAX			8.26%		24.61%		33.50%

Table A.3
Runtime overhead of the individual components of SUM on the BEEBS benchmark suite.

Workload	Baseline (ms)	Shadow Stack (ms)		Selective Masking (ms)		CFI (ms)	
bubblesort	5902.53	5902.50	0.00%	5902.90	0.01%	5902.53	0.00%
ctl-string	1195.02	1192.87	-0.18%	1203.20	0.68%	1195.02	0.00%
cubic	58638.90	58825.87	0.32%	60057.75	2.42%	58469.12	-0.29%
dijkstra	75488.38	79803.80	5.72%	75525.71	0.05%	75490.01	0.00%
edn	3617.72	3693.64	2.10%	3618.49	0.02%	3617.81	0.00%
fasta	35077.26	35086.34	0.03%	35077.26	0.00%	35077.27	0.00%
fir	23975.98	23987.37	0.05%	23975.98	0.00%	23975.98	0.00%
frac	11444.64	11568.74	1.08%	11969.34	4.58%	11486.36	0.36%
huffbench	33988.26	35190.09	3.54%	33989.39	0.00%	33988.97	0.00%
levenshtein	6122.72	6268.52	2.38%	6132.17	0.15%	6122.55	0.00%
matmult-int	4948.82	4999.47	1.02%	4949.23	0.01%	4948.84	0.00%
nbody	231309.90	232370.21	0.46%	233507.54	0.95%	233386.12	0.90%
ndes	3346.92	3376.54	0.88%	3359.88	0.39%	3347.22	0.01%
nettle-aes	7931.83	8373.51	5.57%	7933.91	0.03%	7935.32	0.04%
picojpeg	73217.46	75227.83	2.75%	74151.81	1.28%	73227.80	0.01%
qrduino	78783.96	80190.19	1.78%	78794.52	0.01%	78784.19	0.00%
rijndael	73759.36	74225.86	0.63%	73853.50	0.13%	73760.35	0.00%
sglib-dllist	2669.85	2735.65	2.46%	2667.82	-0.08%	2669.65	-0.01%
sglib-listinsertsort	2795.66	2796.64	0.03%	2795.66	0.00%	2795.66	0.00%
sglib-listsort	2231.70	2222.32	-0.42%	2231.70	0.00%	2231.70	0.00%
sglib-queue	1974.20	2026.10	2.63%	1974.42	0.01%	1974.33	0.01%
sglib-rbtree	8968.64	9075.30	1.19%	9278.27	3.45%	8968.64	0.00%
slre	4250.94	4367.56	2.74%	4366.25	2.71%	4261.29	0.24%
sqrt	64131.32	65331.67	1.87%	66275.99	3.34%	64548.63	0.65%
st	38996.77	39127.81	0.34%	39717.92	1.85%	38993.57	-0.01%
stb_perlin	5497.29	5499.83	0.05%	5536.89	0.72%	5492.96	-0.08%
trio-sscanf	1233.80	1247.71	1.13%	1248.03	1.15%	1307.49	5.97%
whetstone	222884.98	223088.53	0.09%	227358.98	2.01%	223053.78	0.08%
wikisort	197747.08	201075.26	1.68%	201919.16	2.11%	202976.02	2.64%
GEOMEAN			1.43%		0.96%		0.36%
MIN			-0.42%		-0.08%		-0.29%
MAX			5.72%		4.58%		5.97%

Table A.4
Flash memory overhead of SUM, SUM-SFI, and Silhouette on the BEEBS benchmark suite.

Workload	Baseline (bytes)	SuM (bytes)		Silhouette (bytes)		Silhouette-without-const (bytes)		SFI (bytes)		SFI-without-const (bytes)	
bubblesort	14316	15794	10.32%	17006	18.79%	16900	18.05%	17670	23.43%	16948	18.39%
ctl-string	16056	17682	10.13%	18706	16.50%	18600	15.84%	19198	19.57%	18476	15.07%
cubic	40252	42470	5.51%	45254	12.43%	45148	12.16%	44866	11.46%	44144	9.67%
dijkstra	14820	16318	10.11%	17442	17.69%	17336	16.98%	18214	22.90%	17492	18.03%
edn	17496	18990	8.54%	19958	14.07%	19852	13.47%	20782	18.78%	20060	14.65%
fasta	13876	15354	10.65%	16166	16.50%	16060	15.74%	16738	20.63%	16016	15.42%
fir	16268	17706	8.84%	18498	13.71%	18392	13.06%	19054	17.13%	18332	12.69%
frac	16664	18346	10.09%	19618	17.73%	19512	17.09%	19798	18.81%	19076	14.47%
huffbench	17320	18922	9.25%	20490	18.30%	20384	17.69%	21694	25.25%	20972	21.09%
levenshtein	14616	16134	10.39%	18130	24.04%	18024	23.32%	17806	21.83%	17084	16.89%
matmult-int	16072	17554	9.22%	18458	14.85%	18352	14.19%	19102	18.85%	18380	14.36%
nbody	20264	21938	8.26%	23338	15.17%	23232	14.65%	23502	15.98%	22780	12.42%
ndes	17836	19346	8.47%	20794	16.58%	20688	15.99%	21270	19.25%	20548	15.21%
nettle-aes	25572	27146	6.16%	27966	9.36%	27860	8.95%	28766	12.49%	28044	9.67%
picojpeg	28672	30302	5.68%	33878	18.16%	33772	17.79%	36142	26.05%	35420	23.54%
qrduino	30480	31878	4.59%	35430	16.24%	35324	15.89%	36778	20.66%	36056	18.29%
rijndael	27208	29018	6.65%	30326	11.46%	30220	11.07%	32246	18.52%	31524	15.86%
sglib-dllist	14320	15870	10.82%	16638	16.19%	16532	15.45%	17446	21.83%	16724	16.79%
sglib-listinsertsort	13744	15210	10.67%	15974	16.23%	15868	15.45%	16570	20.56%	15848	15.31%
sglib-listsort	14096	15558	10.37%	16370	16.13%	16264	15.38%	17062	21.04%	16340	15.92%
sglib-queue	14844	16314	9.90%	17662	18.98%	17556	18.27%	18926	27.50%	18204	22.64%
sglib-rbtree	14292	15822	10.71%	16818	17.67%	16712	16.93%	17494	22.40%	16772	17.35%
slre	16986	18504	8.94%	19648	15.67%	19542	15.05%	20408	20.15%	19686	15.90%
sqrt	15628	17222	10.20%	18246	16.75%	18140	16.07%	18554	18.72%	17832	14.10%
st	21732	23446	7.89%	24790	14.07%	24684	13.58%	25058	15.30%	24336	11.98%
stb_perlin	19084	20706	8.50%	21806	14.26%	21700	13.71%	22058	15.58%	21336	11.80%
trio-sscanf	19024	21486	12.94%	22882	20.28%	22776	19.72%	24122	26.80%	23400	23.00%
whetstone	39080	41258	5.57%	44210	13.13%	44104	12.86%	43734	11.91%	43012	10.06%
wikisort	26380	28270	7.16%	31222	18.35%	31116	17.95%	33170	25.74%	32448	23.00%
GEOMEAN			8.83%		16.15%		15.57%		19.90%		15.92%
MIN			4.59%		9.36%		8.95%		11.46%		9.67%
MAX			12.94%		24.04%		23.32%		27.50%		23.54%

Table A.5
Flash memory overhead of the individual components of SUM on the BEEBS benchmark suite. In this table, the bytes represent the flash memory consumption increment due to each component.

Workload	Shadow Stack (bytes)		Selective Masking (bytes)		CFI (bytes)		Configuration (bytes)		ERP (bytes)	
bubblesort	424	2.96%	116	0.81%	284	1.98%	174	1.22%	464	3.24%
ctl-string	520	3.24%	128	0.80%	324	2.02%	174	1.08%	464	2.89%
cubic	1000	2.48%	200	0.50%	404	1.00%	174	0.43%	464	1.15%
dijkstra	440	2.97%	120	0.81%	284	1.92%	174	1.17%	464	3.13%
edn	432	2.47%	116	0.66%	292	1.67%	174	0.99%	464	2.65%
fasta	428	3.08%	112	0.81%	284	2.05%	174	1.25%	464	3.34%
fir	388	2.39%	112	0.69%	284	1.75%	174	1.07%	464	2.85%
frac	576	3.46%	152	0.91%	324	1.94%	174	1.04%	464	2.78%
huffbench	548	3.16%	120	0.69%	296	1.71%	174	1.00%	464	2.68%
levenshtein	464	3.17%	116	0.79%	284	1.94%	174	1.19%	464	3.17%
matmult-int	428	2.66%	116	0.72%	284	1.77%	174	1.08%	464	2.89%
nbody	580	2.86%	128	0.63%	312	1.54%	174	0.86%	464	2.29%
ndes	452	2.53%	120	0.67%	284	1.59%	174	0.98%	464	2.60%
nettle-aes	520	2.03%	116	0.45%	284	1.11%	174	0.68%	464	1.81%
picojpeg	528	1.84%	156	0.54%	292	1.02%	174	0.61%	464	1.62%
qrduino	316	1.04%	132	0.43%	296	0.97%	174	0.57%	464	1.52%
rijndael	748	2.75%	128	0.47%	296	1.09%	174	0.64%	464	1.71%
sglib-dllist	500	3.49%	112	0.78%	284	1.98%	174	1.22%	464	3.24%
sglib-listinsertsort	416	3.03%	112	0.81%	284	2.07%	174	1.27%	464	3.38%
sglib-listsort	412	2.92%	112	0.79%	284	2.01%	174	1.23%	464	3.29%
sglib-queue	420	2.83%	112	0.75%	284	1.91%	174	1.17%	464	3.13%
sglib-rbtree	436	3.05%	120	0.84%	320	2.24%	174	1.22%	464	3.25%
slre	456	2.68%	124	0.73%	284	1.67%	174	1.02%	464	2.73%
sqrt	508	3.25%	136	0.87%	312	2.00%	174	1.11%	464	2.97%
st	612	2.82%	136	0.63%	320	1.47%	174	0.80%	464	2.14%
stb_perlin	524	2.75%	132	0.69%	312	1.63%	174	0.91%	464	2.43%
trio-sscanf	612	3.22%	140	0.74%	1040	5.47%	174	0.91%	464	2.44%
whetstone	944	2.42%	212	0.54%	424	1.08%	174	0.45%	464	1.19%
wikisort	688	2.61%	144	0.55%	388	1.47%	174	0.66%	464	1.76%
GEOMEAN		2.76%		0.69%		1.79%		0.96%		2.56%
MIN		1.04%		0.43%		0.97%		0.43%		1.15%
MAX		3.49%		0.91%		5.47%		1.27%		3.38%

References

Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J., 2005. Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS).

Abbasi, A., Wetzels, J., Holz, T., Etalle, S., 2019. Challenges in designing exploit mitigations for deeply embedded systems. In: IEEE European Symposium on Security and Privacy (EuroS&P).

Almakhdhub, N.S., Clements, A.A., Bagchi, S., Payer, M., 2020. μRAI: securing embedded systems with return address integrity. In: Network and Distributed Systems Security Symposium (NDSS).

ARM, 2006. ARMv7-M architecture reference manual. https://developer.arm.com/documentation/ddi0403/latest/.

ARM, 2016. ARMv8-M architecture reference manual. https://developer.arm.com/documentation/ddi0553/latest.

ARM, 2021. ARM ARMv8-A architecture registers. https://developer.arm.com/documentation/ddi0595/2021-03/.

Bresch, C., Lysecky, R., Hély, D., 2020. BackFlow: backward edge control flow enforcement for low end ARM microcontrollers. In: Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE.

Burow, N., Zhang, X., Payer, M., 2019. SoK: shining light on shadow stacks. In: IEEE Symposium on Security and Privacy (S&P).

Carlini, N., Wagner, D., 2014. ROP is still dangerous: breaking modern defenses. In: 23rd USENIX Security Symposium (USENIX Security).

Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R., 2015. Control-flow bending: on the effectiveness of control-flow integrity. In: 24th USENIX Security Symposium (USENIX Security).

Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K., 2005. Non-control-data attacks are realistic threats. In: 14th USENIX Security Symposium (USENIX Security).

Cho, Y., Kwon, D., Paek, Y., 2017. Instruction-level data isolation for the kernel on ARM. In: Proceedings of the 54th Annual Design Automation Conference (DAC).

Clements, A.A., Almakhdhub, N.S., Saab, K.S., Srivastava, P., Koo, J., Bagchi, S., Payer, M., 2017. Protecting bare-metal embedded systems with privilege overlays. In: Symposium on Security and Privacy (S&P). IEEE.

Dang, T.H., Maniatis, P., Wagner, D., 2015. The performance cost of shadow stacks and stack canaries. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS).

Davi, L., Sadeghi, A.-R., Lehmann, D., Monrose, F., 2014. Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In: 23rd USENIX Security Symposium (USENIX Security).

Ding, R., Qian, C., Song, C., Harris, B., Kim, T., Lee, W., 2017. Efficient protection of path-sensitive control security. In: 26th USENIX Security Symposium (USENIX Security 17).

Du, Y., Shen, Z., Dharsee, K., Zhou, J., Walls, R.J., Criswell, J., 2022. Holistic control-flow protection on real-time embedded systems with Kage. In: 31st USENIX Security Symposium (USENIX Security 22).

EEMBC CoreMark - industry-standard benchmarks for embedded systems. http://www.eembc.org/coremark.

Evans, I., Fingeret, S., Gonzalez, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., Okhravi, H., 2015. Missing the point(er): on the effectiveness of code pointer integrity. In: IEEE Symposium on Security and Privacy (S&P).

Frassetto, T., Jauernig, P., Liebchen, C., Sadeghi, A.-R., 2018. IMIX: in-process memory isolation extension. In: 27th USENIX Security Symposium (USENIX Security).

Gawlik, R., Kollenda, B., Koppe, P., Garmany, B., Holz, T., 2016. Enabling client-side crash-resistance to overcome diversification and information hiding. In: Network and Distributed Systems Security Symposium (NDSS).

Göktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G., 2014. Out of control: overcoming control-flow integrity. In: IEEE Symposium on Security and Privacy (S&P).

Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., Portokalidis, G., 2014. Size does matter: why using gadget-chain length to prevent code-reuse attacks is hard. In: 23rd USENIX Security Symposium (USENIX Security).

Göktaş, E., Gawlik, R., Kollenda, B., Athanasopoulos, E., Portokalidis, G., Giuffrida, C., Bos, H., 2016. Undermining information hiding (and what to do about it). In: 25th USENIX Security Symposium (USENIX Security).

Gravani, S., Hedayati, M., Criswell, J., Scott, M.L., 2021. Fast intra-kernel isolation and security with IskiOS. In: Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID).

Hedayati, M., Gravani, S., Johnson, E., Criswell, J., Scott, M.L., Shen, K., Marty, M., 2019. Hodor: intra-process isolation for high-throughput data plane libraries. In: USENIX Annual Technical Conference (USENIX ATC).

Hu, H., Qian, C., Yagemann, C., Chung, S.P.H., Harris, W.R., Kim, T., Lee, W., 2018. Enforcing unique code target property for control-flow integrity. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS).

Intel, 2016. Intel 64 and Ia-32 architectures software developer's manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf.

Ismail, M., Yom, J., Jelesnianski, C., Jang, Y., Min Vip, C., 2021. Safeguard value invariant property for thwarting critical memory corruption attacks. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS).

Kawada, T., Honda, S., Matsubara, Y., Takada, H., 2021. TZmCFI: RTOS-aware control-flow integrity using TrustZone for ARMv8-M. Int. J. Parallel Program.

Kemerlis, V.P., Polychronakis, M., Keromytis, A.D., 2014. ret2dir: rethinking kernel isolation. In: 23rd USENIX Security Symposium (USENIX Security).

Kim, C.H., Kim, T., Choi, H., Gu, Z., Lee, B., Zhang, X., Xu, D., 2018. Securing real-time microcontroller systems through customized memory view switching. In: Network and Distributed System Security Symposium (NDSS).

Koning, K., Chen, X., Bos, H., Giuffrida, C., Athanasopoulos, E., 2017. No need to hide: protecting safe regions on commodity hardware. In: Proceedings of the Twelfth European Conference on Computer Systems (EuroSys).

Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D., 2014. Code-pointer integrity. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI).

Kwon, D., Shin, J., Kim, G., Lee, B., Cho, Y., Paek, Y., 2019. uXOM: efficient execute-only memory on ARM Cortex-M. In: 28th USENIX Security Symposium (USENIX Security).

Lattner, C., Adve, V., 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization (CGO). IEEE.

Lu, K., Song, C., Lee, B., Chung, S.P., Kim, T., Lee, W., 2015. ASLR-GUARD: stopping address space leakage for code reuse attacks. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS).

Microsoft, 2018. Data execution prevention. https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

Niu, B., Tan, G., 2014. Modular control-flow integrity. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

Niu, B., Tan, G., 2015. Per-input control-flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS).

Nyman, T., Ekberg, J.-E., Davi, L., Asokan, N., 2017. CFI CaRE: hardware-supported call and return enforcement for commercial microcontrollers. In: International Symposium on Research in Attacks, Intrusions, and Defenses (RAID). Springer.

Oikonomopoulos, A., Athanasopoulos, E., Bos, H., Giuffrida, C., 2016. Poking holes in information hiding. In: 25th USENIX Security Symposium (USENIX Security).

Packard, K., 2018. Picolibc. https://github.com/picolibc/picolibc.

Pallister, J., Hollis, S., Bennett, J., 2013. BEEBS: open benchmarks for energy measurements on embedded platforms. arXiv preprint. arXiv:1308.5174.

Pappas, V., Polychronakis, M., Keromytis, A.D., 2013. Transparent ROP exploit mitigation using indirect branch tracing. In: 22nd USENIX Security Symposium (USENIX Security).

Pomonis, M., Petsios, T., Keromytis, A.D., Polychronakis, M., Kemerlis, V.P., 2017. kr^x: comprehensive kernel protection against just-in-time code reuse. In: Proceedings of the Twelfth European Conference on Computer Systems (EuroSys).

Programming Guide, Intel® 64 and Ia-32 architectures software developer's manual, Volume 3B: System programming Guide, Part 2 (11) (2011) 0–40.

Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., Chen, B., 2010. Adapting software fault isolation to contemporary CPU architectures. In: 19th USENIX Security Symposium (USENIX Security).

Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., Boneh, D., 2004. On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS).

Shen, Z., Dharsee, K., Criswell, J., 2020. Fast execute-only memory for embedded systems. In: 2020 IEEE Secure Development (SecDev). IEEE, pp. 7–14.

Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G., 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In: 23rd USENIX Security Symposium (USENIX Security).

Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N.O., Sammler, M., Druschel, P., Garg, D., 2019. ERIM: secure, efficient in-process isolation with protection keys (MPK). In: 28th USENIX Security Symposium (USENIX Security).

Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L., 1993. Efficient software-based fault isolation. In: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP), pp. 203–216.

Wang, Z., Wu, C., Xie, M., Zhang, Y., Lu, K., Zhang, X., Lai, Y., Kang, Y., Yang, M., 2020. SEIMI: efficient and secure SMAP-enabled intra-process memory isolation. In: IEEE Symposium on Security and Privacy (S&P).

Xie, M., Wu, C., Zhang, Y., Xu, J., Lai, Y., Kang, Y., Wang, W., Wang, Z., 2022. CETIS: retrofitting intel CET for generic and efficient intra-process memory isolation. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS).

Yiu, J., 2015. ARMv8-M architecture technical overview. ARM white paper.

Yu, R., Del Nin, F., Zhang, Y., Huang, S., Kaliyar, P., Zakto, S., Conti, M., Portokalidis, G., Xu, J., 2022. Building embedded systems like it's 1996. In: Network and Distributed Systems Security Symposium (NDSS).

Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W., 2013. Practical control flow integrity and randomization for binary executables. In: IEEE Symposium on Security and Privacy (S&P).

Zhang, M., Sekar, R., 2013. Control flow integrity for COTS binaries. In: 22nd USENIX Security Symposium (USENIX Security).

Zhou, J., Du, Y., Shen, Z., Ma, L., Criswell, J., Walls, R.J., 2020. Silhouette: efficient protected shadow stacks for embedded systems. In: 29th USENIX Security Symposium (USENIX Security).

Zhou, J., Du, Y., Shen, Z., Ma, L., Criswell, J., Walls, R.J., 2021. Silhouette-compiler. https://github.com/URSec/Silhouette-Compiler.

Zieris, P., Horsch, J., 2018. A leak-resilient dual stack scheme for backward-edge control-flow integrity. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS).