
# uXOM: Efficient eXecute-Only Memory on ARM Cortex-M

**Donghyun Kwon, Jangseop Shin, and Giyeol Kim,** ***Seoul National University;*** **Byoungyoung Lee,** ***Seoul National University, Purdue University;*** **Yeongpil Cho,** ***Soongsil University;*** **Yunheung Paek,** ***Seoul National University***

https://www.usenix.org/conference/usenixsecurity19/presentation/kwon

**This paper is included in the Proceedings of the 28th USENIX Security Symposium.**

**August 14–16, 2019 • Santa Clara, CA, USA**

978-1-939133-06-9

**Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.**

# *u*XOM: Efficient eXecute-Only Memory on ARM Cortex-M

Donghyun Kwon[1,2] Jangseop Shin[1,2] Giyeol Kim[1,2]
Byoungyoung Lee[1,3] Yeongpil Cho[4] Yunheung Paek[1,2]

[1]*ECE, Seoul National University,* [2]*ISRC, Seoul National University*
[3]*Computer Science, Purdue University,* [4]*School of Software, Soongsil University*

{dhkwon, jsshin, gykim}@sor.snu.ac.kr,
{byoungyoung, ypaek}@snu.ac.kr, ypcho@ssu.ac.kr

## Abstract

Code disclosure attacks are one of the major threats to a computer system, considering that code often contains security sensitive information, such as intellectual properties (e.g., secret algorithm), sensitive data (e.g., cryptographic keys) and the gadgets for launching code reuse attacks. To stymie this class of attacks, security researchers have devised a strong memory protection mechanism, called eXecute-Only-Memory (XOM), that defines special memory regions where instruction execution is permitted but data reads and writes are prohibited. Reflecting the value of XOM, many recent high-end processors have added support for XOM in their hardware. Unfortunately, however, low-end embedded processors have yet to provide hardware support for XOM.

In this paper, we propose a novel technique, named *u*XOM, that realizes XOM in a way that is secure and highly optimized to work on Cortex-M, which is a prominent processor series used in low-end embedded devices. *u*XOM achieves its security and efficiency by using special architectural features in Cortex-M: *unprivileged memory instructions* and an *MPU*. We present several challenges in making XOM non-bypassable under strong attackers and introduce our code analysis and instrumentation to solve these challenges. Our evaluation reveals that *u*XOM successfully realizes XOM in Cortex-M processor with much better efficiency in terms of execution time, code size and energy consumption compared to a software-only XOM implementation for Cortex-M.

## 1 Introduction

When it comes to the security of a computing system, the protection of the code running on the system should be of top priority because the code defines security critical behaviors of the system. For instance, if attackers are able to modify existing code or inject new code, they may place the victim system under their control. Fortunately, code injection attacks nowadays can be mitigated by simply enforcing the well-known security policy, W⊕X. Since virtually all processors today are equipped with at least five basic memory permissions: read-write-execute (RWX), read-write (RW), read-execute (RX), read-only (RO) and no-access (NA), W⊕X can be efficiently enforced in hardware for a memory region solely by disabling RWX.

However, even if attackers are not able to modify the system's code, the system can still be threatened by *disclosure attacks* that attempt to read part of or possibly the entire code. Because code often contains intellectual properties (IPs) including core algorithms and sensitive data like cryptographic keys, disclosure attacks severely damage the security of victim systems by exposing critical information to unauthorized users. Even worse, disclosure attacks can be abused by attackers to launch *code reuse attacks* (CRAs), which allow the attacker to perform adversarial behaviors without modifying its code contents. It has been shown that attackers who can see the instructions in the code may launch a CRA wherein they craft a malicious code sequence by chaining the existing code snippets scattered around the program binary [34].

In order to prevent disclosure attacks, *eXecute-Only-Memory* (XOM) has been a core security mechanism of various countermeasure techniques [6–8, 13, 16, 17, 31, 37]. XOM is a strong memory protection mechanism that defines a special memory region where only instruction executions are allowed, and any attempts for instruction reads or writes are prohibited. Thus, as long as sensitive information such as IPs and the code contents are stored inside the region protected by XOM, developers are in principle able to prevent direct exposure of the code content as well as the code layout. This simple but tangible security benefit of XOM has led several researchers to propose hardware-assisted XOM on various architectures. For example, some have proposed an architecture that implements XOM by encrypting executable memory and decrypting instructions only when they are loaded [24]. However, since their approach mostly imposes significant changes and overhead on the underlying hardware, it cannot

Donghyun Kwon has been affiliated with Electronics and Telecommunications Research Institute (ETRI) since March 2019.

Corresponding authors are Yeongpil Cho and Yunheung Paek.

be adopted readily by the processor vendors for their existing products. Instead, many vendors opt for a less drastic approach that simply augments the basic memory permissions with the new execute-only (XO) permission [8, 10].

As of today, many high-end processors provide XOM capabilities by supporting augmented memory permissions. Consequently, by taking benefits from the hardware support for XOM, low-cost security solutions have been built to mitigate real attacks [8, 10, 13, 16]. However, these security benefits are confined to computing systems for general applications since the XO permission is only available in relatively high-end processors targeting general-purpose machines such as servers, desktops and smartphones. More specifically, applications running on tiny embedded devices cannot enjoy such benefits because only the basic memory permissions (not XOM) are supported in their target processors, which are primarily intended for use in low-cost, low-power computations. As one example of such processors that hardware-level XOM is not built into, we have the ARM *Cortex-M* series, which are prominent processors adopted by numerous low-cost computing devices today [38].

Fortunately, researchers have demonstrated that software fault isolation (SFI) techniques can be used to thwart these prevalent attacks without hardware-level XOM [7, 31]. They are purely software techniques, and thus are able to cope with any types of processors regardless of the underlying architectures. However, the drawback we observed is that SFI-based XOM techniques perform less optimally on certain types of processors, including Cortex-M in particular. More importantly, such techniques can even be circumvented, leading to critical security issues (refer to § 6.4). Motivated by this observation, this paper proposes a novel technique, called *u*XOM, to realize XOM in a way that is secure and highly optimized to work on Cortex-M processors. Since performance is a pivotal concern of tiny embedded devices such as Cortex-M, efficiency must be the most important objective of any technique targeting these low-end processors. To achieve this objective, *u*XOM leverages a special type of instructions, called *unprivileged loads/stores*, provided by the instruction set architecture for ARM Cortex-M. In an ARM-based system, memory can be divided into two classes of regions according to privilege levels: *non-privileged* and *privileged* memory regions. Unprivileged loads/stores can only access non-privileged memory regions, irrespective to the processor's current privilege level (either in a privileged or non-privileged). On the contrary, ordinary loads/stores are permitted to access privileged regions as long as they are executed under the privileged level. This striking difference between unprivileged and ordinary load/store instructions is the key enabler of our technique.

By capitalizing on this difference, we also need to exploit a unique style of running embedded software on the processors to achieve this ultimate goal of *u*XOM. In computing systems, software entities are typically assigned certain privileges during execution. For instance, user applications run as unprivileged, and the OS kernel as privileged. In practice, however, applications and the kernel in tiny embedded devices are designed to operate with the same privilege level [12, 21]. This is because these embedded systems are typically given real-time constraints, and the privilege mode switching involved in user-kernel privilege isolation is considered very expensive [21]. For the goal of *u*XOM stated above, we utilize these unique architectural characteristics of Cortex-M processors. More specifically, *u*XOM converts all memory instructions into unprivileged ones and sets the code region as privileged. As a result, converted instructions cannot access code regions, thereby effectively enforcing the XO permission onto the code regions. Since the processor is running with privileged level, code execution is still allowed without any permission error.

However, in order to actually realize *u*XOM, we need to tackle the problem that some memory instructions cannot be changed into unprivileged memory instructions. For example, memory instructions accessing critical system resources, such as an interrupt controller, a system timer and a Memory Protection Unit (MPU), should not be converted. Accesses to these resources always require privilege, so the program will crash if instructions accessing these resources are converted to unprivileged ones. In addition, load/store exclusive instructions, which are the special memory instructions for exclusive memory access, do not have unprivileged counterparts. For these instructions, there is no way to implement the intended functionality with unprivileged memory instructions. Therefore, we should analyze the code thoroughly to find these instructions and leave them as the original instructions.

Unfortunately, these unconverted memory instructions can be exploited by attackers to subvert *u*XOM. For example, if the attackers manage to execute these instructions by altering the control flow, they may bypass *u*XOM by (1) turning off the MPU protection or (2) reading the code directly. To prevent such attacks, the unconverted memory instructions need to be instrumented with verification routines to ensure that each memory access using these instructions does not break *u*XOM's protection. However, the attackers can still bypass the verification routines and directly execute the problematic memory instructions. To handle this challenge, we have devised the *atomic verification* technique that virtually enables memory instructions to be executed atomically with the verification routine, thereby preventing potential attackers from executing the memory instructions without passing the verification.

Another important problem *u*XOM needs to handle is that the attackers can alter control flow to execute *unintended* instructions, which may result from unaligned execution of 32-bit Thumb instructions or execution of the data embedded inside the code region [4]. Among the unintended instructions, attackers may find useful instructions for bypassing *u*XOM, such as ordinary memory instructions. To mitigate this attack vector, *u*XOM analyzes the code to find all potentially harm-

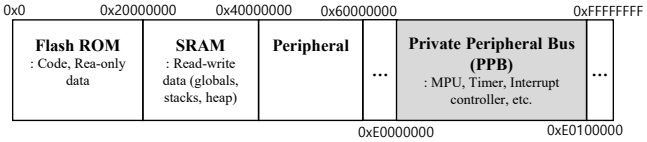



Figure 1: System address map for ARMv7-M [18]

ful unintended instructions and replaces them with alternative instruction sequences that have an equivalent function but do not contain any exploitable unintended instructions.

Built upon LLVM compiler and Radare2 binary analysis framework [32], *u*XOM automatically transforms every software component (i.e., real-time operating systems (RTOSs), the C standard library, and the user application) into a *u*XOM-enabled binary. Currently, *u*XOM supports processors based on ARMv7-M architecture, including Cortex-M3/4/7 processors. To evaluate *u*XOM, we experimented on an Arduino Due board, which ships with a Cortex-M3 processor. Our experiment confirms that *u*XOM works efficiently, empowered with the optimized use of the underlying hardware features. In particular, *u*XOM incurs only 15.7%, 7.3% and 7.5% overhead for code size, execution time and energy, while SFI-based XOM incurs overhead of 50.8%, 22.7%, and 22.3%, respectively. To demonstrate the compatibility of *u*XOM with other XOM-based security solutions, we discuss two use cases of *u*XOM: secret key protection and CRA defense. We implemented and evaluated the second use case, the CRA defense. Even when the CRA defense is applied on top of *u*XOM, it shows only moderate performance overhead, which is 19.3%, 8.6% and 9.7% for code size, execution time and energy, respectively.

The remainder of this paper is organized as follows. § 2 provides the background information. § 3 explains the threat model and assumptions. § 4 and § 5 describe the approach and design of *u*XOM, respectively. § 6 provides experimental results for *u*XOM and its use cases. § 7 presents several discussions regarding *u*XOM, and § 8 explains related works. § 9 concludes the paper.

## 2 Background

Cortex-M(3/4/7) processors targeted in this paper implement the ARMv7-M architecture, the microcontroller ('M') profile of the ARMv7 architecture, which features low-latency and highly deterministic operation for embedded systems. In this section, we give background information on the key architectural features of ARMv7-M that are required to understand the design and implementation of *u*XOM.

### 2.1 ARMv7-M Address Map and the Private Peripheral Bus (PPB)

ARMv7-M does not support memory virtualization and the regions for code, data, and other resources are fixed at specific address ranges. Figure 1 shows the system address map for ARMv7-M architecture. The first 0.5 GB (0x0-0x20000000) is the region where the flash ROM is typically mapped. Code and read-only data are placed here. The memory range 0x20000000-0x40000000 is the SRAM region where read-write data (globals, stack, and heap) are placed. Devices only use a small subset of each region; our test platform (SAM3X8E) has 512KB of flash and 96KB of SRAM. The memory range 0x40000000-0x60000000 is where device peripherals, such as GPIO and UART, are mapped. The 1 MB memory region ranging from 0xE0000000 to 0xE00FFFFF is the PPB region. Various system registers for controlling system configuration and monitoring system status, such as the system timer, the interrupt controller and the MPU, are mapped in this region. The PPB differs from the other memory regions of the system in that only privileged memory instructions are allowed to read from or write to the region. Generally, access permissions for memory regions can be configured through the MPU which we describe in detail below. However, the access permission for the PPB is fixed and even the MPU cannot override the default configuration.

### 2.2 Memory Protection Unit (MPU)

The MPU provides a memory access control functionality for Cortex-M processors. The biggest difference between the MPU and the Memory Management Unit (MMU) equipped in high-end processors is that the MPU does not provide memory virtualization and thus the access control rules are applied on the physical address space. Depending on the setting of the MPU's memory-mapped registers between 0xE000ED90 and 0xE000EDEC, a limited number (typically 8 or 16) of possibly overlapping regions can be set up, each of which is defined by the base address and the region size. Each region defines separate access permissions for privileged and non-privileged access through the combination of eXecute-Never (XN)-bit and Access Permission (AP)-bits. The available permission settings are RWX, RW, RX, RO, and NA, but in any case, unprivileged access is granted the same or more restrictive permission than privileged accesses. For example, when RO permission is given to a privileged access, unprivileged access can only have NA or RO permissions. If two or more regions have overlapping ranges, the access permission for the higher-numbered region takes effect. For access to memory ranges not covered by any region, it can be configured to always generate a fault or to follow the default access permission, which depends on the specific processor implementation. It is important to note that the read permission should be included in order for the memory region to be executable. This is the reason that XOM cannot be implemented simply by configuring the MPU in Cortex-M processors.

### 2.3 Unprivileged Loads/Stores

The ARMv7-M architecture only supports a thumb instruction set, which is a variable-length instruction set including a mix of traditional 16-bit thumb instructions and 32-bit instructions introduced in Thumb-2 technology. The unprivileged

loads/stores are special types of memory access instructions provided in the instruction set architecture [18]. The main distinction of these instructions is that they always perform memory accesses as if they are executed as unprivileged regardless of the current privilege mode. Thus, memory accesses using these instructions are regulated by the MPU's permission setting for unprivileged accesses. Unprivileged loads/stores are only available in 32-bit encoding and only have immediate-offset addressing mode. They do not support exclusive memory access. They are distinguished by the common suffix 'T' (e.g., LDRT and STRT).

### 2.4 Exception Entry and Return

An exception is a special event indicating that the system has encountered a specific condition that requires attention. It typically results in a forced transfer of control to a special software routine called an exception handler. On ARMv7-M, the location of the exception handlers corresponding to each exception are specified in the vector table pointed to by the Vector Table Offset Register (VTOR). Note that unlike the other ARMv7 profiles, the ARMv7-M has introduced a hardware mechanism that automatically stores and restores core context data (in particular, Program Status Register (xPSR), return address[1], lr, r12, r3, r2, r1 and r0) on the stack upon exception entry and return. The ARMv7-M profile also exhibits an interesting feature where an exception return occurs when a unique value of EXC_RETURN (e.g., 0xFFFFFFF1) is loaded into the pc via memory load instructions, such as POP, LDM and LDR, or indirect branch instructions, such as BX. Another thing to note about the exception handling in ARMv7-M is that different stack pointer (sp) can be used before and after the exception. ARMv7-M provides two types of sp, called main sp and process sp. The exception handler can only use main sp but the non-handler code can choose which of the two sps to use. The type of stack pointer being currently used is internally managed through CONTROL register, so that stack pointers are always represented as sp in the binary regardless of its actual type.

## 3 Threat Model and Assumptions

Several conditions must be met to realize *u*XOM. First, the target processor must support the MPU and the unprivileged load/store instructions. We also assume that the target devices run standard bare-metal software in which all included software components, such as applications, libraries, and an OS, share a single address space. Notably, we assume that the entire software executes at a privileged level as mentioned in § 1.

Next, we define the capabilities of an attacker. We assume that attackers are only capable of launching software attacks at runtime. We do not consider offline attacks on firmware images, such as disassembling, manipulating, or replacing the firmware, because we believe that these attacks can be thwarted by orthogonal techniques such as code encryption or signing. We also leave hardware attacks, such as bus probing [9] and memory tampering [22] out of consideration. However, we believe that our attackers are still strong enough to jeopardize the security of the target devices. The bare-metal software installed in the device is considered benign but internally holds software vulnerabilities, so that the attackers may exploit the vulnerabilities and ultimately have arbitrary memory read and write capability. With such a strong memory access capability, attackers can access any memory region including code, stack, heap and even the PPB region for system controls. They can also subvert control flow by manipulating function pointers or return addresses. We do not trust any software components, including the exception handlers. Event-driven nature of tiny embedded systems signifies that exception handlers can take a large portion of embedded software components [14], so we cannot just assume the security of these handlers. Thus, we assume that attackers can trigger a vulnerability inside the exception handler and manipulate any data including the cpu context saved on exception entry.

## 4 Approach and Challenges

*u*XOM aims to provide XO permission, which enables effective protection against disclosure attacks for code contents, for commodity bare-metal embedded systems based on the Cortex-M processor. *u*XOM tries to minimize the performance penalty by utilizing hardware features, such as unprivileged memory instructions and the MPU provided by Cortex-M processors. Ideally, *u*XOM converts *all* memory instructions into unprivileged ones. It then configures the MPU upon system boot to set code regions to RX for privileged access and NA for unprivileged access. It also sets the other memory regions (i.e., data regions) to non-executable for both privileged and unprivileged accesses. After the configuration, *u*XOM executes code as privileged. All the converted memory instructions (i.e., unprivileged memory instructions) are allowed to access the data regions in the same way as before. However, these instructions are prohibited from accessing the code region and the PPB region in which the MPU and VTOR are located that are essential for the security of *u*XOM (see the blue arrows in Figure 2). This is because these regions are set to the NA memory permission for unprivileged accesses. As all of the memory instructions have been converted to unprivileged ones, code disclosure attacks are effectively thwarted. In addition, *u*XOM by default enforces W⊕X policy that prevents code execution from writable regions. Therefore, any attempt to inject ordinary memory instructions for code disclosure is blocked as well.

**Challenges.** The basic principle of *u*XOM is simple and intuitive as described above. To realize *u*XOM in practice, however, we have to overcome some challenges to build a system that works for real programs and cannot be bypassed by any means. We summarize the challenges of realizing

[1] the value of the program counter (pc) at the moment of the exception

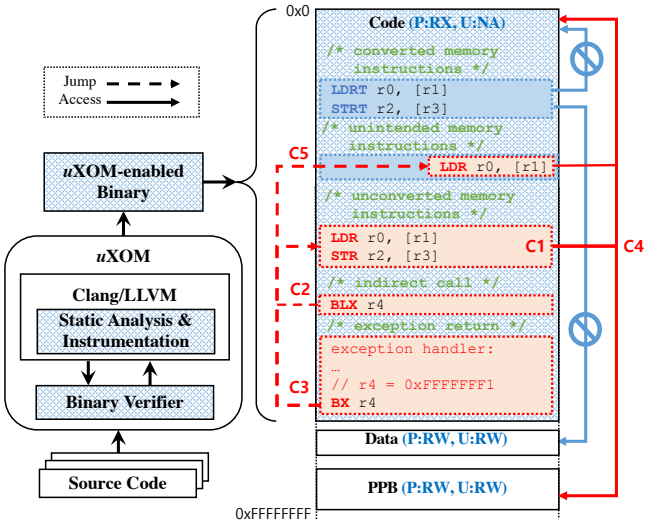



Figure 2: *u*XOM approach

*u*XOM as follows.

- **C1. Unconvertible memory instructions:** To implement *u*XOM, we initially tried to convert all memory instructions into unprivileged ones. However, this naïve attempt will be unsuccessful because unprivileged memory instructions do not support the exclusive memory access that is mainly utilized to implement lock mechanisms, and they cannot access the PPB region to which accesses must be privileged regardless of the MPU configuration. Therefore, we need to thoroughly analyze the entire code, find all these unconvertible instructions, and leave those instructions as the original types. However, these unconverted loads/stores in the program binary resulted in the other challenges, **C2**, **C3** and **C4**.
- **C2. Malicious indirect branches:** In § 3, we assumed that attackers are capable of altering the control flow at runtime by manipulating function pointers or return addresses. Therefore, attackers can deliberately jump to the unconverted loads/stores and exploit them. Unlike unprivileged loads, the unconverted ones can access the code region. Thus, the attackers are now able to read the code without a permission fault. Furthermore, the attackers can also use the unconverted stores to manipulate memory-mapped system registers in the PPB. For example, they can configure the MPU to enable unprivileged access to the code region, completely neutralizing the protection offered by *u*XOM.
- **C3. Malicious exception returns:** This challenge is similar to **C2** in that attackers can hijack control flow and eventually exploit the unconverted loads/stores to thwart *u*XOM. As explained in § 2.4, Cortex-M employs a hardware-based context save and restore mechanism for fast exception entry and return. The problem is that as the context is stored in the stack, attackers can exploit a vulnerability while in the exception handling mode to corrupt any context on the stack. In particular, the context includes a return address that represents the program point at the moment the exception is taken. If the attackers corrupt the return address and then trigger an exception return by assigning `EXC_RETURN` value to the `pc`, they will be able to execute any instruction in the program including the unconverted loads/stores.
- **C4. Malicious data manipulation:** As stated in § 3, the attackers can perform arbitrary memory read/write, and as a result, they have full control over all kind of program data, such as globals, heap objects, and local variables on the stack. With such control, they can exploit the unconverted loads/stores even while following a legitimate control flow. For example, they can call a MPU configuration function with a crafted argument to neutralize *u*XOM by compromising the necessary memory access permissions.
- **C5. Unintended instructions:** An attacker capable of manipulating control flow may be able to compromise *u*XOM by executing unintended instructions that are not found at compile-time. Concretely, Cortex-M processors targeted in this work support Thumb-2 instruction set architecture [18] that intermixes 16-bit and 32-bit width instructions with 16-bit alignment. Therefore, the attackers can execute unintended instructions by jumping into the middle of a 32-bit instruction. The attackers can also execute unintended instructions through immediate values embedded in code, whose bit-patterns can coincidentally be interpreted as a valid instruction.

# 5 *u*XOM

In this section, we describe the comprehensive details of *u*XOM. We first explain the basic design of *u*XOM for realizing the XO permission (§ 5.1). We then discuss our techniques for overcoming the challenges **C1-C5** (§ 5.2). Next, we present the optimizations applied to reduce performance penalty imposed by *u*XOM (§ 5.3). Lastly, we perform a security analysis to demonstrate that *u*XOM contains no security hazard (§ 5.4).

## 5.1 Basic Design

Before digging into the design details, we briefly describe how *u*XOM works on the system. As illustrated in Figure 2, *u*XOM is implemented as a compiler pass in the LLVM framework and a binary verifier. During compilation, *u*XOM performs static analyses and code instrumentation to generate a *u*XOM-enabled binary (i.e., firmware). Now, when the binary is flashed on to the board and the system boots, *u*XOM automatically enforces the XO permission on the running code.

### 5.1.1 Instruction Conversion

As RWX or RX is a mandatory permission for code execution on ARMv7-M, executable code regions are always readable and, as a result, are subject to disclosure attacks. Unfortunately, we cannot omit the read permission to imple-

| Case | Original Instruction | Converted Instructions |
|---|---|---|
| 1 | LDR rt, [rn, #imm5] | LDRT rt, [rn, #imm8] |
| 2 | LDR rt, [rn, #imm12] | (ADD rx, rn, #imm12)<br>LDRT rt, [rx, (#imm8)] |
| 3 | LDR rt, [rn, #-imm8] | SUB rx, rn #imm8<br>LDRT rt, [rx] |
| 4 | LDR rt, [rn, #+/-imm8]!<br>(pre-indexed) | ADD/SUB rx, rn, #imm12<br>LDRT rt, [rx] |
| 5 | LDR rt, [rn], #+/-imm8<br>(post-indexed) | LDRT rt, [rn]<br>ADD/SUB rx, rn, #imm12 |
| 6 | LDR rt, [rn, rm] | ADD rx, rn, rm<br>LDRT rt, [rx] |
| 7 | LDRD rt, rt2, [rn, #+/-imm8] | (ADD/SUB rx, rn, #imm8)<br>LDRT rt, [rx, (#imm8)]<br>LDRT rt2, [rx, (#imm8)+4] |

Table 1: Basic instruction conversion (only shown for load word instruction)

ment XOM because the read permission is required for the processor to fetch instructions from memory. Therefore, our strategy for XOM is to deprive all memory instructions of the access capability for code regions. Briefly put, we convert the memory instructions into unprivileged ones and set the code regions to be accessible only with a privileged manner.

Converting the type of the memory instruction may seem to be a trivial task, but not all memory instructions can be readily converted as unprivileged. The unprivileged loads/stores only support one addressing mode with a base register and an immediate offset which must be positive and fit within 8 bits. On the other hand, the original memory instructions vary in addressing modes, such as register-offset addressing and pre/post-indexed addressing, which updates the base register. Also, there are unprivileged counterparts to the load/store byte and load/store halfword instructions, but there are no corresponding unprivileged instructions for load/store dual (`LDRD`/`STRD`) and load/store multiple (`LDM`/`STM`), which respectively load/store two or multiple registers. To correctly convert all the memory instructions while preserving the program semantics, we sometimes need extra instructions.

Table 1 summarizes the conversions we apply to different types of load instructions. Cases 3-6 always need an extra `ADD` or `SUB` instruction for calculating the memory address. We omit an extra instruction for other cases if we can fit the immediate in the unprivileged instruction. Note that we may need an extra register for storing the calculated address if `rn` is used again in other instructions. We implement our conversion before the register allocation phase so that we do not have to worry about the physical registers and let the compiler choose the best register for the temporary results. `LDM`/`STM` instructions are not shown in the table because they only appear during an optimization pass after register allocation. Therefore, when the optimization pass tries to create `LDM`/`STM` instructions, we disable the optimization to prevent the generation of those instructions.

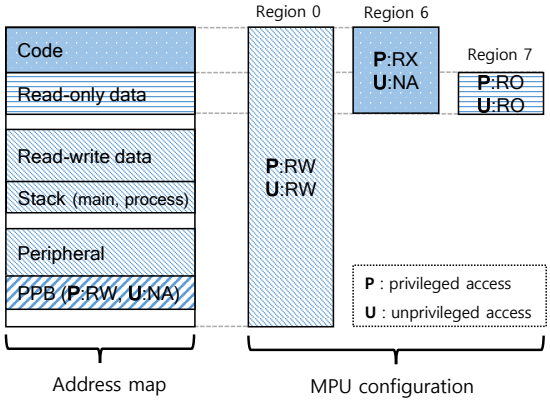



Figure 3: *u*XOM-specific memory permission. Unlabeled regions (white-colored regions) in the address map indicate the unused regions where the memory access generates data abort. The PPB region has a default memory permission (P:RW, U:NA) regardless of the MPU configuration.

### 5.1.2 Permission Control

In order for the XO permission based on the unprivileged load/store instructions to take effect, *u*XOM has to configure the MPU to enforce certain memory access permissions. Figure 3 shows the default MPU configuration for *u*XOM. Recall that when multiple regions overlap, the permission setting for the higher-numbered region is applied. We create Region 0 covering the entire address space with RW permission for both privileged and unprivileged modes. This is needed to allow unprivileged instructions to access the SRAM and the peripheral region. Otherwise, unprivileged access to those regions is not permitted due to the processor's default permission setting. We assign several higher-numbered regions to *u*XOM protection. (Here, we assumed that the number of MPU regions is 8.) Region 6 covers the entire flash region and assigns RX for privileged accesses and NA for unprivileged accesses. Since flash also contains read-only data, we configure Region 7 to let the unprivileged load instruction access the read-only data. To determine the base and size of this region, we need to know the size of the read-only data. To do this, we first compile, find out the read-only data size and generate an include file that is fed back into the MPU configuration code. The linker script is also modified to take this information and place the read-only data appropriately. The configurations are done in the early stage of the reset handler, which is called upon processor reset. In this way, the *u*XOM-specific permission is activated at the early stage of the system boot before attackers can seize control of the system.

## 5.2 Solving the Challenges

So far we have explained the basic design of *u*XOM for activating the XO permission. In the following, we describe how *u*XOM addresses the challenges presented in § 4.

| Instruction Type | Verification Details |
|---|---|
| Ordinary stores (STR) | if $\text{Target}_{address}$ points to MPU,<br>$\text{Target}_{value}$ must not violate *u*XOM-specific memory permissions.<br>else if $\text{Target}_{address}$ points to VTOR,<br>$\text{Target}_{value}$ must have one of the valid VTOR values.<br>else,<br>$\text{Target}_{address}$ must point to the PPB region excluding MPU region and VTOR region |
| Exclusive stores (STREX) | $\text{Target}_{address}$ must not point to the PPB region. |
| Ordinary loads (LDR)<br>Exclusive loads (LDREX) | $\text{Target}_{address}$ must not point to the code region. |

Table 2: Verification details by the type of unconverted memory instructions. $Target_{address}$ denotes the memory address accessed by load/store instructions and $Target_{value}$ denotes the value to be written by the store instructions.

#### 5.2.1 Finding Unconvertible Memory Instructions

Unprivileged memory instructions do not provide exclusive memory accesses and they cannot access the PPB region. As stated in **C1**, therefore, we need to identify the memory instructions that must not be converted to unprivileged ones and leave them as they are. We simply exclude exclusive memory loads/stores (e.g., LDREX and STREX) from the conversion candidate. We perform compiler analysis to find loads/stores accessing the PPB. Our analysis of the code base reveals that accesses to the PPB involve calculating the base address from a hard coded address pointing to the PPB region. This is consistent with the claims made in previous work [12]. We conduct a similar backward slicing technique to track how the base address of each memory instruction is calculated. If its address is a constant with the value corresponding to the PPB region, or if it is calculated by adding some offset to that constant value, we identify it as an access to the PPB region and leave it as an original form. For our test platform, intra-procedural analysis suffices to identify all PPB accesses. If a PPB address is passed through a function argument and used in a memory access, we can manually identify those particular cases and add annotations to prevent the compiler from converting the memory instructions as done in previous work [12]. Fortunately, most PPB accesses tend to be performed by the hardware abstraction layer (HAL) provided by the device manufacturer, so no significant amount of annotations are required to complement the static analysis.

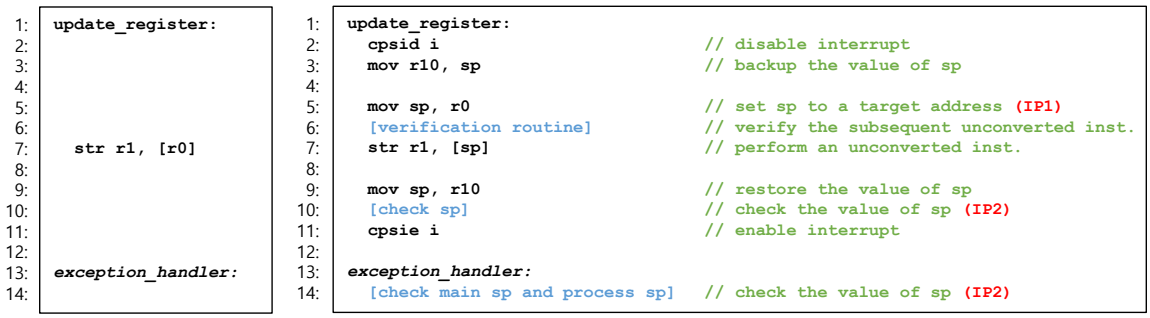
#### 5.2.2 Atomic Verification Technique

Our solution to deal with **C1** is necessary but may endanger the system. The problem is that, as stated in **C2**, **C3** and **C4**, the strong attackers assumed in § 3 can easily exploit the unconverted instructions to neutralize *u*XOM. To address this problem, we devise a *atomic verification* technique inspired by the concept of the reference monitor [15, 35]. The key of our technique is to verify memory accesses by the unconverted loads/stores. More specifically, it inserts a routine that performs verification as described in Table 2 before every unconverted load/store so that we can confirm whether or not the instruction tries to access code regions or manipulate system configuration necessary for *u*XOM, such as *u*XOM-specific memory permission (solve **C4**). At this point, however, the inserted verification may be bypassed by the attackers who can divert control flow. To prevent this, therefore, the technique enforces the atomic execution of the instruction sequence composed of the verification routine and the following untrusted load/store instruction, ensuring that the attackers cannot execute the unconverted loads/stores without a proper verification (solve **C2** and **C3**). Our basic strategy for atomic verification is to (1) allocate a dedicated register as a base register of every unconverted load/store, and then (2) enforce the following two invariant properties regarding the dedicated register.

- **Invariant 1**: The dedicated register must be set to a target address of each unconverted load/store immediately before the associated verification routine. The set value will be maintained only during the execution of the atomic instruction sequence due to **Invariant 2**.
- **Invariant 2**: The dedicated register must hold a non-harmful address (i.e., not a code or the PPB address) when the atomic instruction sequence is not executed.

Now, the accessible memory of the unconverted loads/stores is limited by the value of the dedicated register, which is used as their base register. **Invariant 1** allows the unconverted loads/stores to be executed for their original purpose (e.g., access to the PPB) only through the atomic instruction sequence with a verification. Also, **Invariant 2** prevents any attempt to execute the unconverted loads/stores to access code or the PPB without going through the atomic instruction sequence. As a result, the atomic verification is achieved and the challenges, **C2**, **C3** and **C4**, are addressed successfully. Unfortunately, this implementation strategy decreases the number of available registers by exclusively allocating one register for the PPB access, which may incur additional register spills and occasionally cause a performance drop in some code with a high register pressure.

Therefore, we employ an alternative strategy that is similar to the basic strategy but differs in that it uses the sp as a base register of every ordinary load/store rather than using the dedicated register. Now, we can achieve the atomic verification if we are able to enforce on the sp the same invariant properties as the dedicated register. Enforcing **Invariant 1** is straightforward, but enforcing **Invariant 2** is challenging because it can cause side effects on the program as the sp is used throughout the program, unlike the dedicated register, which is exclusively used only in the atomic instruction sequence. Fortunately, recall that the sp is a special purpose register that should always point to the stack, so **Invariant 2** can be safely enforced without worrying about side effects.




(a) Before (b) After

Figure 4: An unconverted store before and after applying the atomic verification technique. In the update_register functions `r0` and `r1` are used to pass arguments that will be used as unconverted store's base register and source register, respectively.

**Enforcing Invariant 2 on `sp`.** We achieve this by adopting the idea suggested by the previous work on SFI [7, 33, 39]—we check the value of the `sp` whenever the attackers could have modified it to point to the outside of the valid region (i.e., the stack region). There are three kinds of program points where we need to insert the `sp` check routines: (1) when the `sp` is modified by a non-constant (i.e., register), (2) when the `sp` is increased or decreased by a constant, and (3) at the entry of an exception handler.

We can usually find the first case when the `alloca` function is called, the variable size array is used, or a stack environment stored by the `setjmp` function is restored by the `longjmp` function, which involves an assignment from a general register to the `sp`. As these cases are rare, we insert the `sp` check routines at all the corresponding points. [2]

The second case is very frequently found in the prolog and epilog of a function when the `sp` is adjusted according to the frame size of the function. The attackers could, although not easily, find a suitable gadget consisting of such an instruction and repeatedly execute the gadget until the `sp` is set to a certain value. As pointed out in the previous SFI work [39], if there is a memory instruction based on the `sp` following the `sp` modification, the `sp` can be regulated by placing redzones (i.e., non-accessible memory regions) around the valid stack region. If the redzones are larger than the changes in the value of the `sp`, the following `sp`-based memory instruction ensures that any attempt to use the gadget to jump over the redzones will be detected. Fortunately, the address map illustrated in Figure 3 shows that there already exist large unused regions that can do the role of redzones. This is because in most cases, the stack, code and PPB reside in a separate memory space, such as SRAM, flash memory and system bus, respectively. Therefore, we create redzones only when the stack is created adjacent to the code and PPB without unused regions in between. Note that redzones can detect the corruption of the `sp` only if there is an actual memory access using `sp`. It implies that if, after the `sp` is corrupted, an indirect branch is executed prior to a `sp`-based memory instruction, attackers may be able to evade the execution of the memory instruction by manipulating control flow. Therefore, to ensure the success of this method, we implement an analysis that explores all path from each constant `sp` modification. The analysis checks if there are any `sp`-based memory instructions before a potentially exploitable indirect branch is encountered. According to our experiments, there are some `sp`-based memory instructions preceding indirect branches most of the time. However, we sometimes fail to find any `sp`-based memory instructions or encounter a function call that disables further analysis, and in this case, we insert `sp` check routines because we can no more guarantee the `sp` corruption can be detected by the redzones.

Lastly, the attackers can try to avoid all the checks for `sp` mentioned above by triggering an interrupt right after they corrupt the `sp`. To neutralize this attempt, we have to validate the `sp` by inserting another `sp` check routine at the entry of the exception handlers. Note that as explained in § 2.4, there are two `sp`s in Cortex-M, and different `sp` may be activated before and after the exception, so the `sp` check routine at the entry of the exception handler checks the validity of both `sp`s as shown in Figure 4. The attackers may try to avoid the `sp` check routine by modifying `VTOR` to alter the exception handlers. To avert this attempt, we identify at compile-time the valid values of `VTOR`, and regulate `VTOR` at run-time so that it does not deviate from the identified values, as described in Table 2.

**Fulfillment of the Atomic Verification Technique.** Now, as both **Invariant 1** and **Invariant 2** can be enforced on the `sp`, we can implement the atomic verification technique using the `sp` without allocating a dedicated register. Figure 4 shows an example code on how the atomic verification technique is applied to harden an unconverted store. The original value of the `sp` is backed up while it is used in the unconverted store instruction (Line 3 and 9). The `sp` is assigned a target address (Line 5) and the verification routine verifies the subsequent unconverted store by checking the validity of its target address and target value (Line 6). If the verification is passed, the unconverted store performs memory access (Line 7). Note that

[2] Currently, *u*XOM can handle only C code, so we manually insert the `sp` check routine for the `longjmp` function written in assembly language.

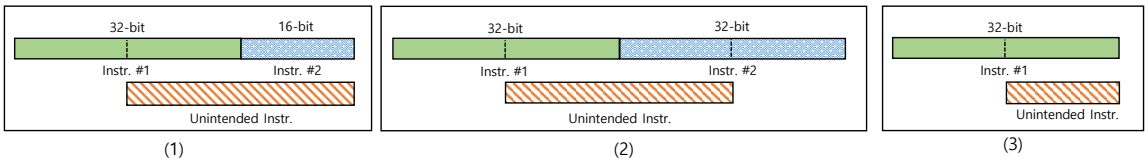


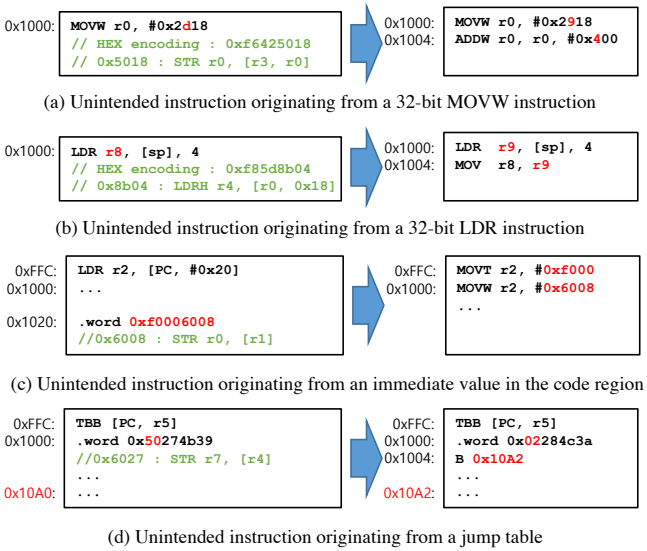

Figure 5: The generation of an unintended instruction by an unaligned execution of a 32-bit instruction.

because **Invariant 2** is enforced by instrumenting `sp`-update instructions and exception handlers (Line 10 and 14), the `sp` always is forced to point to the stack region except when it is used for the unconverted loads/stores. Therefore, to execute the unconverted store for its original purpose (i.e., accessing the PPB), storing the target address (i.e., the address of the PPB) to the `sp` must be preceded (Line 5), which in turn ensures that the verification routine will be performed (Line 6). At the same time, as the `sp` is used for the unconverted loads/stores and may point to out of the stack region, we temporarily disable interrupts (Line 2 and 11), thereby preventing the register from being erroneously checked at the exception handler.

### 5.2.3 Handling Unintended Instructions

As stated in **C5**, our strong attackers capable of manipulating the control flow of the program can execute unintended instructions to bypass the security of *u*XOM. The unintended instructions are mainly caused by the unique property of Thumb-2 instruction set architecture that intermingle 16-bit and 32-bit instructions. Specifically, as shown in Figure 5, when the attackers deliberately jump into the middle of a 32-bit instruction, unintended 16-bit or 32-bit instructions can be decoded and executed. Unintended instructions can also appear in the immediate values in code memory that match the bit patterns of some valid instructions, as illustrated in Figure 6.(b). As such, a number of unintended instructions are lurking in code. Fortunately, however, only a minority of them that can be interpreted as ordinary memory instructions or `sp`-modifying instructions can actually be exploited to compromise *u*XOM.

Against this problem, we have implemented the code instrumentation technique based on the idea in the previous work [4] that replaces each exploitable unintended instruction into safe instruction sequences that serve the same function as the original instruction. There was one complication in solving the problem that not all exploitable unintended instructions can be identified at compile time. Many of the exploitable unintended instructions result from immediate values (i.e., symbol addresses) in instructions which are not resolved until all the object files are linked by the linker. Simply transforming all those instructions that use unresolved symbol addresses will result in unacceptable overhead in both performance and code size. Thus, it is preferable to implement the transformation inside the linker or use the static binary transformation tool. However, adding extra instructions at this stage is almost impossible because it will require us to




(a) Unintended instruction originating from a 32-bit MOVW instruction

(b) Unintended instruction originating from a 32-bit LDR instruction

(c) Unintended instruction originating from an immediate value in the code region

(d) Unintended instruction originating from a jump table

Figure 6: Examples of unintended instructions and code transformations to remove them.

adjust all the `pc`-relative offsets that are used in many ARM instructions. Adding this capability to current ARM GNU linker implementation will require significant engineering effort.[3] As a work around, we implemented a binary verifier that scans the binary executable for exploitable unintended instructions and records the position of each instruction inside the function. With that information, the program is then recompiled and the exploitable unintended instructions are replaced into alternative instruction sequences. Sometimes, new exploitable unintended instructions are revealed after this process, as code and object layouts are changed and offsets and addresses embedded in the code are changed accordingly. Thus, the interaction between the compiler and the verifier is repeated until there are no exploitable unintended instructions in the binary.

Figure 6 demonstrates a few examples showing how the transformation is applied to remove exploitable unintended instructions. Figure 6.(a) shows the case where an exploitable unintended instruction (`STR`) is generated from the immediate value of 32-bit instruction (`MOVW`). To remove the exploitable instruction, we divide the original immediate value into two

[3]This capability is available in the linker for some architectures like RISC-V which implements aggressive linker relaxation. For those architectures, the `pc`-relative offset resolution is deferred until the linking time to enable linker optimizations that reduce instructions and thus may change the `pc`-relative offsets in the code.

numbers A and B. Then we replace the original 32-bit instruction to use A and add an extra instruction (e.g., ADDW) to add B to the register written by the original instruction. Note that for 32-bit instructions whose immediate value is only determined at link time, we only add the extra instruction at compile time and make sure that the linker puts value A and B instead of the original immediate value. Figure 6.(b) shows another example that the destination register of the 32-bit instruction (LDR) generates the exploitable unintended instruction (LDRH). We solve this case by putting the value loaded from memory into the other register and then use an extra MOV instruction to copy the value into the original destination register. We have also implemented an optimization in the register allocation pass to prefer invulnerable registers over the others for the destination of these 32-bit instructions so that exploitable unintended instructions can be avoided as much as possible. This saves the use of extra instructions and reduces the performance and code size overhead. Figure 6.(c) shows an unintended instruction that exists in a constant embedded in a code region to be loaded by a pc-relative load. To sanitize it, we remove the constant value and replace the associated pc-relative load with two move instructions. If the resulting MOVT or MOVW instruction creates new exploitable unintended instructions, it is further transformed similarly to the example in Figure 6.(a). Finally, Figure 6.(d) shows the case where the offsets in a jump table embedded in the code create an exploitable unintended instruction. In the example, the value 0xA0 (0x50 * 2) is added to pc and the control is transferred to 0x10A0. To remove the unintended instruction in this case, we add a trampoline code right after the jump table for the targets with the problematic offsets.

## 5.3 Optimizations

According to our experiments (see § 6.1), unprivileged memory instructions consume the same CPU cycles as ordinary memory instructions. However, unprivileged instructions are 32-bits in size while many ordinary memory instructions have a 16-bit form. Also, extra instructions that are added as described in § 5.1.1 can increase both the code size and the performance overhead. Since code size is another critical factor in an embedded application due to its scarce memory, it can be beneficial to leave the memory instructions in their original form if we can ensure that this does not harm the security guarantees of *u*XOM. In fact, a large number of the instructions do not need to be converted either because they are safe by nature or because they can be made safe through some additional effort. For example, ARM supports pc-relative memory instructions which access a memory location that is a fixed distance away from the current pc—i.e., the address of the current instruction. As these instructions can only access certain data embedded in the code region, attackers cannot exploit them to access other memory locations. Therefore, we do not need to convert these instructions, so we leave them as long as it is not exploitable as unintended instructions (§ 5.2.3). We also do not convert stack-based ordinary memory instructions. Numerous instructions use the sp as the base address. Almost all of them are 16-bits in size since Cortex-M provides special 16-bit encoding for stack-based memory instructions. Converting all of these as the unprivileged will significantly add to the code size of the final binary. Most of the LDM/STM instructions, including all the PUSH/POP instructions, are also based on sp. Converting them would require multiple unprivileged instructions which would further increase the code size and even the performance overhead. Luckily, recall that *u*XOM already enforces the invariant properties noted in § 5.2.2 on the sp. Therefore, attackers cannot exploit the ordinary memory instructions based on sp, and we can safely leave sp based memory instructions in their original forms.

## 5.4 Security Analysis

*u*XOM builds on the premise that there remains no abusable instructions in a firmware binary. *u*XOM satisfies this through its compiler-based static analysis (§ 5.1.1 and § 5.2.3) that (1) identifies all abusable instructions, such as ordinary memory instructions and unintended instructions, and (2) converts them into safe alternative instructions. This conservative analysis does not make false negative conversions, so *u*XOM is fail-safe in terms of security. In the following, we show that attackers we assumed in the threat model (§ 3) will not be able to compromise *u*XOM.

### 5.4.1 At Boot-up

As noted in § 3, we trust the integrity and confidentiality of the firmware image. The firmware image will be distributed and installed with the *u*XOM-related code instrumentation applied. As soon as the system is powered up, the reset exception handler starts to run and the code snippet that *u*XOM inserted at the start of the handler is executed to enforce *u*XOM-specific memory access permissions. Note that the firmware has started its execution from a known good state and the attackers have not yet injected any malicious payloads. Therefore, we can guarantee that *u*XOM will safely enable XOM without being disturbed by the attackers.

### 5.4.2 At Runtime

Once *u*XOM enables XOM, the attackers are completely prevented from accessing the code. They cannot use unprivileged loads/stores to bypass *u*XOM, so they have to resort to the unconverted loads/stores. Through the instruction conversions and optimizations of *u*XOM, only three types of unconverted loads/stores remain in the binary: stack-based loads/stores, exclusive loads/stores and ordinary loads/stores for the PPB access.

**Stack-based loads/stores.** *u*XOM's optimization excludes sp based loads/stores from the conversion candidates. The attackers may be able to execute these loads/stores, but they cannot access the PPB region or code regions. This is because the sp is forced to point to the stack regions due to the invariant property (**Invariant 2** in § 5.2.2) enforced on the

sp.

**Exclusive loads/stores and ordinary loads/stores for the PPB access.** These unconverted loads/stores are protected by the atomic verification technique. Verification routines are inserted just before each unconverted load/store and the atomic execution of the inserted routine and the corresponding unconverted load/store is guaranteed. Of course, the attacker may jump into the middle of the atomic instruction sequence to directly execute the unconverted load/store without a proper verification. However, as the unconverted loads/stores use the sp as their base register, the attackers still cannot access the code and the PPB regions.

## 6 Evaluation

*u*XOM transformations are implemented in LLVM 5.0, and *u*XOM's binary verifier is implemented using the Radare2 binary analysis framework [32]. We used the RIOT-OS [5] version 2018.10 as the embedded operating system. As the whole binary, including the OS, runs in a single physical address space at the same privilege level, *u*XOM compiler transformations are applied to the OS code as well as the application code to enable complete protection. We also applied our transformations to the C library (newlib) included in arm-none-eabi toolchain, which had to be patched in a few places to compile and run correctly with LLVM.

To better show the merits of our approach, we also implemented and evaluated SFI-based XOM to compare against *u*XOM. Originally, SFI is developed to sandbox an untrusted module in the same address space. It restricts the store and indirect branch instructions (i.e., by masking or checking the store/branch address) in the untrusted module so that the untrusted module cannot corrupt or jump into the trusted module. It also bundles the checks with the store/branch instructions and prevents jumps into the bundle so that the restrictions applied to the store or branch address cannot be skipped. Capitalizing on the SFI's access control scheme, some studies [7, 31] have implemented the SFI-based XOM that instruments every load instructions with masking instructions to prevent them from reading the code region. However, as these studies focus on high-end devices like smartphones and desktop PCs, we adapted the SFI-based XOM to work on Cortex-M based devices. As our target device do not use virtual memory, code and data must reside in a specific memory region. This prevents us from using simple masking to restrict load addresses and forces us to use a compare instruction to validate the address. Furthermore, the instruction set of Cortex-M requires us to insert additional IT (If-Then) instruction to make load instruction execute conditionally on the comparison result. Next, we place the compare and load inside a 16-byte aligned bundle and make sure that they do not cross the bundle boundary. We insert NOPs in the resulting gaps. Lower bits of indirect branch targets are masked (cleared) to prevent control flows into the bundle. We also make sure that all possible targets of an indirect branch (i.e., functions and call-sites) are aligned.

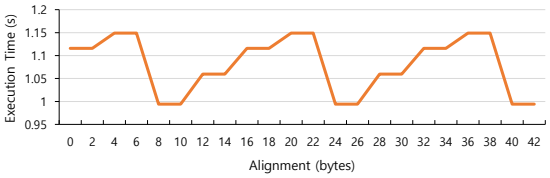



Figure 7: Execution time of bitcount according to the different alignments of the code region.

POP instructions used for function returns are converted to masking and return sequence as described in the previous work on SFI [33]. Following the optimization done in the paper [39], the memory load instructions based on the sp are not checked and the sp is regulated in the same way as in *u*XOM.

To evaluate *u*XOM and the SFI-based XOM, we used the publicly available BEEBs benchmark suite (version 2.1) [29]. We selected 33 benchmarks that are claimed to have relatively long execution time [12][4]. We ran each benchmark on an Arduino Due [1] board which ships with an Atmel SAM3X8E microcontroller based on the Cortex-M3 processor. During the experiment, we found that the program runs give very inconsistent timing results depending on how the code is aligned, even though there are no caches in the processor. After some investigation, we found that the reason is due to the flash memory. The Arduino Due core runs at 84MHz in the default setting, which makes it necessary to wait for 4 cycles (called flash wait state) to get stable results from the flash memory. SAM3X3E chips are equipped with a flash read buffer to accelerate sequential reads [3], which gave us variable results depending on where the branches are located. As a preliminary experiment, we measured the execution time while changing the displacement of the entire code region for bitcount benchmark. As shown in Figure 7, the changes in execution time show a pattern that is repeated every 16-byte, which corresponds to the size of the flash read buffer. Because of this result, to get a consistent result, we decreased the core frequency to 18.5MHz in all our experiments.

### 6.1 Runtime Overhead

Figure 8 shows the runtime overhead of *u*XOM and SFI-based XOM. The geomean overhead of all benchmarks is 7.3% for *u*XOM and 22.7% for SFI-based XOM. The worst case overhead for *u*XOM is 22.3% for huffbench benchmark and that for SFI-based XOM is 75.1% for edn benchmark. Note that the performance overhead of SFI reported in the previous work [33] for a high-end ARM device (Cortex-A9) is 5%. In the paper, they mention that overhead induced by additional instructions for SFI can be hidden by cache misses and out-of-order execution. Based on this, we presume that the large overhead of SFI-based XOM for Cortex-M3 observed in our experiment is due to the low-power and cache-less

[4] Some of the benchmarks have been dropped in the newest version due to the license problem.

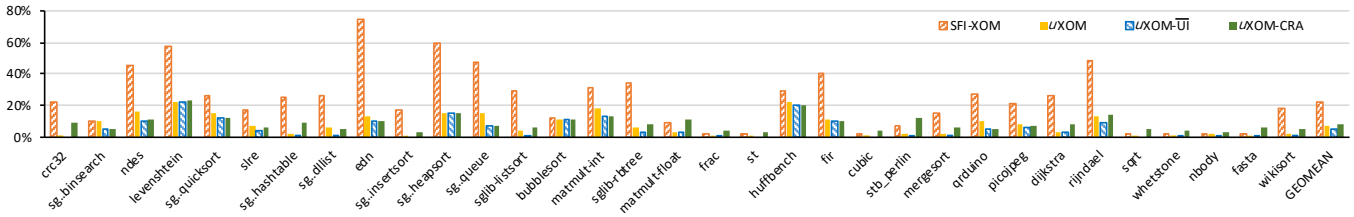



Figure 8: Runtime overhead on BEEBs benchmark suite.

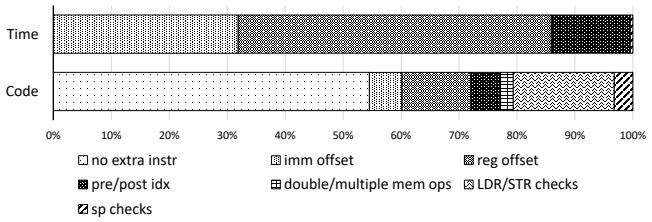



Figure 9: Performance overhead breakdown for the different components of *u*XOM-$\overline{\text{UI}}$ transformation.

| Cases | Count (ratio %) |
|---|---|
| Instruction conversion | |
| no extra instr. | 25932 (77.0) |
| imm. offset | 2547 ( 7.6) |
| pre/post idx. | 1671 ( 4.9) |
| reg. offset | 2891 ( 8.6) |
| double/multiple mem. ops. | 641 ( 1.9) |
| `sp` check | |
| non-const `sp` mod. | 18 ( 0.7) |
| const `sp` mod. (checked) | 769 (28.8) |
| const `sp` mod. (no check) | 1881 (70.5) |

Table 3: Statistics for instruction conversion and `sp` check instrumentation.

processor implementation. This strongly shows the need for an efficient low-end device oriented XOM implementation like *u*XOM.

To inspect the sources of overhead, we built and ran multiple partially instrumented versions of binaries with different kinds of transformations applied. First, to examine the performance impact of removing exploitable unintended instructions, we measured the runtime overhead for *u*XOM-$\overline{\text{UI}}$—a variation of *u*XOM that does not handle unintended instructions. As a result, we measured that the geomean overhead for *u*XOM-$\overline{\text{UI}}$ is 5.2%, which shows that removing unintended instructions incurs 2.1% of overhead in *u*XOM. We then gathered the statistics on the number of conversions and check codes inserted in *u*XOM-$\overline{\text{UI}}$ (Table 3). We also measured the overhead ratio in terms of code size and execution time according to the type of conversions and checks (Figure 9). In Table 3 and Figure 9, `no extra instr.` denotes the case where a memory instruction is converted to an unprivileged one without an additional instruction. `imm. offset` denotes the case where an additional instruction is required because the immediate offset is too large or is negative. `pre/post idx.` represents the pre/post-indexed addressing mode and `reg. offset` represents the register-register addressing mode. `double/multiple mem. ops.` represents LDRD/STRD/LDM/STM instructions. For the `sp` check part, `non-const sp mod.` is the case where the `sp` is modified by the non-constant (and the check is required). `const sp mod. (checked)` is the case where the `sp` is modified by the constant and requires checking since no load/store based on the `sp` is found afterwards. `const sp mod. (no check)` is the case where the `sp` is modified by the constant but does not need to be checked. Finally, `LDR/STR checks` denotes the instructions inserted for the atomic verification technique.

The statistics shown in Table 3 are gathered while compiling the C standard library, RIOT-OS, and each of the benchmarks. Note that although the numbers do not represent those executed at runtime, we can expect some correlation between them. Among the converted memory instructions, the majority of the cases is the one where a memory instruction is directly converted to a single unprivileged memory instruction without any extra instruction (`no extra instr.` accounts for 77% of all conversions). This tells us that most of the load/store instructions are using an immediate-offset addressing mode and the offset is usually small so that it fits in the immediate field of the unprivileged instructions. As we can see, instructions converted in this way do not contribute to the runtime overhead albeit being the majority. Even though the unprivileged instructions are 32-bits long, they do not increase the overhead unless additional instructions are inserted. This is a big advantage for *u*XOM, and it is the main reason why *u*XOM can be much more efficient than SFI-based XOM.

As illustrated in Figure 9, the type of instruction conversions that contributes the most of the overhead is the one for the register-register addressing mode (`reg. offset`). Even though they represent only 8.6% of all conversions, they cause 54% of the total overhead for *u*XOM-$\overline{\text{UI}}$. The reason would be that they are frequently used in time-consuming loops, for example, to index array variables. `imm. offset` and `pre/post idx.` take up the other half of the overhead. Memory instructions that load/store multiple registers (`double/multiple mem. ops.`) cause a negligible runtime overhead; they are rare in number and also, although they are converted into multiple unprivileged instructions, the original instruction also takes up extra cycles to load/store multiple registers. The `sp` checks that are inserted for stack modification have an only negligible impact on performance as our

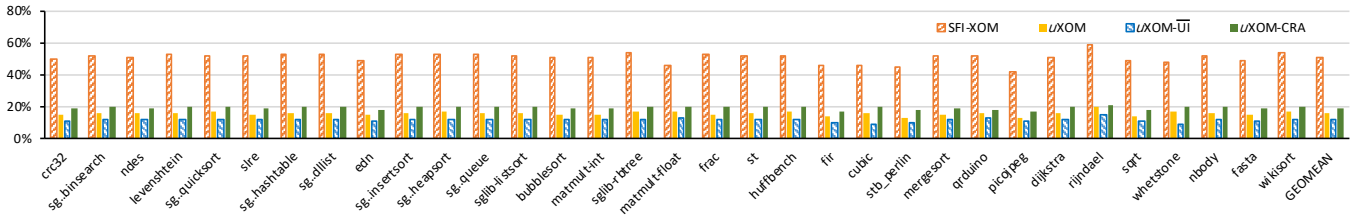



Figure 10: Code size overhead on BEEBs benchmark suite.

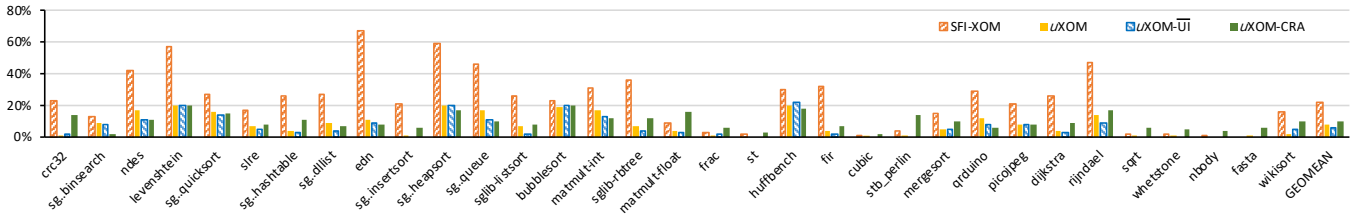



Figure 11: Energy overhead on BEEBs benchmark suite.

analysis finds that the `sp` checks are only needed for less than 30% of `sp`-based memory instructions.

## 6.2 Code Size Overhead

To see the impact of instruction insertion by *u*XOM, we measured the size of the code in the final binary, excluding the data size. Figure 10 shows the result for both *u*XOM and SFI-based XOM. For *u*XOM, code size is increased by 15.7%, and for SFI-based XOM, it is increased by 50.8%. It shows that *u*XOM can implement XOM with much less code size overhead compared to SFI-based XOM. In addition, we measured that the geomean overhead of *u*XOM-$\overline{\text{UI}}$ is 11.6%, which indicates the amount of increased code for removing unintended instructions is 4.1%. Figure 9 shows the source of the overhead that is caused by instruction conversions and checks. First, `no extra inst.` accounts for 54.5% of the code size overhead for *u*XOM-$\overline{\text{UI}}$, differently from the impact that it had on the runtime performance. This is because the original 16-bit load/store instructions are converted to 32-bit unprivileged instruction, and they are large in number, too. Other types of instructions that need additional instructions also increases the code size to some degree. The instructions added for the atomic verification technique (`ldr/str check`) accounts for 17.4% of the code size overhead for *u*XOM-$\overline{\text{UI}}$. Although there are not many instructions accessing the PPB region, around ten instructions are inserted for each of those points, which adds some overhead to the code size especially since the benchmark code size are only around 30KB. We expect the overhead from the atomic verification to be a smaller percentage in the real program with a larger code base.

## 6.3 Energy Overhead

Since many embedded devices running on Cortex-M processors often operate based on constrained battery, energy efficiency is one of the important performance factors for these devices. To measure the impact of *u*XOM on energy consumption, we recorded the power while running the individual benchmarks using the ODROID Smart Power [28]. For the convenience of measurement, the benchmarks were repeatedly executed to run for at least 30 seconds. Figure 11 shows the results. For *u*XOM, the geometric mean of all benchmarks is 7.5%, which is slightly larger than 5.8% of *u*XOM-$\overline{\text{UI}}$ but much lower than 22.3% of SFI-based XOM. The results share a similar trend with the execution time since the energy is also affected by the execution time.

## 6.4 Security and Usability

Other than its excellence for performance, we also need to mention the security and flexibility benefits of *u*XOM over SFI-based XOM. *u*XOM provides a better security guarantee against privileged attackers than SFI-based XOM. SFI-based XOM, including the existing studies, focus only on the code disclosure through memory read instructions, because they assume that W⊕X policy is assured by a Trusted Computing Base (TCB) such as the OS kernel. However, as described in § 3, *u*XOM cannot assume any TCB in the bare-metal environment in which all software components are running with privileges in a single address space. The privileged attacker could neutralize W⊕X by manipulating the MPU configuration register using memory vulnerabilities in the code. To prevent such an attack, SFI-based XOM for Cortex-M would also have to regulate memory write instructions to protect memory-mapped registers for the MPU. However, this would undoubtedly lead to more severe performance overheads, and even worse, SFI-style masking of write instructions would still leave the system vulnerable against attacks through the exception handler (**C3** of § 4). In addition, the current implementation of SFI-based XOM is vulnerable to unintended instructions. To defend this, it should eliminate all exploitable unintended instructions either by using the instruction replacement technique similar to *u*XOM or selectively aligning

32-bit instructions so that jump into the middle of those instructions can be prevented by the masking of indirect jump addresses. Either way, additional performance overhead will be unavoidable.

*u*XOM is also more flexible in placing the code and data. For *u*XOM, the XOM region can be placed anywhere in the address space. For example, *u*XOM can be applied for the code placed in SRAM for performance or firmware updates [20]. Also, *u*XOM can set multiple XOM regions as long as the number of MPU regions supports it. However, SFI-based XOM must place the code at one end and the data on the other to simplify code instrumentation. Moreover, SFI-based XOM needs a guardzone between the code and the data region [39] which further restricts the code and data placement and also causes the memory to be wasted for the guardzone.

## 6.5 Use Cases

*u*XOM can be used to hide sensitive information in the code region, such as secret keys and code layout. We describe two use cases to illustrate how *u*XOM can be applied to a security solution.

**Secret key protection.** In tiny devices, secret keys are frequently used for various purposes, such as device authentication and communication channel protection. *u*XOM can protect these keys against arbitrary memory read vulnerabilities by embedding them inside the code. For example, consider the following code that defines the constant global `key`.

```
const unsigned char key[32] =
    {0xcb, 0x21, 0xad, 0x38, ...};
```

The code that reads the first 4-byte of this value is compiled to the assembly code composed of `MOVW` and `MOVT` as follows:

```
MOVW r0, #0x21cb
MOVT r0, #0x38ad
```

Now, if we use *u*XOM to apply the XO permission to this code, attackers cannot access the key value by arbitrary memory reads. As an example, we applied *u*XOM to `rijndael` benchmark, which uses a symmetric key for encryption. By declaring the key as a global constant, we could confirm that the key is embedded in the code protected by *u*XOM. Such a protection offered by *u*XOM can further be combined with in-register computation techniques [26] for a secure computation robust against memory vulnerabilities.

**CRA defense.** To date, many researchers have proposed code diversification-based CRA defense techniques [7, 12, 13, 30]. They randomize code layout to prevent attackers from using the existing gadgets for CRA. As the code disclosure attack emerged as a serious threat to randomization-based defenses, XOM has been proposed as an effective solution to fortify these defenses.

As another use case of *u*XOM, we implemented a CRA defense solution based on Readactor [13], which is a representative code diversification based CRA defense with resistance to code disclosure attacks. Readactor aims to defend against two classes of code disclosure attacks: direct disclosure where the attackers disclose code layout by directly reading the code and indirect disclosure where attackers indirectly infer the code layout through the value of the code pointers. Readactor first places all code in XOM to prevent the direct disclosure attacks. It then replaces all code pointers with pointers to *trampolines* so that all indirect control transfers must go through the trampoline. In this way, code pointers containing the original code location are never stored in a register or memory, thereby preventing the indirect disclosure attacks. To demonstrate this use case, we implemented function reordering and the trampoline mechanism. Every function call is replaced with a direct branch to the trampoline followed by the call to the original function. When the original function returns, another direct branch takes the control flow back to the original callsite. Also, every function pointer is replaced with a pointer to the corresponding function trampoline. We implemented this use case on top of *u*XOM-$\overline{\text{UI}}$ because the code diversification based CRA defense mitigates control flow hijacking, and consequently hinders an attacker from exploiting unintended instructions. The experimental results of our CRA defense are presented together with the results for *u*XOM, *u*XOM-$\overline{\text{UI}}$ and SFI-based XOM. It imposes average runtime overhead of 8.6%, the code size overhead of 19.3%, and the energy overhead of 9.7%. The runtime overhead is only slightly larger than that for original Readactor implementation (6.4%) which shows the applicability of *u*XOM technique in low-end embedded devices.

# 7 Discussion

**Cortex-M Processors based on ARMv8-M Architecture.** ARMv8-M [19] is a recently introduced instruction set architecture for the microcontroller profile. Basically, ARMv8-M provides backward compatibility with ARMv7-M, so *u*XOM is also applicable to ARMv8-M based Cortex-M(23/33/35) processors. Here, we list several possible changes in *u*XOM implementation due to the newly added hardware feature in ARMv8-M. First of all, ARMv8-M includes the stack pointer limit register (`SPLR`) that defines a lower limit for the stack pointer and prevents the stack pointer from pointing below the limit. When enabling `SPLR`, therefore, *u*XOM only needs to ensure that the stack pointer does not point to the PPB region. Secondly, load-acquire and store-release memory instructions are newly added in ARMv8-M. Since these instructions do not have unprivileged counterparts, they should be protected by the atomic verification technique.

**False Positive Conversion.** When it comes to the instruction conversion of *u*XOM, false positive cases could happen where unconvertible instructions are converted to unprivileged ones. The false positive conversion does not harm the security

aspect of *u*XOM but may cause an unexpected system fault. For instance, if PPB-accessing memory instructions are converted to unprivileged ones, it would not expose the PPB to attackers but raise a memory access fault when executed. To avoid an unexpected system halt due to the fault, uXOM can install a custom fault handler, which in turn may invoke the fail-safe handler already implemented in the existing system (e.g., emergency landing in drones).

**Dynamic Data Protection.** Although the current *u*XOM implementation aims to defeat the code disclosure attacks, it may be extended to provide protection for the dynamic data as well. To be concrete, *u*XOM can be expanded to implement a data isolation scheme [23, 35, 36] that minimizes the possibility of exposures of critical data by only allowing access through authorized instructions. More specifically, we may allow only authorized instructions (i.e., ordinary loads/stores that are not converted into unprivileged types) to access critical data (e.g., return addresses/session keys) by placing the data on a certain memory region marked as "privileged". To implement such an extension, some modifications to *u*XOM are required. First of all, authorized instructions should be predetermined through the help of programmers or compilers and prevented from being converted to unprivileged ones. Since attackers can exploit these data-accessing instructions to compromise *u*XOM, usage of these instructions should be regulated in a way similar to PPB-accessing instructions through the atomic verification technique with a new verification routine that confines memory access target to the memory region of the critical data.

## 8 Related Work

**Hardware-assisted Execute Only Memory.** Due to the compelling security guarantee provided by XOM, today's high-end processor architectures (e.g., x64 and AArch64) provide the XO permission setting in the MMU [8, 10]. Apart from that, various works have attempted to implement XOM in the system with the help of the hardware. David et al. [24] implemented XOM by encrypting the code in memory and decrypting it only when it is executed. However, since it requires significant processor redesign, it is not suitable for wide adoption. In subsequent works, XOM has been implemented by capitalizing on the built-in hardware features. Shadow Walker [37] and HideM [17] presented an implementation of XOM using the split translation lookaside buffer (TLB) architecture, which separates the TLB for instruction fetches and data accesses. They configure the two TLBs so that the same virtual address is translated into different physical addresses for data access and instruction fetch, preventing the data accesses to the code region. XOM-switch [25] implemented XOM using Intel Memory Protection Keys (MPK), which can be used to set memory pages execute-only. Shadow Walker, HideM and XOM-switch are not applicable to Cortex-M based devices because they rely on specific hardware features (i.e., split-TLB or Intel MPK) that do not exist in the Cortex-M processor.

**Software-based Execute Only Memory.** On the other hand, there have been attempts to emulate XOM in software for processors that do not have the above hardware supports. XnR [6] sets all code pages as non-accessible except for the currently executed code pages called *sliding window* and detects illegal memory reads and writes for non-accessible pages by augmenting the MMU page fault handler. For Cortex-M/R processors, since MPU also provides non-accessible permission setting for memory regions, XOM can be implemented in a similar way. However, this approach cannot detect memory reads for code pages in the sliding window, and also, the performance overhead becomes larger as the sliding window size is reduced. $LR^2$ [7] and kR^X [31] realize XOM by SFI-inspired techniques [39, 40]. They prevent code reads by masking load instructions, instead of stores as done in the SFI technique. As shown in our evaluation, however, such SFI-based XOM implementation can be bypassed and is inefficient in low-end devices.

**Security Solutions using XOM.** Many researchers have proposed various security solutions based on XOM. Early works [27] proposed XOM for the purpose of protecting intellectual properties and preventing tampering or leakage of sensitive information stored in the code. Since the advent of code disclosure attacks (i.e., JIT-ROP), a number of works [7, 13, 16, 31] have utilized XOM to prevent the attackers from reading code to learn code layout and launch CRAs. In § 6.5, we have shown that these solutions can be implemented with *u*XOM.

**Security for Tiny Embedded Devices.** Recently, much research has been done on enhancing the security of tiny embedded devices. Mbed uvisor [2], MINION [21], uSFI [4] and ACES [11] proposed memory isolation techniques for software modules based on MPU. At compile time, they define memory views (stack, heap, and peripherals) for each of the software modules, and at runtime, MPU enforces one of the memory views according to the active software module. Epoxy [12] and AVRAND [30] developed diversification based security solutions for tiny embedded devices. As with these solutions, *u*XOM also seeks to enhance the security of tiny embedded devices. *u*XOM is the first to implement efficient execute-only memory in Cortex-M processors.

## 9 Conclusion

XOM is a prominent protection mechanism that can be used in various security purposes such as intellectual property protection and CRA defense. However, for a low-end embedded processor such as Cortex-M, there has been no efficient way to implement XOM. In this paper, we present *u*XOM, a novel technique to realize XOM in a way that is secure and highly optimized to work on Cortex-M processors. *u*XOM achieves this by leveraging hardware features (i.e., unpriv-

ileged load/store instructions and MPU) in Cortex-M processors. Our evaluation shows that not only *u*XOM is more efficient than SFI-based XOM in terms of execution time, code size and energy consumption, and that *u*XOM is compatible with existing XOM-based security solutions.

## Acknowledgments

We thank the anonymous reviewers and our shepherd, Vasileios P. Kemerlis, for their valuable comments that helped to improve our paper. This work was partly supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (NRF-2017R1A2A1A17069478, NRF-2018R1D1A1B07049870, No. 2019R1C1C1006095), Institute of Information Communications Technology Planning Evaluation (IITP) grant funded by Korea government (Ministry of Science and ICT) (No. 2016-0-00078, No.2018-0-00230, No. 2017-0-00168), and the Brain Korea 21 Plus Project in 2019. The ICT at Seoul National University provides research facilities for this study.

## References

[1] Arduino. arduino-due. https://store.arduino.cc/usa/arduino-due.

[2] ARM. The mebed os uvisor. https://www.mbed.com/en/technologies/security/uvisor/.

[3] Atmel. Atmel-11057c-atarm-sam3x-sam3a-datasheet, 2015.

[4] Zelalem Birhanu Aweke and Todd Austin. usfi: Ultra-lightweight software fault isolation for iot-class devices. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1015–1020. IEEE, 2018.

[5] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas C Schmidt. Riot os: Towards an os for the internet of things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 79–80. IEEE, 2013.

[6] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1342–1353. ACM, 2014.

[7] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-resilient layout randomization for mobile devices. In *NDSS*, 2016.

[8] Scott Brookes, Robert Denz, Martin Osterloh, and Stephen Taylor. Exoshim: Preventing memory disclosure using execute-only kernel code. In *Proceedings of the 11th International Conference on Cyber Warfare and Security*, pages 56–66, 2016.

[9] Xi Chen, Robert P Dick, and Alok Choudhary. Operating system controlled processor-memory bus encryption. In *Design, Automation and Test in Europe, 2008. DATE'08*, pages 1154–1159. IEEE, 2008.

[10] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. Norax: Enabling execute-only memory for cots binaries on aarch64. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 304–319. IEEE, 2017.

[11] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. Aces: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 65–82, 2018.

[12] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 289–303. IEEE, 2017.

[13] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 763–780. IEEE, 2015.

[14] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*, pages 463–478, 2013.

[15] Úlfar Erlingsson. The inlined reference monitor approach to security policy enforcement. Technical report, Cornell University, 2003.

[16] Jason Gionta, William Enck, and Per Larsen. Preventing kernel code-reuse attacks through disclosure resistant code diversification. In *Communications and Network Security (CNS), 2016 IEEE Conference on*, pages 189–197. IEEE, 2016.

[17] Jason Gionta, William Enck, and Peng Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 325–336. ACM, 2015.

[18] ARM Holdings. Armv7-m architecture reference manual, 2010.

[19] ARM Holdings. Armv8-m architecture reference manual, 2017.

[20] IAR. Execute in ram after copying from flash or rom. https://www.iar.com/support/tech-notes/general/execute-in-ram-after-copying-from-flashrom-v5.20-and-later/.

[21] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *Network and Distributed Systems Security Symp.(NDSS)*, 2018.

[22] Oliver Kömmerling and Markus G Kuhn. Design principles for tamper-resistant smartcard processors. *Smartcard*, 99:9–20, 1999.

[23] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, 2014.

[24] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[25] Ravi Sahita Mingwei Zhang and Daiping Liu. executable-only-memory-switch (xom-switch). *Black Hat Asia*, 2018.

[26] Tilo Müller, Felix C Freiling, and Andreas Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, volume 17, 2011.

[27] Gleb Naumovich and Nasir Memon. Preventing piracy, reverse engineering, and tampering. *Computer*, 36(7):64–71, 2003.

[28] ODROID. smart-power. https://wiki.odroid.com/old_product/accessory/odroidsmartpower.

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

[30] Sergio Pastrana, Juan Tapiador, Guillermo Suarez-Tangil, and Pedro Peris-López. Avrand: a software-based defense against code reuse attacks for avr embedded devices. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 58–77. Springer, 2016.

[31] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. krˆx: Comprehensive kernel protection against just-in-time code reuse. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 420–436. ACM, 2017.

[32] Radare2. unix-like reverse engineering framework and commandline tools. https://www.radare.org/r/.

[33] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *USENIX Security Symposium*, pages 1–12, 2010.

[34] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[35] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[36] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[37] Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, 11(63):504–533, 2005.

[38] Menasveta Tim, Soubra Diya, and Yiu Joseph. Introducing arm cortex-m23 and cortex-m33 processors with trustzone for armv8-m, 2016.

[39] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, 1994.

[40] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.