*u*BOX: A Lightweight and Hardware-Assisted Sandbox for Multicore Embedded Systems

Xia Zhou, Yujie Bu, Meng Xu, *Member, IEEE*, Yajin Zhou, and Lei Wu

***Abstract*—Multicore embedded systems employ a big.LITTLE architecture to combine different cores into a single microcontroller (MCU). However, resources sharing among cores raises security challenges. Once LITTLE cores (which often receive external inputs) are compromised, the whole system will be affected. Existing hardware-assisted isolation approaches use privilege separation and code instrumentation to enforce memory isolation, which suffer from inefficiencies. This paper presents *u*BOX, a lightweight sandbox for multicore embedded systems. The goal of *u*BOX is to enforce memory isolation over untrusted software (on LITTLE cores) at the same privileged level. Specifically, it uses the Memory Protection Unit (MPU) to restrict memory access by untrusted software. To protect sandbox policies, *u*BOX deprives the write capability of untrusted software towards MPU configurations by replacing its regular store instructions with unprivileged counterparts. Additionally, to protect *u*BOX's necessary regular store instructions from being abused, *u*BOX's memory is set to read-only and non-executable when running untrusted software. For the normal operation of *u*BOX, we use an overlooked feature of the MPU and develop secure gates that quickly disable and re-enable the MPU, allowing *u*BOX to execute at a permissive memory view. Our evaluation demonstrates that *u*BOX effectively enforces isolation with average 1.27% runtime overhead, 0.83X Flash overhead, and 36.50X SRAM overhead.**

***Index Terms*—Embedded systems, memory protection unit, security isolation.**

I. Introduction

In RECENT years, the big.LITTLE architecture has become widely adopted [1], [2], [3], [4], [5]. This heterogeneous design couples slow but power-saving cores (LITTLE) with fast but power-hungry ones (big), allowing the cores to adjust to dynamic computing needs with less power consumption. Many multicore embedded systems have incorporated this design [6], [7], [8] due to the prolonged battery life.

However, this heterogeneous architecture also raises new security challenges due to the shared resources between big and LITTLE cores. Specifically, memories and peripherals are shared among the cores. If a vulnerability exists in the software stack on one side of the cores, (e.g., the LITTLE cores, which usually interact with external inputs), attackers can first exploit one side and then use it as an intermediate to further compromise the other side (e.g., the big cores). Moreover, embedded systems often disregard privilege separation and run the entire firmware at the privileged level for performance reasons, which exaggerates this issue. As shown in previous attacks, attackers can first exploit the vulnerabilities of a Wi-Fi SoC and then compromise the application processor [9], [10]. Therefore, it is essential to enforce isolation between big and LITTLE cores.

Software Fault Isolation (SFI) is a mechanism to enforce the establishment of logical protection domains through software instrumentation and hardware-assisted methods [11], [12], [13], [14], [15], [16]. Because pure software-based SFI systems usually rely on heavyweight code instrumentation or safe programming languages, they have high performance overhead and compatibility issues [17], [18]. Hardware-assisted schemes, on the other hand, leverage hardware primitives to achieve protection. They are thus more efficient and are getting more attention.

A classic theme in hardware-assisted SFI is to utilize the MPU to confine the memory access of untrusted software [19], [20], [21], [22], [23]. Memory isolation is achieved via privilege separation and demotion of most untrusted code to the unprivileged level. Any modification to system configurations from unprivileged code will be trapped and checked by a trusted and privileged reference monitor. However, some security-sensitive instructions that modify system status must run at the privileged level. To protect them from being abused by attackers to further break the isolation, additional measures like CFI [24], [25] and shadow stack [26], [27] need to be deployed, which incurs overhead [28], [29], [30], [31]. So the research question is: can we design an SFI mechanism that efficiently prevents itself from being circumvented?

This paper presents *u*BOX, a lightweight sandbox isolating the firmware without relying on full privilege separation, with an overview illustrated in Fig. 1. Note that in this paper, we hypothetically consider LITTLE cores as untrustworthy for illustration purposes, but *u*BOX can be applied to big cores if they are deemed untrusted. *u*BOX adopts the MPU to confine memory access of untrusted software. More importantly, to comply with the convention of embedded systems development, *u*BOX runs all untrusted code *at the privileged level*. However, this compatibility accommodation brings two challenges to be addressed:

Received 24 February 2024; revised 12 August 2024; accepted 2 September 2024. Date of publication 11 September 2024; date of current version 14 March 2025. This work was supported in part by the National Key R&D Program of China under Grant 2022YFE0113200 and in part by the National Natural Science Foundation of China (NSFC) under Grant 62172360 and Grant U21A20464. *(Corresponding author: Yajin Zhou.)*

Xia Zhou, Yujie Bu, Yajin Zhou, and Lei Wu are with the School of Computer Science and Technology,Zhejiang University, Hangzhou, CA 310027, China (e-mail: zhouxia_icsr@zju.edu.cn; insomnia6974@gmail.com; yajin_zhou@zju.edu.cn; lei_wu@zju.edu.cn).

Meng Xu is with the Cheriton School of Computer Science,University of Waterloo, Waterloo N2L 3G1, Canada (e-mail: meng.xu.cs@uwaterloo.ca).

This article has supplementary downloadable material available at https://doi.org/10.1109/TDSC.2024.3454421, provided by the authors.

Digital Object Identifier 10.1109/TDSC.2024.3454421

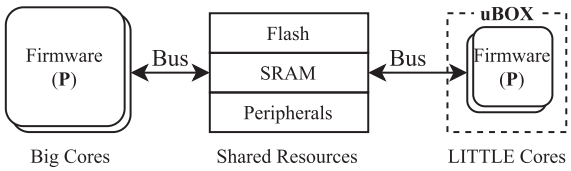

Fig. 1. *u*BOX confines memory access of LITTLE cores. **P**: **P**rivileged.

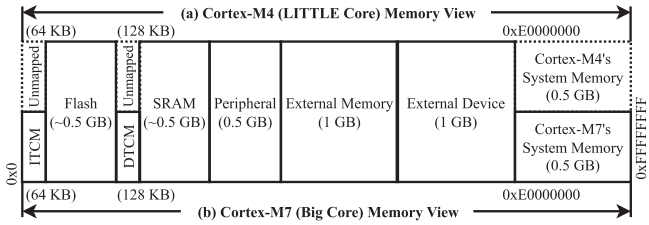

Fig. 2. Divergent memory views of the Cortex-M7 and Cortex-M4 cores of the STM32H745 MCU. Although the two cores share most of the address space and resources, each core maintains a distinct system memory starting at `0xE0000000`.

Challenge I: The MPU is not sufficient to protect memory isolation policies as any privileged memory access to system configurations is always permitted [32].

To resolve this challenge, we develop *fault-based configuration protection* to deprive untrusted code of the capability that writes to privileged-only memory ranges. To construct this protection, we employ unprivileged store instructions of the ARMv7-M architecture [33].[1] Specifically, regular store instructions from untrusted code are replaced with unprivileged counterparts. Therefore, untrusted modifications to system configurations will be trapped to a trusted reference monitor. This challenge is faced similarly in prior works as well and is also resolved with unprivileged store instructions [35], [36]. The difference between *u*BOX and prior works, however, lies in the handling of store instructions that cannot be "unprivileged".

Challenge II: Even most regular store instructions have been replaced with unprivileged ones, a few of them must be reserved in our reference monitor for the normal functionality of *u*BOX. Therefore, privileged attackers may abuse these instructions and leverage them to breach the sandbox. Moreover, the ARMv7-M architecture lacks higher privilege levels, such as hypervisor or monitor levels to host our reference monitor. Although the TrustZone-M extension has been introduced for Cortex-M MCUs, it is available only from the ARMv8-M architecture [37], which is not widely used in mainstream multicore MCUs.[2]

To resolve this challenge, we develop *state-based execution protection* that quickly changes the executable state of our reference monitor. Initially, the memory of our reference monitor is configured as read-only and non-executable when running the untrusted code. To properly run the reference monitor, we establish a secure memory domain for it and design *secure gates* that quickly change domains [38], [39], [40], [41], [42], [43]. In particular, we adopt an overlooked feature of the MPU, which can be used to quickly disable and re-enable the MPU itself. The MPU can be disabled automatically once the current execution priority value is equal to or less than -1 (Section II-C). Consequently, our reference monitor can operate normally at a permissive memory view, i.e., the secure domain. In such a view, our reference monitor has unrestricted access to read and write across the entire address space, with code execution similarly unrestricted.

We have implemented a prototype of *u*BOX. It consists of two components: 1) `uBOX-Compiler` is an LLVM-based compiler that compiles the source code and produces an executable program image finally; 2) `uBOX-Monitor` acts as the reference monitor for the sandbox. It mediates all write access to sensitive memory regions and blocks any malicious operations. We have evaluated *u*BOX on the STM32H745I-DISCO board [44]. It features a dual-core STM32H745 MCU, consisting of a Cortex-M7 core (referred to as "CM7 core" or "big core") and a Cortex-M4 core (referred to as "CM4 core" or "LITTLE core") [45]. We conduct experiments on 6 representative applications and CoreMark [46] benchmark to evaluate the overall performance overhead of *u*BOX. The experimental results show that *u*BOX is lightweight, incurring average 1.27% of runtime overhead. It incurs 0.83X of Flash overhead and 36.50X of SRAM overhead. Although the SRAM overhead is relatively high, *u*BOX does not cause too much memory overhead as the SRAM consumption is constant, which is acceptable. Furthermore, we evaluated *u*BOX with 6 microbenchmarks to understand the performance impact of its each functionality.

Summary: The contributions of this paper are threefold:

- We propose *u*BOX, a new lightweight sandbox for multicore embedded systems with intra-address space memory isolation.
- We design two key techniques to protect our reference monitor, `uBOX-Monitor`. We employ two essential hardware primitives to construct these techniques.
- We implement a prototype of *u*BOX and evaluate it with 6 representative applications, CoreMark benchmark, and 6 microbenchmarks on the STM32H745I-DISCO board. Our evaluation demonstrates that *u*BOX incurs a negligible performance overhead and does not cause excessive memory pressure.

II. Background

A. The Multicore Architecture

The STM32H745 MCU comprises two cores, namely a CM7 core and a CM4 core [45], and both cores are based on the ARMv7-M architecture [32]. Each core is assigned to different tasks and operates independently. In general, the CM7 core runs computation-intensive tasks such as AI inference and human-machine interface. The CM4 core runs lightweight tasks such as sensing, and communication. Furthermore, each core has a distinct memory view. As illustrated in Fig. 2, most of the address space and resources are shared between the two cores, including

[1] In our survey of top 5 MCU suppliers based on the Arm architecture [34], we discovered that 98.04% of them feature at least one core with the ARMv7-M architecture.

[2] In our survey, 5.22% of the multicore MCUs use the ARMv8-M architecture.

Flash, SRAM, and peripherals. However, each core has its distinct address space for system configurations, which starts at `0xE0000000`. This special address space includes the Private Peripheral Bus (PPB), where core peripherals, such as the MPU and the Data Watchpoint and Trace (DWT) unit, are located. As a result, one core is incapable of confining the accessible memory of the other core by configuring its corresponding MPU. Moreover, the CM7 core has dedicated Instruction Memory (ITCM) and Data Tightly Coupled Memory (DTCM) for fast instruction fetch and data access (Fig. 2(b)). The CM4 core can access DTCM and ITCM only through the Master DMA (MDMA) controller indirectly. The MDMA controller is located at the address space of peripherals.

B. Memory Protection Unit

The Memory Protection Unit (MPU) enforces memory permissions on the physical address space. Depending on the specific MCU customization, the MPU can have either 8 or 16 memory regions. Each region is capable of independently setting the memory permissions of a memory range and is assigned with a region id starting from 0. When multiple memory regions overlap with each other, the region with the highest id among them determines the memory permissions of the overlapped address range. A `MemManage` (memory management) exception will be raised if any memory access violates the memory permissions. The size of a region must be a power of 2, and the minimum region size is 32 bytes. The starting address of a region must align with its size, otherwise the configuration will be invalid. Additionally, one region can be divided into eight subregions of equal size. Each subregion can be disabled or enabled independently. If a subregion within a region is disabled, the memory permissions defined by that region will not be enforced at the memory covered by the corresponding subregion.

Moreover, the ARMv7-M architecture also features a default memory view in which privileged software can execute while unprivileged one cannot. This default view is permissive and is used if the MPU is disabled. In this view, privileged software has unrestricted access to the entire address space for both reading and writing, and code execution is equally unrestricted.

C. Execution Priority & MPU Bypass

In the ARMv7-M architecture, every exception has execution priority determined by a priority value. A lower priority value indicates a higher execution priority. High-priority tasks or exception handlers can preempt the execution of low-priority ones. The execution priority is configurable at runtime only by privileged code and the maximum configurable execution priority is 0. Moreover, three exceptions have fixed execution priority: the `Reset` exception (-3), the Non-Maskable Interrupt (`NMI`) exception (-2), and the `HardFault` exception (-1).

The MPU has an *MPU bypass* feature associated with certain execution priorities, which is often overlooked. Specifically, if the current priority value is equal to or less than -1 (i.e., the execution priority of the `HardFault` exception), and the `HFNMIENA` bit of the `MPU_CTRL` register is set to 0, the MPU will be disabled automatically and the privileged software will run in the permissive memory view. To raise the execution priority to -1, software can set the 1-bit `FAULTMASK` register to 1 by executing `CPS` or `MSR` instructions. Conversely, the process of recovering from an execution priority of -1 is to set `FAULTMASK` to 0. This feature can be used to quickly disable and enable the MPU.

D. Unprivileged Store Instructions

The ARMv7-M architecture features two privilege levels, privileged and unprivileged [32]. Notably, access to system memory is only confined by the default memory view rather than the MPU. In other words, privileged software can always access the system memory without the constraints of the MPU. In contrast, direct access to the system memory region by unprivileged software will always trigger a `BusFault` exception. Unprivileged software can use an `SVC` instruction to trigger a Supervisor call (`SVCall`) to elevate itself to the privileged level. Moreover, the ARMv7-M architecture features unprivileged store instructions (`STRT`) [33]. At the privileged level, unprivileged store instructions are confined in the same way as regular store instructions at the unprivileged level. In particular, writing to the system memory via unprivileged store instructions will always trigger a `BusFault` exception.

III. Assumptions & Threat Model

Our system assumes a strong threat model. As LITTLE cores usually process external inputs, we consider the firmware running on the CM4 core as untrusted. Moreover, we assume that the CM4 core runs its firmware at the privileged level, which conforms to the convention of embedded systems. Moreover, we assume that both cores run firmware at the privileged level, which conforms to the convention of embedded systems. Attackers can exploit vulnerabilities of the firmware running on the CM4, thereby gaining primitives to arbitrary memory reads and writes to further compromise the whole system. We also assume that the source code of the firmware operating on the CM4 core is available. Additionally, we assume that each of the trusted/untrusted code and data section is continuous. This assumption can be achieved by arranging sections layout through `uBOX-Compiler` (Section IV-A). Both firmware can be either bare-metal or compiled with a RTOS (e.g., FreeRTOS). The goal of attackers is to corrupt the memory of other cores and compromise the whole system.

We assume that the compilation toolchain for code compilation and instrumentation is trustworthy. Moreover, we assume that the code of `uBOX-Monitor` is trusted and free of memory safety issues. Our reference monitor includes exception handlers for `HardFault`, `BusFault`, `MemManage`, and `SVCall` exceptions, and wrapper functions used for updating the MPU. In addition, the `Reset` and `NMI` exception handlers are trusted. The boot processes of both cores are trusted as well. Furthermore, we assume that the CM4 core features a hardware MPU and unprivileged store instructions (specifically, `STRT` instructions), both of which are essential hardware primitives for our *u*BOX design. The primary goal of *u*BOX is to isolate attackers

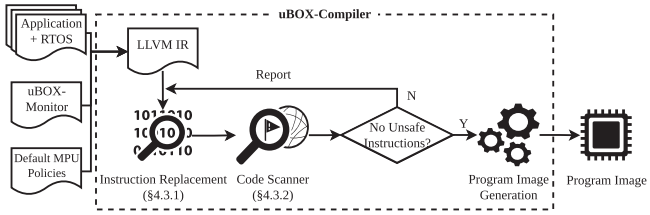

Fig. 3. The workflow of *u*BOX.

within the boundaries of the CM4 core without affecting the execution of the other core.

Denial of Service (DoS) attacks that disrupt the `uBOX-Monitor`'s operation by raising the top-priority `Reset` exception, leading to system reboot, are beyond the scope of this work. Side-channel attacks and physical attacks are considered out-of-scope in this work. Since Cortex-M MCUs lack the Input-Output Memory Management Unit (IOMMU), we also disregard DMA attacks performed by malicious peripherals through DMA controllers.

IV. Design

*u*BOX is a lightweight sandbox designed for multicore embedded systems. It addresses the two challenges discussed in Section I while achieving the following goals:

- G1 *Complete mediation:* Our reference monitor must comprehensively restrict memory access from untrusted software to prevent potential breaches of the sandbox.
- G2 *Tamperproofness:* Privileged attackers should not affect the normal operation of our reference monitor. Therefore, the sandbox policies and the memory of our reference monitor should be well protected to guarantee integrity.
- G3 *Scalability:* The hardware primitives utilized by our sandbox should be widely available in MCUs, ensuring scalability.
- G4 *Lightweight:* Our sandbox should introduce minimal performance overhead.
- G5 *Compatibility:* Our sandbox should be compatible with mainstream RTOSes such as FreeRTOS.

A. Workflow

Our system comprises two distinct components: `uBOX-Compiler` and `uBOX-Monitor`. `uBOX-Compiler` is an LLVM-based compiler that performs the compilation and instrumentation. `uBOX-Monitor` is our reference monitor that enforces the isolation at the same privileged level as adversaries. Fig. 3 depicts the workflow of our system.

The input of our compiler can be categorized into three parts. The first part includes the source code of the application and the RTOS kernel. The second part is the source code of `uBOX-Monitor`. The third part is the default MPU policies used for enforcing memory isolation at runtime. First, `uBOX-Compiler` compiles all the source code to produce the LLVM IR bytecode. Subsequently, it runs several LLVM passes to apply code instrumentation (Section IV-C1) at the link-time optimization stage to produce an intermediate program image. After that, our compiler employs a code scanner (Section IV-C2) to analyze the generated image and verifies that there are no unsafe instructions left. Otherwise, it will produce a report to help developers eliminate unsafe instructions. Eventually, a program image compiled and linked with `uBOX-Monitor` is generated as the output.

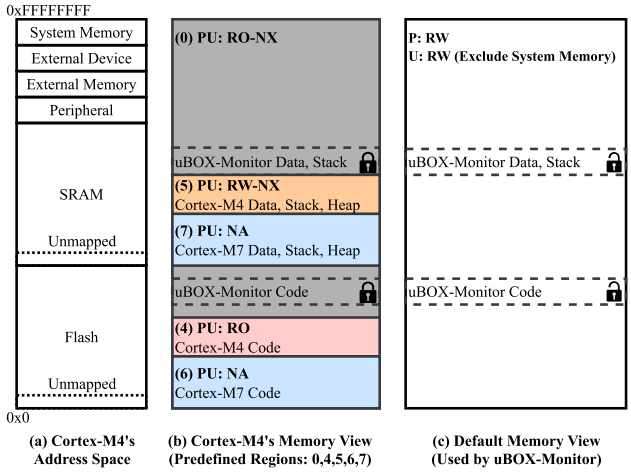

Fig. 4. *u*BOX's default MPU policies for the Cortex-M4 core. **P**: **P**rivileged. **U**: **U**nprivileged. **RO**: **R**ead-**O**nly. **RW**: **R**ead & **W**rite. **NX**: **N**on-e**X**ecutable. **NA**: **N**o **A**ccess.

B. MPU Region Arrangement

In the default memory configuration, the CM7 core and the CM4 each utilize a separate address range for Flash and SRAM to store their respective private code and data, yet they share the peripheral address space (Section II-A). Furthermore, a portion of SRAM is designed as shared memory between the two cores to facilitate inter-core communication. Consequently, our system must establish a security boundary to segregate the private address spaces of each core and regulate access to the shared memory.

`uBOX-Monitor` utilizes the MPU to regulate the memory access of the untrusted software at runtime. In our workflow, `uBOX-Monitor` uses the default MPU policies (Fig. 3) to arrange MPU regions. The default MPU policies adhere to the default memory configuration of the two cores. Note that we assume that the untrusted software runs at the privileged level. The policies employ eight MPU regions and an extra default memory view to protect sensitive memories efficiently and securely. As depicted in Fig. 4(b), the eight MPU regions of the default MPU policies are arranged as follows:

- Region 0 is designated to set the entire memory view of the CM4 core as read-only and non-executable, which meets **G1**.
- Region 1-3 are reserved for enabling dynamic write access to peripherals. If the application is compiled with a FreeRTOS-MPU, region 3 is reserved for memory write to extra memory ranges of an untrusted task and regions 1-2 are still reserved for peripherals, which meets **G5**.

- Region 4 is designated to set the code memory of the CM4 core as executable, which enables the normal execution of untrusted software.
- Region 5 is designated to allow the untrusted software to write its global data, stack, and heap.
- Region 6 and region 7 are designed to protect sensitive memories. Specifically, they prohibit memory read and write to the CM7 core's memory.

To meet **G2**, the memory of uBOX-Monitor should be non-writable for untrusted code. The protection to this dedicated range of memory varies according to the size of the CM4 core's writable memory. If the size is a power of 2, which can be covered by region 5, the final subregion of region 5 will be disabled and the memory covered by the final subregion is used to place uBOX-Monitor's writable memory. Otherwise, the CM4 core's writable memory will be shrunk to a smaller power of 2 to fit region 5. Then the CM4 core's writable memory uncovered by region 5 will be reserved for uBOX-Monitor. Moreover, uBOX-Monitor's code is well placed to ensure that it is uncovered by region 4. Eventually, uBOX-Monitor's memory is regulated by region 0 and configured as read-only and non-executable, as illustrated in Fig. 4(b). Therefore, if attackers try to modify the data of uBOX-Monitor or hijack the control flow to execute uBOX-Monitor's code, a MemManage exception will be raised and further handled by our reference monitor. The default MPU policies are saved to the read-only memory in Flash, which also meets **G2**.

C. Fault-Based Configuration Protection

Although *u*BOX confines the memory access of untrusted software with the MPU, privileged memory access to system configurations is always permitted (Challenge I in Section I). Therefore, attackers can corrupt the MPU policies to break the isolation with regular store instructions directly. To resolve this challenge and meet **G1**, *u*BOX devises the fault-based configuration protection to confine privileged attackers from tampering with sandbox policies. *u*BOX takes two steps to implement this protection through uBOX-Compiler.

1) Instruction Replacement: uBOX-Compiler eliminates regular store instructions by transforming them into the unprivileged store instructions. It utilizes several LLVM passes to perform instruction analysis and replacement. Note that regular store instructions of uBOX-Monitor are left unmodified. However, the ARMv7-M architecture lacks unprivileged counterparts for store exclusive (STREX) instructions. To address this issue, *u*BOX provides wrapper functions for them. Further elaboration on it will be discussed in Section IV-D1. In this step, unprivileged store instructions are available in other architectures such as ARMv8-M, which satisfies **G3**.

2) Code Scanner: As shown in Fig. 3, uBOX-Compiler integrates a code scanner to check the produced program image and verify that there are no unsafe instructions within the untrusted software. Unsafe instructions consist of regular store instructions and system instructions such as CPS and MSR that can modify system status. Although most unsafe instructions are eliminated by instruction replacement, a few of them still exist as assembly code cannot be handled by LLVM passes. Moreover, the instruction misalignment issue complicates the situation [28], [35]. The ARMv7-M architecture incorporates the Thumb2 instruction set, which supports instruction lengths of either 16-bit or 32-bit while maintaining instruction alignment on a two-byte boundary [32]. A 32-bit instruction can be split from the middle to form new 16-bit or 32-bit instructions, which may potentially be unsafe instructions. Therefore, it is still possible for unsafe instructions to persist within untrusted software. By exploiting them, adversaries can regain the capability of breaking the isolation.

uBOX-Compiler traverses the untrusted code to guarantee the following: 1) The absence of regular store instructions. 2) The absence of system instructions that modify the FAULTMASK register outside of secure gates (Section IV-D) and wrapper functions (Listing 1). Once detecting unsafe instructions, a report will be generated. The report includes details of misaligned instructions, such as their addresses, mnemonics, and encoding. Utilizing this report, developers can employ disassemblers such as IDA Pro [47] and Binary Ninja [48] to quickly find out the source of misaligned instructions and fix them. Our appendix in the supplementary material illustrates a few examples of misaligned unsafe instructions and how to address them. Subsequently, a new program image will be generated and subjected to further verification. This interactive process continues until no unsafe instructions are found, which meets **G1**.

D. State-Based Execution Protection

Nonetheless, attackers may try to abuse uBOX-Monitor's unsafe instructions to break our protection, as those unsafe instructions are unchanged to support normal functionalities (Challenge II in Section I). To prevent such attacks, *u*BOX devises state-based execution protection to establish a secure execution domain for our reference monitor through two primary steps. First, the memory permissions of uBOX-Monitor are set as read-only and non-executable (Section IV-B). As a result, attackers cannot execute any unsafe instructions within it. Second, *u*BOX uses two secure gates to quickly transition between the disabled and enabled states of the MPU to facilitate the normal execution of uBOX-Monitor.

As illustrated in Listing 1 for handling the trusted exceptions, at the entry_gate, the current execution priority is first raised to -1 by setting the FAULTMASK register to 1 (Line 5). Consequently, the MPU is disabled and the execution is confined by the default memory view only (Fig. 4(c)). Subsequently, the CPU context including all the general purpose registers and a status register (xPSR) are saved to the memory which is only writable by uBOX-Monitor (Line 7). Note that registers from co-processors such as the Floating Point Unit (FPU) are not saved as uBOX-Monitor does not use the FPU. Next, the stack is switched to uBOX-Monitor's trusted stack (Line 9). Following this, it invokes the function that handles the exception (Line 12). Upon the completion of exception handling, the control flow will transfer to the exit_gate. The stack is then switched back to the one used by the untrusted software (Line 15). Next, the saved context is restored (Line 17). After

```
1  void Exception_Handler(void) {
2    __asm volatile (
3    entry_gate:
4      /* Set FAULTMASK to 1, raise priority to -1 */
5      CPSID f
6      /* Save context */
7      Save {R0-R12,SP,LR,PC,xPSR}
8      /* Switch to uBOX-Monitor's stack */
9      Switch Stack
10   handler_main:
11     /* Handle the exception by uBOX-Monitor */
12     bl Exception_Handler_Main
13   exit_gate:
14     /* Switch back to untrusted code's stack */
15     Switch Stack
16     /* Restore context */
17     Restore {R0-R12,SP,LR,PC,xPSR}
18     /* Set FAULTMASK to 0, recover priority */
19     CPSIE f
20   );
21 }
```

Listing 1. Secure Gates Designed for Handling (Trusted) Exceptions of *u*BOX-Monitor

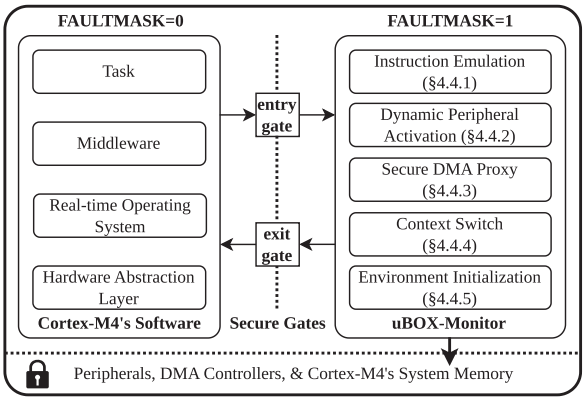

Fig. 5. Runtime protection of *u*BOX-Monitor.

that, the execution priority is recovered (Line 19), thereby re-enabling the MPU. Finally, the control flow will transfer back to the untrusted code.

Our state-based execution protection effectively prevents attackers from abusing unsafe instructions within *u*BOX-Monitor, which meets **G1**. Unlike using code instrumentation approaches, our secure gates efficiently change the MPU states and enable the normal execution of *u*BOX-Monitor, thereby meeting **G4**. The hardware primitives required for the state-based execution protection include the MPU, and the default memory view, which are fundamental features and can be found in other architectures such as ARMv8-M, which satisfies **G3**.

With the secure domain established, *u*BOX-Monitor can run its services securely. Specifically, the services include instruction emulation, dynamical peripherals activation, and the secure DMA proxy, as depicted in Fig. 5. These services uphold the functionality of the untrusted software and confining its memory access, thereby meeting **G1**.

1) Instruction Emulation: As discussed in Section IV-C1, our compiler ensures the absence of regular store instructions, only unprivileged ones remain. Once an unprivileged store instruction writes system configurations, a BusFault exception will be triggered. Consequently, *u*BOX-Monitor utilizes the BusFault exception handler to emulate the unprivileged store instruction.

The process of emulating an unprivileged store instruction is straightforward. Initially, *u*BOX-Monitor verifies the fault status register to validate that the BusFault is indeed triggered by unprivileged access to system configurations. Once confirmed, it proceeds to retrieve the instruction's address and parse the instruction to identify the target address for the write and the value to be written. After that, *u*BOX-Monitor checks the target address to ensure that it is security-insensitive. Specifically, *u*BOX-Monitor prohibits write access to MPU registers and the Vector Table Offset Register (VTOR). Finally, *u*BOX-Monitor writes the parsed value to the target address via a regular store instruction.

Additionally, *u*BOX-Monitor provides wrapper functions for STREX instructions that lack the unprivileged counterparts. STREX instructions are commonly used together with load exclusive (LDREX) instructions and both serve as a synchronization primitive for the ARMv7-M architecture. In particular, an LDREX instruction reads a value from a memory address and tries to tag that address as exclusive. Following this, an STREX instruction attempts to write to a tagged address. The 1-bit return value of the STREX instruction indicates whether the memory write is successful. Otherwise, the LDREX fails to tag the address and the above process should repeat. If an exception occurs between the LDREX and STREX instructions, the exclusive tag will be automatically cleared. Therefore, *u*BOX-Monitor designs wrapper functions for STREX instructions, without the need for exception handling assistance. As shown in Listing 2, the wrapper function first raises its execution priority to -1 by executing a CPS instruction (Line 9). Next it checks the target address that the STREX instruction writes (Line 10). Finally, it invokes the function that executes a STREX instruction by inline assembly. Note that function _strex at Line 1 is located at *u*BOX-Monitor's code memory, which is only executable between Line 9 and Line 16.

```
1  uint32_t _strex(uint32_t value, volatile void *addr) {
2    __asm volatile (
3      "STREX %0, %2, %1" :
4      "=&r" (ret), "=Q" (*addr): "r" (value));
5    return ret;
6  }
7  uint32_t strex(uint32_t value, volatile void *addr) {
8    /* Set FAULTMASK to 1, raise priority to -1 */
9    __asm volatile ("CPSID f");   /* entry_gate */
10   if (check_addr(addr) == True) {
11     ret = _strex(value, addr);
12   }
13   else
14     ret = 1;
15   /* Set FAULTMASK to 0, recover priority */
16   __asm volatile ("CPSIE f");   /* exit_gate */
17   return ret;
18 }
```

Listing 2. Wrapper Function for the STREX Instruction

2) Dynamic Peripheral Activation: As discussed in Section IV-B, the memory of peripherals is designated as read-only by the MPU. Any write to this area will trigger a MemManage exception. *u*BOX-Monitor dynamically activates the peripheral to facilitate the normal execution of untrusted software.

Algorithm 1: Dynamic Peripheral Activation & Secure DMA Proxy.

```
Data:
  Array for peripherals excluding DMA controllers: ArrayPeri
  Array for DMA controllers: ArrayDMAC
  Number of free MPU regions: FreeRegionNum
  ID of the first free MPU region: FreeRegionStart
  ID of the next MPU region for activation: FreeRegionNext
Function DynPeriActivation(inst):
    addr, value ← ParseInst (inst)
    foreach peri ∈ ArrayPeri do
        if peri.base ≤ addr < (peri.base + peri.size) then
            base ← peri.base
            size ← peri.size
            SetMPURegionRW (FreeRegionNext, base, size)
            UpdateFreeRegionNext (&FreeRegionNext,
              FreeRegionNum, FreeRegionStart)
            break
        end
    end
return
Function SecureDMAProxy(inst):
    addr, value ← ParseInst (inst)
    ch_base ← 0                          ▷ DMA channel base address.
    foreach peri ∈ ArrayDMAC do
        if peri.base ≤ addr < (peri.base + peri.size) then
            ch_base ← GetDMAChannelBase (peri.base, addr)
            break
        end
    end
    if ch_base ≠ 0 then
        if DMATXHasEnabled (ch_base) = True or ToEnableDMA
          (ch_base, addr, value) = True then
            if DMATXBenign(ch_base) = True then
                EmulateSTRT (addr, value)
            end
        else
            EmulateSTRT (addr, value)
        end
    end
return
```

This design rationale stems from the observation that a single peripheral may contain multiple memory-mapped registers that will be subsequently accessed. Emulating every individual store instruction that targets the same peripheral will incur unnecessary performance overhead.

The process of dynamic peripheral activation is illustrated in Algorithm 1. First, uBOX-Monitor parses the store instruction that triggers the MemManage exception, which is similar to the process in Section IV-D1. Next, it compares the target address to an array of peripherals (excluding DMA controllers), to identify the correct peripheral. These peripherals are arranged in an ascending order according to their base addresses. Hence, uBOX-Monitor applies the binary search method to match the peripheral. Once the peripheral is identified, uBOX-Monitor configures the MPU region indicated by FreeRegionNext and sets the peripheral as readable and writable. Lastly, it updates the variable FreeRegionNext that indicates the next MPU region for peripheral activation.

3) Secure DMA Proxy: As studied previously, the data transfers facilitated by the DMA are not restricted by the MPU [21], [31]. Consequently, it is important to carefully examine every memory transmission through DMA. We observed that before issuing the data transmission through a DMA channel, the enable bit of the control register associated with that DMA channel must be set to 1. Therefore, uBOX-Monitor only needs to examine the DMA configuration if the DMA transmission is either already enabled or about to be enabled.

As illustrated in Algorithm 1, uBOX-Monitor devises a secure DMA proxy to handle DMA transactions, which meets **G1**. Similarly, it initially compares the target address against an array of DMA controllers. Once the target DMA controller is identified, uBOX-Monitor proceeds to determine the specific DMA channel that the store instruction accesses. Subsequently, it identifies whether the DMA channel is already enabled or if the store instruction intends to enable the DMA channel. If either condition is satisfied, uBOX-Monitor thoroughly evaluates the configuration of that particular DMA channel, ensuring that the memory transmission process never overlaps with security-sensitive memories. Conversely, if both the conditions are not met, uBOX-Monitor simply emulates that (unprivileged) store instruction.

4) Context Switch: *u*BOX also designs API functions that update MPU configurations for FreeRTOS, which meets **G5**. In particular, FreeRTOS employs the PendSV exceptions to facilitate context switching. During a context switch, the MPU is updated for the task that is about to be scheduled. Consequently, the MPU configurations for the task need careful examination. To this end, uBOX-Monitor re-implemented the PendSV exception handler and inserted invocations to the API functions, which verify the MPU configurations for the region configurable by a task, i.e., region 3 in our design (Section IV-B). The verification ensures two key aspects: 1) the memory range covered by region 3 does not overlap with other MPU regions (except for region 0); 2) the memory permissions defined by region 3 disallow code execution.

5) Environment Initialization: *u*BOX sets up the initial execution environment before executing untrusted code. Specifically, uBOX-Compiler inserts an initialization routine before invoking the main function. The routine takes four essential steps. First, it activates the handling of BusFault and MemManage exceptions by setting necessary bits in the System Handler Control and State Register (SHCSR). Note that this write operation is also performed through an unprivileged store instruction. However, the handling of BusFault is not activated at that time and the BusFault exception will escalate to a HardFault exception. Hence this instruction emulation will be performed by the HardFault handler instead, which is also part of uBOX-Monitor. Second, the routine executes an SVC instruction and invokes an SVCall to configure the MPU with default MPU policies. Third, the HFNMIENA bit is set to 0 to ensure that the MPU can be deactivated once the current execution priority is -1. Finally, the execution flow will transfer to the main function and the untrusted code will run with privilege.

V. Implementation

We have implemented a prototype of *u*BOX on the STM32H745I-DISCO board [44], which features a dual-core MCU consisting of a CM7 core and a CM4 core. uBOX-Compiler is implemented as passes based on the LLVM 15.0.0 [49], which includes around 3.0 K lines of C++ code.

`uBOX-Monitor` includes around 2.3 K lines of C and assembly code, which is compatible with FreeRTOS-MPU version 10.3.1. Our system also includes around 700 lines of Python code for generating the default MPU policies.

We also adjust the system initialization procedure for the board. Upon board power-on, each core independently initializes its respective clocks and hardware semaphores. The CM4 core then suspends itself by executing a `WFE` instruction. Subsequently, the CM7 core proceeds to initialize other peripherals. Afterward, the CM7 core notifies the CM4 core by issuing an external interrupt. The CM4 core wakes up upon receiving this interrupt, and both cores continue execution.

Notably, the `WFE` instruction only suspends the processor when the 1-bit `Event Register` is 0. Otherwise, this instruction simply sets the register to 0 without affecting the execution. The `Event Register` can be automatically set to 1 when an exception return occurs or after executing an `SEV` instruction. It can be set to 0 by a system reset or by executing a `WFE` instruction. As `uBOX-Monitor` utilizes the `MemManage` exception to dynamically activate peripherals (Section IV-D2), multiple exception returns occurred before the CM4 core executes the `WFE` instruction. Consequently, the CM4 core will fail to suspend itself. To address this issue, we insert an additional `SEV` followed by a `WFE` instruction before the original `WFE` instruction. This ensures that the `Event Register` is set to 0 when the CM4 core tries to enter into the stop mode.

VI. Evaluation

In this section, we evaluate the efficiency and effectiveness of *u*BOX. In summary, we evaluate our prototype of *u*BOX to answer the following research questions:

- RQ1 What are the security benefits of *u*BOX? (Section VI-A)
- RQ2 How does *u*BOX prevent attacks targeting the sandbox itself? (Section VI-B)
- RQ3 What is the performance overhead of *u*BOX? (Section VI-C)

A. Security Evaluation

For **RQ1**, we evaluated the security benefits of *u*BOX by illustrating how *u*BOX defends various attacks from a compromised CM4 core in this section. We assume that the CM4 core is compromised by attackers through exploiting various vulnerabilities, such as buffer overflow [50], [51], [52] and integer overflow [53], [54], [55]. To this end, we modified application PingPong and inserted additional code to facilitate compromising the CM4 core. In particular, the code receives user input through USART and contains an integer overflow vulnerability.

Due to shared resources among cores, a compromised CM4 core can exploit various heterogeneous attack vectors to compromise the CM7 core. We evaluated the security benefits of our system in 6 attack cases. As shown in Table I, *u*BOX successfully prevents all of them.

Case ❶ and case ❷ depend on the reading capability of the compromised CM4 core. In case ❶, the CM4 core can read shared resources between the two cores directly. Furthermore, it can use MDMA to read the private content from ITCM and DTCM of the CM7 core indirectly (Section II-A). Consequently, the compromised CM4 core in the baseline is able to perform memory dumping targeting the CM7 core with the arbitrary read primitive. In case ❷, attackers can leak sensitive information such as private password or critical data used by the CM7 core with the specific read primitive. However, with *u*BOX enabled, the read capability of the CM4 core will be restricted. If the CM4 core attempts to read memory that does not belong to itself, an `MemManage` exception will be raised and further handled by `uBOX-Monitor`.

TABLE I
Security Evaluation of the Compromised CM4 Core Attacking the CM7 Core

Attack Information		Security Analysis	
Attacks	**Primitives**	**Baseline**	***u*BOX**
❶ Full Memory Dump	Arbitrary Read	✗	✓
❷ Information Leak	Specific Read	✗	✓
❸ Denial of Service	Arbitrary Write	✗	✓
❹ Data Pointer Corrupt	Specific Write	✗	✓
❺ Control Flow Hijack	Specific Write	✗	✓
❻ Code Injection	Specific Read & Write	✗	✓

✗: successful. ✓: prevented.

Case ❸❹❺ depend on the memory write capability of the compromised CM4 core. Attackers can influence the runtime behavior of the CM7 core by corrupting its memory. In case ❸, the Denial of Service attack is straightforward. In particular, the compromised CM4 core can randomly corrupt CM7's memory, which would result in a crash eventually. With further information leaked through case ❷, case ❹❺ would be feasible. In case ❹, once the CM4 core could locate some critical variables, e.g., variables that control the loop, overwriting to such data may mislead the data processing of the CM7 core and even gain a potential arbitrary read/write primitive. Similarly, in case ❺, if attackers could locate and corrupt code pointers (e.g., function pointers) of the CM7 core, its control flow will be hijacked. With the deployment of *u*BOX, CM7's memory would be non-writable by attackers.

Case ❻ code injection attack is a compound result of previous attack vectors and is more complicated. A compromised CM4 core needs to take three steps to perform this attack. First, the malicious payload needs to be injected into CM7's writable memory. Second, attackers need to hijack the control flow of the CM7 core to configure its MPU and set the memory of the previously injected code as executable. Third, attackers need to divert the control flow of the CM7 core into the injected code. *u*BOX successfully defeats this attack as CM7's memory is inaccessible to attackers.

B. Security Analysis of uBOX

Although attacks targeting the CM7 core are prevented by our system, adversaries may compromise *u*BOX itself to break the isolation through control-flow hijacking or data-only attacks. In this section, to answer **RQ2**, we analyze the potential attack vectors of *u*BOX and how our system prevents them.

Control Flow Hijacking Attacks: Adversaries may directly hijack CM4's control flow to execute unsafe instructions and

disable the MPU. For unsafe instructions from the untrusted software, `uBOX-Compiler` employs a code scanner to check the generated program image. It verifies that the untrusted software is free of unsafe instructions (Section IV-C2). For unsafe instructions from the `uBOX-Monitor`, the code memory of `uBOX-Monitor` is set as read-only and non-executable by the MPU (Section IV-B). It guarantees that any unsafe instruction within the code region cannot be executed.

For indirectly attacks, attackers may try to hijack the control flow of `uBOX-Monitor` to further execute unsafe instructions within it. Specifically, attackers may attempt to corrupt code pointers of `uBOX-Monitor` including the function pointers or return addresses. However, `uBOX-Monitor`'s data and stack is configured as read-only when running untrusted software (Section IV-B). Consequently, this attack can be prevented by *u*BOX.

Data-only Attacks: This attack vector can be divided into three parts. First, attackers may try to tamper with security-critical data of `uBOX-Monitor`, such as the `FreeRegionNext` pointer that indicates the next MPU region for peripheral activation (Section IV-D2). Second, attackers may try to corrupt the context that is saved when entering into `uBOX-Monitor`'s exception handlers. Third, attackers may conduct the time-of-check to time-of-use (TOCTTOU) attack by triggering an controlled exception to disrupt the execution of `uBOX-Monitor` and then corrupts `uBOX-Monitor`'s critical data, such as the MPU configurations of an untrusted task. The first two attack vectors can be prevented as the memory of saved context is read-only when running untrusted code. The third one can be prohibited because the execution of `uBOX-Monitor` cannot be interfered with. In particular, the execution priority of `uBOX-Monitor` has been lifted to -1 after executing `entry_gate` (Section IV-D), which is higher than any other configurable exceptions (Section II-C).

C. Performance Evaluation

For **RQ3**, we first evaluate it with macrobenchmarks to understand the overall runtime and memory overhead of *u*BOX. Then we evaluate *u*BOX with microbenchmarks to measure the performance impact of *u*BOX's each component. After that, we evaluate the manual efforts required for eliminating misaligned regular store instructions. All the tests were performed on the STM32H745I-DISCO board [44]. On this board, both CM7 and CM4 cores have 1 MB Flash memory. It also contains 512 KB SRAM for the CM7 core and 288 KB SRAM for the CM4 core. Additionally, it has 64 KB SRAM used for memory sharing. Each tested program is compiled into two different executable files. The first one is generated by vanilla programs and used as the baseline. The second one is compiled with *u*BOX.

1) Macrobenchmarks: Our macrobenchmarks include 6 representative applications, which are adopted from the STM32CubeH7 package [56]. This benchmark selection complies with previous works [20], [23], [29]. Their descriptions are as follows:

FatFs's CM7 core initially establishes a FAT file system. After that, each core creates a file, writes a message to the file, and subsequently reads the message back to verify whether the content remains consistent. FatFs halts execution after each core concludes a comparison of the content read from the FAT file system against its original counterpart. *MDMA_CM4*'s CM4 core fills a large buffer with a specific number through MDMA. Upon completing the memory transmission, the number to be filled in the buffer will be modified for subsequent memory transmissions through MDMA. Concurrently, the CM7 core continuously reads the large buffer to validate if the filled content aligns with expectations. If the content matches the expected value, the CM7 core will blink the green LED. Otherwise, the red LED will blink. MDMA_CM4 stops execution after the CM4 core fills up the large buffer (by MDMA) for 10 times. *Shared_Res*'s two cores operate in tandem, utilizing a shared semaphore to synchronize their access to the USART peripheral. When a core obtains the permission to utilize the USART, it proceeds to print a message through this peripheral while simultaneously causing the green or red LED to blink accordingly. This process continues until the CM4 core has printed 10 messages. Share_Res finishes execution after the CM4 core prints 10 messages through the USART. *Bare_CM4*'s CM7 core executes a FreeRTOS task that periodically transmits a message to the CM4 core via shared memory. The CM4 core receives the message and verifies its content. If the received message is consistent with the expected one, the green LED will be blinked. Otherwise, the red LED is blinked instead. Note that the application code running on the CM4 core is bare-metal. Bare_CM4 terminates its execution once the CM4 core receives the anticipated message from the CM7 core for 10 iterations. *RTOS_Dual*'s CM7 core runs two FreeRTOS tasks. The first task continuously sends messages to the two FreeRTOS tasks on the CM4 core. CM4 Core's two tasks alternate in receiving the messages and validating their content. Whenever the received content corresponds to the expected value, the respective task increments its associated count. Additionally, the CM7 core has a separate check task that periodically reads the counts of the two CM4 tasks. If the counts differ from the previously read ones, the check task signals this by blinking the green LED, indicating that the CM4 core's two tasks are operating normally. Conversely, the red LED is activated. RTOS_Dual finishes execution after either task of the CM4 core receives its expected message from the CM7 core for 10 times. *PingPong*'s CM7 core sends a message to the CM4 core. The message contains a number starting at 0. The CM4 core receives this message, extracts the number, increments it by 1, and subsequently transmits it back to the CM7 core. This back-and-forth message exchange between the two cores continues until the number reaches 2000. *CoreMark* is a performance benchmark for MCUs. We ported CoreMark to PingPong's CM4 core, setting it to run 4000 iterations. The iterations per second reported by it is used as the results of runtime overhead.

Runtime Overhead: We use the DWT unit [57], which is a common hardware feature in Cortex-M MCUs, to collect the number of CPU cycles executed by the CM4 core at runtime for each individual application. Specifically, we gather two distinct timestamps, i.e., before and after the execution of function `main`. Furthermore, we measure additional CPU cycles required for booting the CM4 core, i.e., prior to executing `main`. By

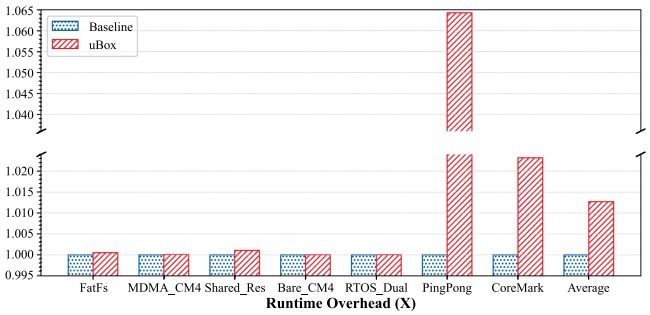

Fig. 6. Runtime overhead of *u*BOX.

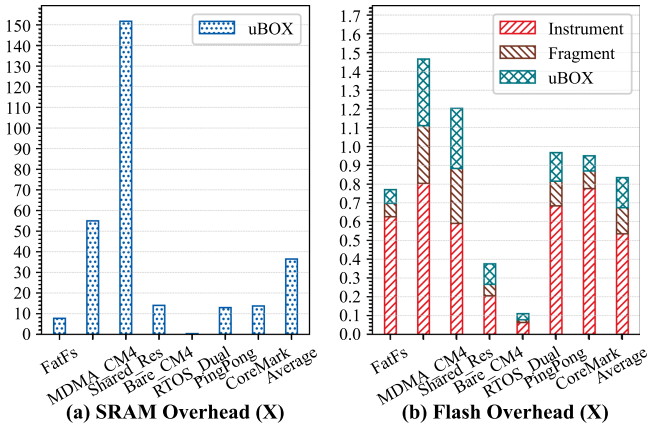

Fig. 7. Memory overhead of *u*BOX.

adding the difference between the two aforementioned timestamps and the increased CPU cycles for system booting, we obtain the increased CPU cycles for *u*BOX. Each application is tested for 5 iterations.

As demonstrated in Fig. 6, the average runtime overhead incurred by *u*BOX is 1.27%, which is negligible. The PingPong application incurs a maximum runtime overhead, which is 6.43%. Such an occurrence can be attributed to the comparatively small number of overall consumed CPU cycles for the PingPong baseline.

SRAM Overhead: We utilized the llvm-size tool to gather section information from both the vanilla and instrumented executable files of each tested application. As demonstrated in Fig. 7(a), the average increase in SRAM usage for *u*BOX amounts to 36.50X. The application with the maximum SRAM overhead is Shared_Res, exhibiting a substantial increase of 151.70X. This is primarily due to the fact that the vanilla version of this application requires only 216 bytes of SRAM, whereas *u*BOX necessitates a considerably larger 32 KB. Conversely, RTOS_Dual showcases the smallest SRAM increase, which is 0.35X. In this case, the vanilla program requires 91.72 KB of SRAM, while *u*BOX requires 32 KB.

As discussed in Section IV-B, a portion of high-address SRAM of the CM4 core is reserved to place the data and stack of `uBOX-Monitor`. The protection is achieved by either disabling the final subregion of region 5 or shrinking the memory of region 5 to uncover this dedicated memory. Consequently, its size accounts for approximately 12.5% (1/8) of the SRAM for the CM4 core, which is constant and will not cause excessive memory pressure at runtime. Despite the relatively high SRAM overhead associated with *u*BOX, the increase in SRAM consumption remains reasonable and acceptable.

To reduce SRAM overhead, an MPU region (1–3), originally for dynamic peripheral activation (Section IV-B), could be reassigned to protect the memory range of `uBOX-Monitor`'s data and stack. Consequently, the SRAM for `uBOX-Monitor`'s data stack could be reduced.

Flash Overhead: The Flash overhead includes code instrumentation, `uBOX-Monitor`'s code, and memory fragmentation. As depicted in Fig. 7(b), the average Flash increase amounts to 0.83X. The maximum Flash overhead is 1.47X in the case of MDMA_CM4. On the other hand, RTOS_Dual presents the minimum Flash overhead, which is 0.11X. Among the tested applications, the primary factor of the Flash overhead is memory fragmentation, which is caused by the strict address alignment requirement imposed by the MPU regions (Section II). To enable code execution of untrusted software, *u*BOX places the code sections (e.g., the `.text` section) to the address aligned to their power of 2 sizes as discussed in Section IV-A. Overall, the Flash overhead demonstrated by *u*BOX is moderate.

2) Microbenchmarks: We design 6 microbenchmarks to understand the performance impact of each individual functionality of *u*BOX. The process of measuring the overhead is similar to the evaluation of macrobenchmarks in Section VI-C1. Each microbenchmark is evaluated for 5 iterations.

Instruction Emulation: To evaluate the additional CPU cycles required for emulating an `STRT` instruction, we use inline assembly code that sets and clears the `USGFAULTENA` bit of the `SHCSR` register, which enables and disables handling the `UsageFault` exception. For the purpose of triggering instruction emulation (Section IV-D1), we use `STRT` instructions to write this bit. In the baseline, we use regular store instructions instead. As the results demonstrated in Table II, our system introduces additional 1963 cycles by emulating one `STRT` instruction twice. This overhead stems from various operations, such as saving and restoring context for handling `BusFault`, analysis of the triggering instruction, verification of the target address, and writing the value to the target address. Context saving involves saving registers to stack while context restoring involves restoring registers from stack (Listing 1).

STREX Wrapper: This microbenchmark measures the overhead of wrapper functions for the `STREX` instruction. We utilize this wrapper (Section IV-D1) in conjunction with a `LDREX`

TABLE II
CONSUMED CPU CYCLES OF MICROBENCHMARKS

Microbenchmark	Baseline	*u*BOX (Increased)
Instruction Emulation	58	2021 (+1963)
STREX Wrapper	112	678 (+566)
Dynamic Peripheral Activation	28	1639 (+1611)
Secure DMA Proxy	1430	31918 (+30488)
System Boot	164	8950 (+8786)
Context Switch	331	927 (+596)

instruction to form an atomic write operation. Conversely, we directly use the STREX instruction in the baseline. Specifically, we employ this operation to consecutively write 1 and 0 to the least significant bit of a 32-bit integer. As shown in Table II, uBOX-Monitor incurs additional 566 cycles. The overhead of the wrapper is significantly lower compared to instruction emulation as it requires no context switch.

Dynamic Peripheral Activation: This benchmark measures the overhead associated with dynamic peripheral activation by writing 0 to the UE bit of the USART control register (USART_CR1). For the application compiled with *u*BOX, the memory write operation to this bit triggers a MemManage exception, which is subsequently handled by uBOX-Monitor (Section IV-D2). According to the results illustrated in Table II, our system incurs an additional 1611 cycles of overhead. This overhead encompasses various factors, including the context switch required for handling the MemManage exception, instruction parsing, the identification of the target peripheral, and the reconfiguration of the reserved MPU region.

Secure DMA Proxy: This synthetic benchmark measures the overhead of a complete DMA operation. Specifically, it involves filling a large buffer with a specific number through MDMA. Once the memory transmission ends, a callback function is invoked to modify a global variable, which indicates the status of the DMA operation. As demonstrated in Table II, *u*BOX incurs an additional 30488 cycles of overhead. Unlike other ordinary peripherals, uBOX-Monitor needs to carefully examine every write operation to the memory-mapped registers of DMA controllers (Section IV-D3).

System Boot: This benchmark measures the overhead of initializing the execution environment by uBOX-Monitor. Specifically, uBOX-Monitor performs two steps for initialization before invoking the main function (Section IV-D1). As depicted in Table II, our system incurs an additional 8786 cycles of overhead. This additional overhead occurs only once.

Context Switch: The context switch in FreeRTOS is executed through the PendSV exception. To quantify the additional overhead of a context switch, we manually add instructions at the beginning and end of the PendSV exception handler to collect the consumed CPU cycles. As shown in Table II, uBOX-Monitor introduces an additional 596 cycles of overhead for a context switch. This overhead arises from the validation of the MPU region settings provided by untrusted tasks and updating the MPU. Specifically, the validation includes checking the memory range and permissions of the provided MPU configurations.

3) Efforts of Removing Misaligned Regular Store Instructions: As discussed in Section IV-C2, our system requires manual work to remove misaligned regular store instructions. To quantify this effort, we measured the prevalence of such instructions across 7 applications in our macrobenchmarks. Specifically, uBOX-Compiler examined 2- or 4-byte sequences at 2-byte intervals from the start of the code section (.text) in each application built with *u*BOX, employing capstone [58] to determine whether those sequences could constitute a regular store instruction. If so, uBOX-Compiler analyzed subsequent instructions to assess their exploitability. If an undefined instruction was reached before a control flow transition instruction (e.g., branches, breakpoints, or pops affecting the PC register), the preceding store instruction was considered as non-exploitable. Otherwise, uBOX-Compiler identified it as a misaligned store instruction requiring further attention.

TABLE III
COUNT OF MISALIGNED REGULAR STORE INSTRUCTIONS

Application	Code Size (KB)	Count
FatFs	77.75	667.0
MDMA_CM4	18.89	144.0
Shared_Res	21.30	140.0
Bare_CM4	52.81	292.0
RTOS_Dual	51.00	287.0
PingPong	37.96	232.0
CoreMark	72.35	392.0
Average	47.44	307.7

Table III shows that the number of misaligned regular store instructions correlates with the size of the .text section. The average count is 307.7. FatFs has the highest number due to the largest .text section of around 77.75 KB. Conversely, Shared_Res has the lowest count, with the second smallest .text section size of around 21.3 KB.

VII. RELATED WORK

Memory Isolation for Embedded Systems: Extensive research has been dedicated to enforcing memory isolation in embedded systems through software-based or hardware-assisted approaches. Regarding software-based methods, memory isolation depends on memory-safe programming languages, such as Webassembly [59] and embedded Rust [60]. eWASM [17] adopts Webassembly to ensure memory safety and type safety at the thread level. Additionally, CRT-C [18] utilizes CheckedC [61] to constrain memory access for each compartment. While suitable for new development, these approaches generally require a re-implementation of existing code (including RTOS) which hinders adaptability.

Hardware-assisted approaches leverage hardware features, such as the MPU and privilege modes, to facilitate memory isolation. For instance, MINION [19] ensures memory isolation for the RTOS-based embedded systems at the thread-level. ACES [20] compartmentalizes the program into compartments and enforces compartment-level memory isolation. ACES supports flexible policies to partition a program. Similarly, OPEC [23] divides the program at the operation granularity. To achieve fine-grained memory isolation, OPEC generates shadow copies of shared global variables of each compartment. Since memory transmission through DMA is not regulated by the MPU, D-BOX [21] designs secure DMA APIs for untrusted software to protect embedded systems from DMA based attacks. However, malicious input from peripherals can also threaten embedded systems. To defend such attacks, M2MON [22] sanitizes the malicious writes from untrusted code to memory-mapped registers of peripherals and filters out malicious signals from the physical world.

The aforementioned hardware-assisted approaches necessitate privilege separation to protect their reference monitors.

Consequently, untrusted code is forced to execute at the unprivileged level. To execute privileged instructions from untrusted code properly, frequent context switches are required. Intra-address space mechanisms, which do not require privilege separation, are more efficient. EC [36] runs untrusted compartments and the reference monitor at the same privileged level. To constrain privileged attackers, EC employs the debug watchpoint to trap the write operations targeting security-sensitive system configurations. Any attempt to modify the sensitive memory will trigger a DebugMon exception, which is further managed by EC. SHERLOC [62] utilizes the Micro Trace Buffer feature of the ARMv8-M architecture to monitor the control flow of both unprivileged and privileged code from the non-secure world. Its reference monitor runs within the secure world. *u*BOX achieves memory isolation and protects its reference monitor at the same privileged level with untrusted code.

Security Mitigation for Embedded Systems: Apart from enforcing memory isolation, various research works mitigate specific attacks to embedded systems. Shen et al. [63] achieves Execute-Only-Memory (XOM) against information leak attacks. Specifically, it uses the debug watchpoint to monitor all the read operations to the protected code memory. uXOM [28] utilizes unprivileged load instructions to deprive the read capability of untrusted code, while simultaneously keeping code executable. Additionally, uXOM employs unprivileged store instructions to protect MPU configurations similar to *u*BOX. RECFISH [64], uRAI [29], and Silhouette [30] protect the return address integrity of embedded systems. In particular, *u*BOX uses the isolated memory domain to securely execute the reference monitor for SFI. Kage [31] enforces holistic control flow integrity on RTOS-based embedded systems. Furthermore, HERA [65] and RapidPatch [66] propose hotpatch approaches, which patch the read-only code of embedded systems without modification at runtime. EPOXY [67], Randezvous [68], and HARM [69] use randomization techniques to defend against code reuse attacks. These studies are orthogonal to *u*BOX.

Strengthened Security Primitives: Plenty of research works aim to propose new or strengthen existing secure primitives for embedded devices to extend their applications. DICE [70] and Lazarus [71] provide mechanisms for regaining control over remotely deployed embedded devices, even if they suffer complete compromise. These protection mechanisms are independent of memory isolation methods and complement our system. PISTIS [72] proposes a software based security architecture which enables efficient memory isolation. In particular, it targets TI MSP430 MCUs [73], which use an instruction set where indirect memory access and indirect control flow transfer can be distinguished from their direct counterparts, setting it apart from the ARMv7-M architecture. MyTEE [74] establishes a trusted execution environment on the ARMv8-A based embedded systems that lack critical TrustZone extensions. This is accomplished by carefully using the 2-stage address translation and the secure monitor mode. RT-TEE [75] strengthens the ARM TrustZone with availability, ensuring the timely completion of real-time tasks. *u*BOX uses a novel approach to run the reference monitor at a permissive memory view and confines privileged attackers with the MPU.

VIII. Discussion

Applications Modifying the MPU: Currently, our reference monitor prohibits applications from writing to MPU registers. Therefore, any attempt by an application to set the MPU will be denied. However, some applications may require MPU customization. To accommodate these, we can repurpose one of the MPU regions (1-3), initially for dynamic peripheral activation (Section IV-D2), for applications needing customized MPU configurations. Applications will be confined to this specific region's configuration without disrupting the existing security policy.

Confidentiality of the Reference Monitor: Currently, the confidentiality of the `uBOX-Monitor` cannot be guaranteed. This limitation arises from the limited number of MPU regions available in the ARMv7-M architecture. In particular, the memory permissions for the `uBOX-Monitor`'s code and data are set to read-only and non-executable when executing untrusted software. Although the integrity of the `uBOX-Monitor`'s code and data is ensured, attackers can still freely read the memory of our reference monitor, thereby compromising its confidentiality. Note that in MCUs equipped with the MPU with more than 8 MPU regions, such as the Cortex-M33 [76] MCU, two dedicated MPU regions can be reserved to establish no-access permissions for `uBOX-Monitor`'s memory. Therefore, any unauthorized memory read to `uBOX-Monitor` will be prohibited.

Scalability of uBOX: Our system utilizes unprivileged store instructions and the MPU bypass feature to create a secure environment for `uBOX-Monitor`. It is compatible with ARMv8-M Main extension MCUs, which include both features [37]. However, it's not applicable to ARMv8-M Baseline [77] or ARMv6-M [78] MCUs due to the absence of the MPU bypass feature. In RISC-V MCUs, Physical Memory Protection (PMP) [79] can be employed, akin to the MPU, to set memory permissions over physical addresses. To establish a secure execution environment at the M-mode, where unsafe operating systems may operate, the Smepmp extension and PMP configuration locking feature can be used. Unlike the MPU, PMP is configured via Control and Status Register (`CSR`) instructions, not memory load and store instructions. Hence, any `CSR` instructions that alter the PMP must be substituted with call gates, as there are no unprivileged equivalents for CSR instructions.

Harden Existing Protections: Recent security protections for ARMv7-M based embedded systems (e.g., uXOM [28]) rely on SFI to instrument unaltered regular store instructions, thereby safeguarding the MPU configurations. Moreover, additional mechanisms are required to enforce control flow integrity to ensure that the SFI cannot be circumvented. In contrast, *u*BOX uses the state-based execution protection to prevent regular store instructions and other system instructions of `uBOX-Monitor` from being abused by attackers. In the future, we plan to improve existing protections with our state-based execution protection.

IX. Conclusion

Multicore embedded systems employ a heterogeneous architecture that consolidate varying performance cores into one MCU. These cores serve different purposes, with LITTLE cores

processing external inputs and big cores handling computation-intensive tasks. Due to the resource sharing among cores, the compromise of one core will affect the whole system. Existing hardware-assisted SFI mechanisms adopt the MPU to enforce memory isolation over untrusted software. However, they rely on privilege separation and code instrumentation to protect the isolation policies from unauthorized modifications.

In this paper, we propose *u*BOX, a lightweight sandbox for multicore embedded systems. Similar to previous works, *u*BOX restricts the memory access of untrusted software through the MPU. On the contrary, *u*BOX isolates the untrusted software at the same privileged level. To prevent attackers from modifying isolation policies, *u*BOX devises the fault-based configuration protection and traps any write access from untrusted software towards system configurations. To prevent attackers from abusing `uBOX-Monitor`'s code, *u*BOX devises the state-based execution protection to securely host `uBOX-Monitor` with the default memory view. Reusing `uBOX-Monitor`'s code by attackers is prohibited by the MPU as its memory is configured as read-only and non-executable when running untrusted software. Our evaluation demonstrates that *u*BOX incurring negligible runtime overhead, moderate Flash overhead, and reasonable and constant SRAM overhead.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful comments and feedback. Additionally, the first author of this paper extends personal thanks to Dr. Yuan Chen, Dingding Wang, Huamao Wu, and Jinyan Xu for their assistance with proofreading and discussions. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

REFERENCES

[1] Apple, "iPhone 14 pro," 2022. [Online]. Available: https://www.apple.com/hk/en/iphone-14-pro/specs/

[2] Qualcomm, "Snapdragon 8 gen 2 mobile platform," 2023. [Online]. Available: https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-8-gen-2-mobile-platform

[3] Hisilicon, "Qirin 9000," 2023. [Online]. Available: https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-9000

[4] Samsung, "Exynos 1380," 2023. [Online]. Available: https://semiconductor.samsung.com/processor/mobile-processor/exynos-1380/

[5] Intel, "Performance hybrid architecture," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/hybrid-architecture.html

[6] A. Semiconductor, "Alif ensemble family," 2023. [Online]. Available: https://alifsemi.com/ensemble/

[7] NXP, "i. MX 8 M nano family - arm cortex-A53, cortex-M7," 2023. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-8-applications-processors/i-mx-8m-nano-family-arm-cortex-a53-cortex-m7:i.MX8MNANO

[8] STMicroelectronics, "STM32H7 series," 2023. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32h7-series.html

[9] P. Zero, "Over the air: Exploiting broadcom's Wi-Fi stack (Part 1)," 2017. [Online]. Available: https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html

[10] P. Zero, "Over the air-vol. 2, PT 1: Exploiting the Wi-Fi stack on apple devices," 2017. [Online]. Available: https://googleprojectzero.blogspot.com/2017/09/over-air-vol-2-pt-1-exploiting-wi-fi.html

[11] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proc. 14th ACM Symp. Operating Syst. Princ.*, 1993, pp. 203–216.

[12] B. Yee et al., "Native client: A sandbox for portable, untrusted x86 native code," in *Proc. IEEE Symp. Secur. Privacy*, 2009, pp. 79–93.

[13] D. Sehr et al., "Adapting software fault isolation to contemporary CPU architectures," in *Proc. 19th USENIX Secur. Symp.*, 2010, pp. 1–12.

[14] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "RockSalt: Better, faster, stronger SFI for the x86," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2012, pp. 395–404.

[15] B. Niu and G. Tan, "Monitor integrity protection with space efficiency and separate compilation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 199–210.

[16] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "ARMlock: Hardware-based fault isolation for ARM," in *Proc. 21st ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 558–569.

[17] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, and L. Cherkasova, "eWASM: Practical software fault isolation for reliable embedded devices," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3492–3505, Nov. 2020.

[18] A. Khan, D. Xu, and D. Tian, "Low-cost privilege separation with compile time compartmentalization for embedded systems," in *Proc. IEEE Symp. Secur. Privacy*, 2023, pp. 3008–3025.

[19] C. H. Kim et al., "Securing real-time microcontroller systems through customized memory view switching," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[20] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: Automatic compartments for embedded systems," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 65–82.

[21] A. Mera, Y. H. Chen, R. Sun, E. Kirda, and L. Lu, "D-Box: DMA-enabled compartmentalization for embedded applications," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2022, pp. 1–17.

[22] A. Khan, H. Kim, B. Lee, D. Xu, A. Bianchi, and D. J. Tian, "M2MON: Building an MMIO-based security reference monitor for unmanned vehicles," in *Proc. USENIX Secur. Symp.*, 2021, pp. 285–302.

[23] X. Zhou, J. Li, W. Zhang, Y. Zhou, W. Shen, and K. Ren, "OPEC: Operation-based security isolation for bare-metal embedded systems," in *Proc. 7th Eur. Conf. Comput. Syst.*, 2022, pp. 317–333.

[24] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 1–40, 2009.

[25] N. Burow et al., "Control-flow integrity: Precision, security, and performance," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 1–33, 2017.

[26] N. Burow, X. Zhang, and M. Payer, "SoK: Shining light on shadow stacks," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 985–999.

[27] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proc. 10th ACM Symp. Inf., Comput. Commun. Secur.*, 2015, pp. 555–566.

[28] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, "uXOM: Efficient eXecute-only memory on ARM cortex-M," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 231–247.

[29] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "uRAI: Securing embedded systems with return address integrity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–18.

[30] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 1219–1236.

[31] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell, "Holistic control-flow protection on real-time embedded systems with kage," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 2281–2298.

[32] Arm, "ARMv7-M architecture reference manual," 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0403/latest/

[33] Arm, "Unprivileged loads and stores," 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0403/d/Application-Level-Architecture/The-ARMv7-M-Instruction-Set/Load-and-store-instructions/Unprivileged-loads-and-stores

[34] Hard Find Electronics Ltd., "Top 5 MCU manufacturers view for the development of the microcontroller market," 2024. [Online]. Available: https://www.hardfindelec.com/a/76030.html

[35] Z. B. Aweke and T. Austin, "uSFI: Ultra-lightweight software fault isolation for IoT-Class devices," in *Proc. Des., Automat. Test Europe Conf. Exhib.*, 2018, pp. 1015–1020.

[36] A. Khan, D. Xu, and D. Tian, "EC: Embedded systems compartmentalization via intra-kernel isolation," in *Proc. IEEE Symp. Secur. Privacy*, 2023, pp. 2990–3007.
[37] Arm, "Armv8-M architecture reference manual," 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0553/latest/
[38] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 241–254.
[39] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "EPK: Scalable and efficient memory protection keys," in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 609–624.
[40] D. Schrammel et al., "Donky: Domain keys–efficient in-process isolation for RISC-V and x86," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 1677–1694.
[41] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1221–1238.
[42] Z. Yuan, S. Hong, R. Chang, Y. Zhou, W. Shen, and K. Ren, "VDom: Fast and unlimited virtual domains on multiple architectures," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 905–919.
[43] S. Park, S. Lee, and T. Kim, "Memory protection keys: Facts, key extension perspectives, and discussions," *IEEE Secur. Privacy*, vol. 21, no. 3, pp. 8–15, May/Jun. 2023.
[44] Arm, "STM32H745I-DISCO, discovery kit with STM32H745XI MCU," 2019. [Online]. Available: https://www.st.com/en/evaluation-tools/stm32h745i-disco.html
[45] Arm, "STM32H745/755," 2019. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32h745--755.html
[46] EEMBC, "CPU benchmark–MCU benchmark – CoreMark – EEMBC embedded microprocessor benchmark consortium," 1997. [Online]. Available: https://www.eembc.org/coremark/
[47] Hex-Rays, "IDA pro," 2024. [Online]. Available: https://hex-rays.com/ida-pro/
[48] V. 35, "Binary ninja," 2024. [Online]. Available: https://binary.ninja/
[49] LLVM, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim.*, 2004, pp. 75–86.
[50] CVE, "CVE-2018–16528," 2018. Accessed: Oct. 7, 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16528
[51] CVE, "CVE-2018–16525," 2018. Accessed: Oct. 7, 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16525
[52] CVE, "CVE-2018–16526," 2018. Accessed: Oct. 7, 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16526
[53] CVE, "CVE-2021–31572," 2021. Accessed: Oct. 7, 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31572
[54] CVE, "CVE-2020–10062," 2020. Accessed: Oct. 7, 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-10062
[55] CVE, "CVE-2020–10067," 2020. Accessed: Oct. 7, 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-10067
[56] STMicroelectronics, "STM32CubeH7," 2023. Accessed: Jun. 17, 2023. [Online]. Available: https://www.st.com/zh/embedded-software/stm32cubeh7.html
[57] Arm, "Data watchpoint and trace unit," 2023. [Online]. Available: https://developer.arm.com/documentation/ddi0439/b/Data-Watchpoint-and-Trace-Unit
[58] Capstone, "Capstone: The ultimate disassembler," 2024. [Online]. Available: https://www.capstone-engine.org/
[59] A. Haas et al., "Bringing the web up to speed with WebAssembly," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2017, pp. 185–200.
[60] E. Rust, "Rust for embedded systems," 2023. [Online]. Available: https://www.rust-lang.org/what/embedded
[61] Microsoft, "Checked C," 2015. [Online]. Available: https://www.microsoft.com/en-us/research/project/checked-c/
[62] X. Tan and Z. Zhao, "SHERLOC: Secure and holistic control-flow violation detection on embedded systems," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2023, pp. 1332–1346.
[63] Z. Shen, K. Dharsee, and J. Criswell, "Fast execute-only memory for embedded systems," in *Proc. IEEE Secure Develop.*, 2020, pp. 7–14.
[64] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-flow integrity for real-time embedded systems," in *Proc. 31st Euromicro Conf. Real-Time Syst.*, 2019, pp. 2:1–2:24.
[65] C. Niesler, S. Surminski, and L. Davi, "HERA: Hotpatching of embedded real-time applications," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–16.
[66] Y. He et al., "RapidPatch: Firmware hotpatching for real-time embedded devices," in *Proc. 31th USENIX Secur. Symp.*, 2022, pp. 2225–2242.
[67] A. A. Clements et al., "Protecting bare-metal embedded systems with privilege overlays," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 289–303.
[68] Z. Shen, K. Dharsee, and J. Criswell, "Randezvous: Making randomization effective on MCUs," in *Proc. 38th Annu. Comput. Secur. Appl. Conf.*, 2022, pp. 28–41.
[69] J. Shi, L. Guan, W. Li, D. Zhang, P. Chen, and N. Zhang, "HARM: Hardware-assisted continuous re-randomization for microcontrollers," in *Proc. IEEE 7th Eur. Symp. Secur. Privacy*, 2022, pp. 520–536.
[70] M. Xu et al., "Dominance as a new trusted computing primitive for the Internet of Things," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 1415–1430.
[71] M. Huber, S. Hristozov, S. Ott, V. Sarafov, and M. Peinado, "The Lazarus effect: Healing compromised devices in the internet of small things," in *Proc. 15th ACM Asia Conf. Comput. Commun. Secur.*, 2020, pp. 6–19.
[72] M. Grisafi, M. Ammar, M. Roveri, and B. Crispo, "PISTIS: Trusted computing architecture for low-end embedded systems," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 3843–3860.
[73] T. Instruments, "MSP430 microcontrollers," 2023. Accessed: Jun. 17, 2023. [Online]. Available: https://www.ti.com/microcontrollers-mcus-processors/msp430-microcontrollers/overview.html
[74] S. Han and J. Jang, "MyTEE: Own the trusted execution environment on embedded devices," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–15.
[75] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang, "RT-TEE: Real-time system availability for cyber-physical systems using arm trustzone," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 352–369.
[76] Arm, "Cortex-M33," 2023. [Online]. Available: https://developer.arm.com/documentation/100230/latest/
[77] Arm, "Introduction to the Armv8-M architecture and its programmers model," 2021. [Online]. Available: https://developer.arm.com/documentation/107656/0101/Introduction-to-Armv8-M-architecture/Architecture-and-micro-architecture/Architecture
[78] Arm, "ARMv6-M architecture reference manual," 2018. [Online]. Available: https://developer.arm.com/documentation/ddi0419/latest/
[79] R.-V. Foundation, "The RISC-V instruction set manual, volume II: Privileged architecture, version 20240411," 2024. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/20240411/priv-isa-asciidoc.pdf

Xia Zhou received the BE degree in information security from Sichuan University, in 2018. He is currently working toward the doctoral degree in cyberspace security with Zhejiang University. His current research interests lie in the hardware-assisted security isolation on embedded systems.

Yujie Bu received the BA degree in broadcasting and TV from Zhejiang University, in 2021. He is currently working toward the PhD degree with the Hong Kong Polytechnic University. His current research interests lie in the OS-hardware boundary security and distributed system security like blockchain.

Meng Xu (Member, IEEE) received the PhD degree from the Georgia Institute of Technology, in 2020. He is an assistant professor with the Cheriton School of Computer Science, University of Waterloo, Canada. His research is in the area of system and software security, with a focus on delivering high-quality solutions to practical security programs, especially in finding and patching vulnerabilities in critical computer systems. This usually includes research and development of automated program analysis testing verification tools that facilitate the security reasoning of critical programs.

Yajin Zhou received the PhD degree in computer science from North Carolina State University, Raleigh, NC, USA. He is currently a ZJU 100 young professor with the School of Cyber Science and Technology, and the College of Computer Science and Technology, Zhejiang University, China. His research mainly focuses on smartphone and system security, such as identifying real-world threats and building practical solutions, mainly in the context of embedded systems (or IoT devices).

Lei Wu received the PhD degree from North Carolina State University, in 2015. He is an associate professor with the School of Cyber Science and Technology, and the College of Computer Science and Technology, Zhejiang University, China. His research interest lies mainly in security areas, including system security and blockchain security.