SHERLOC: Secure and Holistic Control-Flow Violation Detection on Embedded Systems

Xi Tan
CactiLab, University at Buffalo
Buffalo, USA
xitan@buffalo.edu

Ziming Zhao
CactiLab, University at Buffalo
Buffalo, USA
zimingzh@buffalo.edu

ABSTRACT

Microcontroller-based embedded systems are often programmed in low-level languages and are vulnerable to control-flow hijacking attacks. One approach to prevent such attacks is to enforce control-flow integrity (CFI), but inlined CFI enforcement can pose challenges in embedded systems. For example, it increases binary size and changes memory layout. Trace-based control-flow violation detection (CFVD) offers an alternative that doesn't require instrumentation of the protected software or changes to its memory layout. However, existing CFVD methods used in desktop systems require kernel modifications to store and analyze the trace, which limits their use to monitoring unprivileged applications. But, embedded systems are interrupt-driven, with the majority of processing taking place in the privileged mode. Therefore, it is critical to provide a holistic and system-oriented CFVD solution that can monitor control-flow transfers both within and among privileged and unprivileged components.

In this paper, we present SHERLOC, a Secure and Holistic Control-Flow Violation Detection mechanism designed for microcontroller-based embedded systems. SHERLOC ensures security by configuring the hardware tracing unit, storing trace records, and executing the violation detection algorithm in a trusted execution environment, which prevents privileged programs from bypassing monitoring or tampering with the trace. We address the challenges of achieving holistic and system-oriented CFVD by formalizing the problem and monitoring forward and backward edges of unprivileged and privileged programs, as well as control-flow transfers among unprivileged and privileged components. Specifically, SHERLOC overcomes the challenges of identifying legitimate asynchronous interrupts and context switches at run-time by using an interrupt- and scheduling-aware violation detection algorithm. Our evaluations on the ARMv8-M architecture demonstrate the effectiveness and efficiency of SHERLOC.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**; **Operating systems security**.

CCS '23, November 26–30, 2023, Copenhagen, Denmark.

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/XXXXXX.XXXXXX

KEYWORDS

Control-flow violation detection; hardware tracing unit; trusted execution environment

ACM Reference Format:
Xi Tan and Ziming Zhao. 2023. SHERLOC: Secure and Holistic Control-Flow Violation Detection on Embedded Systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

1 INTRODUCTION

Microcontroller-based embedded and Internet of Things (IoT) systems lack memory management units (MMU) and memory virtualization, yet they are ubiquitous and predicted to reach 1 trillion by 2035 [44]. These systems are usually written in low-level languages, e.g., C, whose lack of safety allow attackers to exploit memory corruption bugs to hijack the control flow [45]. While data execution prevention (DEP) and W⊕X can defeat code injection attacks, these systems are still vulnerable to code reuse attacks, e.g., return-oriented programming (ROP) [12, 18, 50]. The situation is exacerbated when many of these software systems are real-time operating systems (RTOS) that are compiled and statically linked with applications or bare-metal systems where applications execute directly on hardware without an OS. Due to the lack of security and fault isolation, a bug anywhere in such systems may lead to serious consequences.

Control-flow integrity (CFI) is a security property that can prevent control-flow hijacking by dictating that indirect control-flow transfers, including forward edges (indirect call and branch) and backward edges (return), must follow a predetermined control-flow graph (CFG) [2–4]. The CFI property can be enforced by either inlined instrumentation or trace-based control-flow violation detection (CFVD) mechanisms. The former instruments the application or kernel software with run-time checks through static and/or dynamic source code or binary rewriting, while the latter relies on hardware tracing features to capture and verify indirect control-flow transfers.

The inlined instrumentation mechanism inserts a label at each destination and a dynamic check before each source to ensure that the run-time destination has the correct label for forward edges [4]. A variety of policies, which provides different levels of precision to balance the trade-off between security and performance, have been proposed and implemented for forward edges [13, 19, 24–26, 30, 36, 46, 54, 56]. To ensure that a return transfers back to its invoking callsite, backward CFI enforcement usually inserts shadow stacks [4, 14, 17, 21] or return address integrity [5] maintenance and checks in the prologue and epilogue of a function.

Inlined instrumentation, unfortunately, has several limitations when deployed on embedded systems: (i) instrumentation increases the binary size and changes the memory layout, which is sometimes infeasible with a small memory of hundred Kilobytes. To preserve the memory layout, CaRE [38], which only monitors bare-metal systems, does not insert but replaces function calls and indirect branches with dispatch instructions, incurring a high performance overhead; (ii) they usually require the source code of the protected system, which IoT vendors are reluctant to share, for compile time instrumentation. In addition, it is difficult to protect shadow stacks in such approaches. RECFISH [48], which only protects unprivileged code, maintains shadow stacks at the privileged level. TzmCFI [29] maintains shadow stacks in the TrustZone secure state and introduces high run-time overhead. Both Silhouette [57] and Kage [20] use unprivileged store instructions introduced in ARMv7-M to achieve a low overhead; nonetheless, Silhouette only works for bare-metal systems and Kage only supports a small number of tasks due to the high memory overhead.

The trace-based control-flow violation detection mechanism analyzes the instruction trace, which is usually collected by a hardware unit. This approach doesn't require instrumenting the protected software, source code, or altering its memory layout, making it a promising solution to enforce CFI on embedded systems. However, existing CFVD approaches used in desktop systems [23, 27, 34, 47, 51, 53] require kernel modifications to store and analyze the trace; hence, they can only monitor unprivileged applications, referred to as *application-oriented* CFVD. Therefore, they are not directly applicable to embedded systems where it is necessary to monitor the control-flow transfers *within* and *among* privileged and unprivileged components, referred to as *system-oriented* CFVD. Additionally, to improve performance, their policies mandate analyzing only those traces that result in specific system calls. Some techniques, such as CFIGuard [53] and PathArmor [47], do not store traces in memory but instead use special registers. However, these methods are susceptible to history flushing attacks [41, 42].

In this paper, we present Sherloc – Secure and Holistic Control-Flow Violation Detection for microcontroller-based embedded systems. To ensure *security*, Sherloc configures the hardware tracing unit, stores the trace records, and executes the CFVD algorithm in a trusted execution environment (TEE), so even non-secure state privileged program cannot bypass the monitoring or tamper the traces. To achieve *holistic* monitoring, Sherloc provides a mechanism that not only monitors the forward and backward edges of unprivileged and privileged programs but also the control-flow transfers among unprivileged and privileged components. Specifically, Sherloc addresses the challenges of identifying legitimate asynchronous interrupts and context switches among applications at run-time with an interrupt- and scheduling-aware violation detection algorithm. To improve performance, Sherloc can also enforce more practical policies, such as analyzing traces only when certain operations, such as changing system registers, are triggered. In addition, we model the detection latency of Sherloc and estimate its upper bound.

We have implemented the offline analysis module of Sherloc using both static and dynamic binary analysis techniques to generate an over-approximation of the interprocedural control-flow graph for protected systems. Additionally, we have implemented the runtime modules of Sherloc on the ARMv8-M architecture [6] with necessary hardware features such as TrustZone as the TEE and macro trace buffer (MTB) as the hardware tracing unit. It is worth noting that the concept of Sherloc can be applied to any embedded architecture with a hardware tracing unit and a secure enclave to perform tracing record analysis. To assess the effectiveness and performance of Sherloc, we conducted an evaluation on a single-core Cortex-M33 microcontroller, employing the BEEBS benchmark suite [39], bare-metal systems [9], and the FreeRTOS profile with preemptive scheduling [22]. Our evaluation results demonstrate that Sherloc can identify control-flow hijacking attacks with satisfactory performance.

The contributions of this paper are summarized as follows:

- We identify and formalize the problem of *system-oriented control-flow violation detection*, which enforces holistic control-flow integrity for microcontroller-based multi-tasking systems, with or without privilege separation. The system-oriented CFVD approach overcomes the limitations of existing *application-oriented* CFVD approaches by extending the monitoring to privileged code;
- We present Sherloc, a secure and interrupt- and scheduling-aware CFVD mechanism that protects itself from malicious privileged program. Sherloc monitors both forward and backward edges of both unprivileged and privileged programs, as well as synchronous and asynchronous control-flow transfers among unprivileged and privileged components. Additionally, we develop a model for estimating the detection latency of Sherloc and determine its upper bound;
- We implement Sherloc for the ARMv8-M architecture and evaluate its performance using embedded benchmark programs, bare-metal systems, and a real-time operating system. Our evaluation results demonstrate that Sherloc is secure, effective, and efficient. We open-source Sherloc.[1]

2 SYSTEM-ORIENTED CONTROL-FLOW VIOLATION DETECTION

In this section, we formalize the definitions of the existing application-oriented CFVD mechanism and the proposed system-oriented CFVD mechanism. In addition, we discuss the motivations and challenges of the proposed mechanism.

2.1 Application-oriented CFVD (ACFVD)

Current CFVD mechanisms for desktop systems with memory virtualization only monitor a specific userland application. We model the interprocedural CFG of such an application $\mathcal{A}$ as $G_{\mathcal{A}} = (V_{\mathcal{A}}, E_{\mathcal{A}})$, where $V_{\mathcal{A}}$ is the set of basic blocks and $E_{\mathcal{A}}$ is the set of control-flow transfers defined by the application. Existing approaches configure hardware tracing units, such as Intel Processor Trace [28], to record only certain control-flow transfers, such as indirect calls/jumps and returns, within a single application $\mathcal{A}$ and exclude synchronous and asynchronous control-flow transfers of other applications or the kernel. We model each trace record as a 2-tuple $r = \langle s, d \rangle$ where s is the virtual source address and d is the virtual destination address.

[1] https://github.com/CactiLab/Sherloc-Cortex-M-CFVD

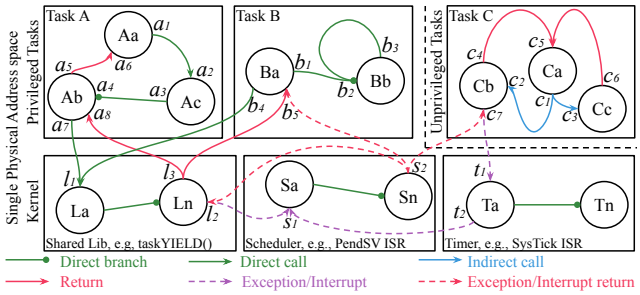


Figure 1: Example legitimate control-flow transfers of a system with an RTOS kernel, two privileged tasks, and one unprivileged task in a single physical address space.

The ACFVD solution verifies each indirect control-flow transfer must match an edge in the $G_{\mathcal{A}}$:

Application-oriented CFVD. Given the trace $R_{\mathcal{A}} = (r_0, r_1, \ldots, r_n)$ of an application $\mathcal{A}$, ACFVD verifies that $r_i \in E_{\mathcal{A}}, \forall i \in \{0, 1, \ldots, n\}$.

ACFVD approaches require the modification of the kernel to capture and analyze traces, which makes it difficult to extend them to monitor privileged code or the kernel itself. Furthermore, to achieve better performance, many solutions perform incomplete monitoring and mandate analyzing only the traces that lead to specific system calls, such as *execve*, *mmap*, and *mprotect*.

2.2 System-oriented CFVD (SCFVD)

Due to the lack of memory virtualization, microcontroller-based systems have all software components residing in the same physical address space. In current practice, a software system for a microcontroller, which includes an RTOS and applications (referred to as tasks) running on top, is compiled and statically linked into a single binary program. As a result, a bug anywhere may lead to serious consequences. Furthermore, microcontroller-based systems are interrupt-driven, with the majority of processing taking place in interrupt service routines (ISR). Therefore, it is critical to provide a holistic and system-oriented CFVD solution that can monitor control-flow transfers both within and among privileged and unprivileged components.

However, an SCFVD solution faces many challenges as interrupts and task scheduling occur asynchronously and cannot be anticipated through static CFG analysis and/or dynamic CFG training [27, 34]. Figure 1 depicts a motivating example with an RTOS kernel and three running tasks in a single physical address space. Even if the interprocedural CFG can include an over-approximated indirect control-flow transfer set (e.g., $\langle c_1, c_2 \rangle$, $\langle c_1, c_3 \rangle$ in Figure 1), asynchronous interrupts can happen anytime (e.g., $\langle c_7, t_1 \rangle$). Additionally, unlike a function or interrupt return, where each return goes to the most recent callsite or interrupted instruction, the scheduler may yield control to any running tasks. For instance, both $\langle s_2, b_5 \rangle$, i.e., scheduling task B, and $\langle s_2, c_7 \rangle$, i.e., scheduling task C, are legitimate control-flow transfers in Figure 1. Therefore, an interrupt- and scheduling-aware control-flow violation detection mechanism is necessary to address these challenges.

We model such a microcontroller-based software system $\mathcal{S}$ including a kernel $\mathcal{K}$ and tasks $\mathcal{T}$ as $(G_{\mathcal{S}}, I_{\mathcal{K}}, Y_{\mathcal{T}})$, where $G_{\mathcal{S}}$ is its interprocedural CFG, $I_{\mathcal{K}}$ is the set of *asynchronous* kernel interrupt service routine addresses (e.g., timer handler), and $Y_{\mathcal{T}}$ is the set of task entry or re-entry addresses. In particular, $G_{\mathcal{S}}$ is modeled as $(V_{\mathcal{S}}, E_{\mathcal{S}})$, where $V_{\mathcal{S}}$ is the set of basic blocks of tasks and the kernel, and $E_{\mathcal{S}}$ is the set of control-flow transfers defined by the tasks and the kernel. Please note that $E_{\mathcal{S}}$ also models *synchronous* exceptions that involve control-flow transfers across privilege levels, such as system calls. At system boot, $Y_{\mathcal{T}} = Y_{\mathcal{TE}} \bigcup Y_{\mathcal{TR}}$ is initially composed of statically retrieved entry addresses of all tasks $Y_{\mathcal{TE}}$, which may be dynamically replaced by re-entry addresses $Y_{\mathcal{TR}}$ when context switches occur. The SCFVD solution verifies each indirect control-flow transfer must match an edge in the $G_{\mathcal{S}}$ or the destination of each asynchronous control-flow transfer must match an address in the $I_{\mathcal{K}}$ or $Y_{\mathcal{T}}$:

System-oriented CFVD (SCFVD). Given the trace $R_{\mathcal{S}} = (r_0, r_1, \ldots, r_n)$ of a system $\mathcal{S}$ including a kernel $\mathcal{K}$ and tasks $\mathcal{T}$, SCFVD verifies that $r_i \in E_{\mathcal{S}} \bigvee r_i.d \in I_{\mathcal{K}} \bigcup Y_{\mathcal{T}}, \forall i \in \{0, 1, \ldots, n\}$.

Another challenge an SCFVD solution faces is how to securely trace the protected system without allowing the privileged but potentially compromised system to disable or disrupt the tracing. Additionally, it is important to ensure the traces and the analysis of the traces are secured from the protected system. To this end, an SCFVD solution needs to prevent the privileged system from configuring the hardware tracing unit, secure the trace in a protected memory region, and perform the analysis in a higher privileged or isolated mode, e.g., hypervisor or trusted execution environment.

3 BACKGROUND: ARMv8-M AND FreeRTOS

In this section, we discuss the ARMv8-M microcontroller architecture and FreeRTOS on which we implemented and evaluated the prototype of SHERLOC on.

3.1 ARMv8-M Architecture

ARMv8-M is a 32-bit architecture that features 16 general-purpose registers: `R0` to `R12`, `R13/SP` (stack pointer), `R14/LR` (link register), and `R15/PC` (program counter). The `LR` holds the return address for a subroutine or a special value `EXC_RETURN` (`0xFFFFFF**`) when an interrupt occurs. There are two execution modes in ARMv8-M: thread and handler mode, with two privilege levels: privileged and unprivileged. The handler mode is always privileged, while thread mode can be either privileged or unprivileged. In thread mode, privilege escalation is done through the SuperVisor Call (`SVC`) instruction. ARMv8-M uses a 32-bit physical address space, which consists of several areas, including code (flash), SRAM, peripheral, and system. The peripheral area contains memory-mapped peripheral control registers, while the system area contains system control units, such as the memory protection unit and the SysTick timer. ARMv8-M adopts automatic stacking and unstacking for interrupt and exception handling. When a higher priority interrupt or exception occurs, the processor automatically saves general registers to the current stack and stores the special value `EXC_RETURN` to `LR`. The processor then executes the interrupt service routine, and upon completion, automatically restores the saved context.

3.2 ARMv8-M Hardware Tracing Unit

ARMv8-M architecture features a hardware tracing unit called Micro Trace Buffer (MTB). MTB captures *all* non-sequential program counter changes on the microcontroller, including calls, branches, and exceptions, and stores trace records, i.e., source and destination address pair of the non-sequential PC change, at the trace buffer within the SRAM area in a circular arrangement manner. The address and size of trace buffer for each hardware system may be different but fixed, and their values can be retrieved from the corresponding registers, e.g., MTB_BASE. Table 1 presents the important MTB registers and fields used by SHERLOC.

Table 1: ARMv8-M MTB registers

Register	Used fields	Description
MTB_MASTER	EN, HALTREQ	Enable MTB
MTB_POSITION	POINTER	Write pointer offset
MTB_FLOW	WATERMARK, AUTOHALT	Stop conditions
MTB_BASE	BASE	Address of the trace buffer

Table 2 summarizes non-sequential control-flow transfer cases in ARMv8-M. Each trace record, denoted as $r = \langle s, d \rangle$, is 8 bytes and contains the source and destination addresses, where the least significant bit (A-bit) of the source indicates whether the transfer originated from an instruction or an exception. Most non-sequential control-flow transfers generate one record, except for ISR returns (e.g., BX LR when LR = EXC_RETURN), which generate two records. In the first record, s is the address of the return instruction, and d is EXC_RETURN. In the second packet, s is EXC_RETURN, and d is the control-flow transfer destination. When a predefined watermark is reached, MTB can trigger a Debug Monitor (DebugMon) exception.

3.3 ARMv8-M Data Watchpoint and Trace Unit

ARMv8-M architecture provides a Data Watchpoint and Trace unit (DWT), which comes with special registers called comparators. These comparators are used for code/data address matching and CPU cycle counting. The comparators can monitor read and/or write operations to specified addresses and trigger a DebugMon exception when there is a match. Each comparator has a DWT_COMP register that specifies the address being monitored and a DWT_FUNCTION register that defines the operation to be monitored, such as read and/or write. Since peripherals are also memory-mapped on ARMv8-M, this mechanism can monitor any system behavior, from memory reads and writes to executing code at any address and modifying system registers.

3.4 ARMv8-M Trusted Execution Environment

In addition, ARMv8-M architecture features a trusted execution environment called TrustZone. TrustZone adds an orthogonal partitioning of states to execution modes and privilege levels, providing an isolated execution environment for secure software. With TrustZone, the processor has the secure and non-secure states, and the division of secure and non-secure is memory map-based, where a memory region can be secure, non-secure callable (NSC), or non-secure, determined by the combination of secure attribution unit (SAU) and vendor-specific implementation-defined attribution unit (IDAU) configurations. SAU is configurable, whereas IDAU configurations are usually fixed. An NSC region serves as a springboard from non-secure regions to secure regions using the Secure Gateway (SG) instruction. Different from ARMv8-A TrustZone that sets the NS bit in the Secure Configuration Register (SCR) to indicate the security state of the processor, the division of secure and non-secure in ARMv8-M is based on the memory map, and state transitions take place automatically.

3.5 FreeRTOS Overview

FreeRTOS is a real-time operating system (RTOS) capable of multitasking and can run on the ARMv8-M architecture. It uses a priority-based preemptive scheduler. The scheduler is implemented in the SysTick timer ISR, i.e., SysTick_Handler(), which checks whether the current task has used up its scheduling quantum. If so, the SysTick_Handler() sets the PENDSVSET bit of the Interrupt Control and State Register (ICSR), which triggers a Pending Supervisor Call (PendSV) exception. The PendSV ISR, i.e., PendSV_Handler(), then performs the context switch by identifying the runnable task with the highest priority and resuming its execution. Unlike other interrupts and exceptions that return to the interrupted task or ISR, the PendSV exception returns to a different task. Additionally, a privileged task can use the taskYIELD() function to request a context switch to another task with higher or equal priority to the current task, which also sets the PENDSVSET bit of the ICSR.

4 SHERLOC

As Figure 2 shows, SHERLOC comprises offline analysis and runtime configuration and enforcement modules. In this section, we first describe the system and threat model. Then, we illustrate how each module of SHERLOC works.

4.1 System and Threat Model

We assume that the microcontroller has a hardware tracing unit that is capable of generating non-sequential control-flow transfer address pairs. We do not assume that the tracing unit has advanced filtering capabilities, such as selective tracing of a particular task. Such features are typically not available for microcontroller tracing unit like MTB. Additionally, we assume the availability of a trusted execution environment, such as TrustZone, which provides secure isolation between the rich execution environment (REE; e.g., non-secure state in TrustZone) and the trusted execution environment (e.g., secure state in TrustZone) where the runtime modules of SHERLOC execute. We assume a secure boot process that ensures the integrity of SHERLOC's code and data, e.g., indirect branch table, at boot-time, preventing any tampering while SHERLOC components are at rest. We assume that TrustZone is secure and that SHERLOC is trusted at run-time. The protected system may be a bare-metal system or an RTOS with multiple tasks. The RTOS may or may not enable privilege separation. Both privileged and unprivileged modules in the rich execution environment may have bugs, and their memory may be corrupted to perform control-flow hijacking attacks. As with all other CFI solutions, we assume the code integrity of the protected system so that direct calls and branches cannot be tampered with. This can be ensured by configuring the code segment of the protected system as read-only or non-writable, to

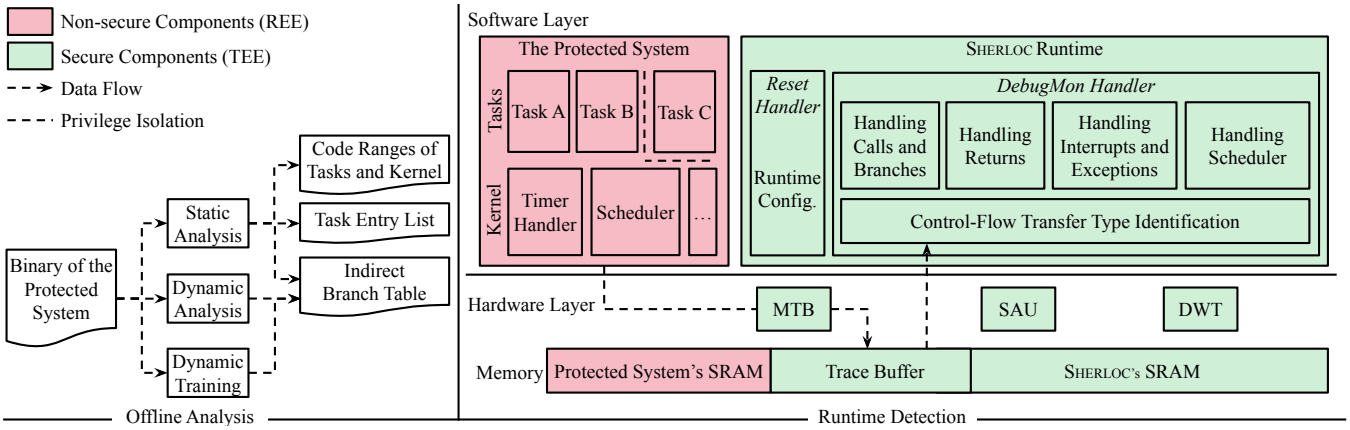


Figure 2: SHERLOC comprises offline analysis and runtime configuration and enforcement modules. The unmodified protected system program runs in the non-secure state, whereas SHERLOC runtime modules execute in the secure state.

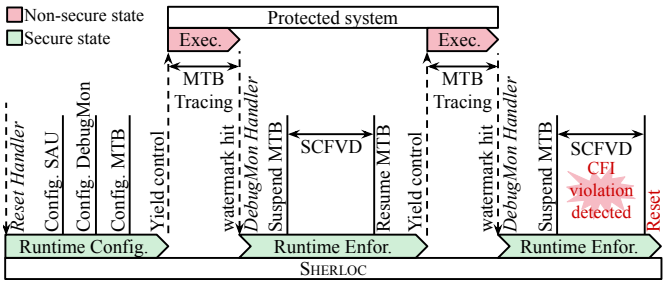


Figure 3: Timeline showing the steps of SHERLOC and protected system on a single core microcontroller. If a CFI violation is detected, SHERLOC resets the system.

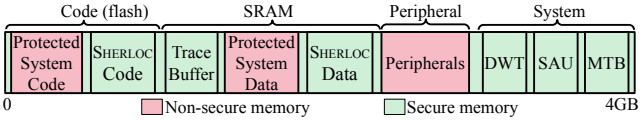


Figure 4: An example memory layout of the entire system

prevent modification of the code by an attacker. We do not consider hardware attacks, such as glitching attacks.

4.2 Offline Analysis

To validate an indirect or asynchronous control-flow transfer, SHERLOC requires an over-approximated CFG, a list of asynchronous ISR addresses, and entry and re-entry addresses of all RTOS tasks. An over-approximated CFG may result in higher false negative rate, but it guarantees no false positives. The offline analysis module only produces the set of legitimate indirect forward edges from the generated CFG and entry addresses of all RTOS tasks. SHERLOC runtime modules can retrieve asynchronous ISR addresses directly from the non-secure state vector table (VT) on SRAM or flash by reading the vector table offset register (`VTOR_NS`). The indirect backward edges, i.e., returns, and the re-entry addresses of tasks are maintained dynamically by the SHERLOC runtime enforcement module.

To obtain an over-approximated CFG, we leverage existing binary analysis tools, such as angr [43], that disassemble and analyze binaries. Such tools employ forced execution for adding basic blocks, backward slicing to identify contexts, and symbolic back-traversal to resolve indirect calls and branches. Prior research has demonstrated that the outcomes are sufficiently precise in real-world scenarios [21, 47, 49, 54, 55, 58]. Additionally, we use dynamic training in a closed environment with benign inputs to capture control-flow transfers that might be missed by the aforementioned tools. Afterwards, the offline analysis module merges the CFG generated from static/dynamic analysis and dynamic training, and reduces the CFG to an indirect branch table (IBT) that contains the source and destination address pairs of all indirect calls and branches. IBT provides higher precision compared to target set-based approaches [51], as it maintains a destination address set for each source address.

As we do not assume advanced task filtering features of the hardware tracing unit, identifying entry addresses of the running tasks by examining the trace at run-time is difficult. To overcome this challenge and facilitate runtime detection, the offline analysis module uses RTOS heuristics to identify the entry addresses of tasks. For instance, on FreeRTOS, the entry addresses have a `TaskFunction_t` type, and tasks are created using the `xTaskCreate()` function or its variants. These functions call the `prvInitialiseNewTask()` function and pass the task entry address as the first parameter. Based on these heuristics, the offline analysis module uses binary analysis techniques to identify the entry addresses of all tasks, which are kept in a task entry list ($Y_{\mathcal{TE}}$).

In addition, the offline analysis module also identifies the code range of various modules for run-time use, such as the start and end addresses of each task's code, shared libraries, and kernel code.

4.3 Configuration for Holistic Enforcement

When a TrustZone-enabled microcontroller boots, it starts in the secure state, and the execution begins with the secure state reset handler as shown in Figure 3. The SHERLOC runtime configuration module is implemented within this reset handler and has two primary functions. First, the module configures SAU to partition

Table 2: The approach that SHERLOC takes for handling each type of dereferenced instruction in the trace. It should be noted that SHERLOC disregards the small number of transfers that occur across states, as well as the transfers within the secure state. $\langle s, d \rangle$: a standard trace record. $(\langle s_1, d_1 \rangle, \langle s_2, d_2 \rangle)$: a pair of interrupt or exception return trace records. IBT: Indirect branch table. VT: non-secure state vector table. RCS: the current task- or kernel-specific reconstructed call stack. $Y_{\mathcal{T}}$: task entry and re-entry address list. `Rx`: any general purpose register.

	Type	Instruction(s)	Ins. Size	How to Identify the Type?	SHERLOC Actions
Bare-metal System and RTOS Cases	Direct branch (§4.4.1)	`B{cond} #imm`	2/4	The dereferenced instruction	Skip the record
	Direct call (§4.4.1)	`BL{cond} #imm`	4	The dereferenced instruction	RCS.push(s + 4)
	Indirect branch (§4.4.1)	`BX{cond} Rx` `TBB/TBH {PC, ...}`	2 4	The dereferenced instruction	*if* $\langle s, d \rangle \notin$ IBT, reset
	Indirect call (§4.4.1)	`BLX Rx`	2	The dereferenced instruction	*if* $\langle s, d \rangle \notin$ IBT, reset; *else* RCS.push(s + 2)

	Function return (§4.4.2)	`BX LR`
		`POP {..., PC}`
		`LDM SP!, {..., PC}`
			2/4	The dereferenced instruction	*if* $d \neq$ RCS.pop(), reset

	Sync. exception (§4.4.3)	`SVC #imm`	2	s[A-bit]	*if* $d \notin$ VT, reset; *else if* $d \neq$ PendSV, RCS.push(s)
	Non-PendSV async. interrupt (§4.4.3)	N/A	N/A	s[A-bit]	*if* $d \notin$ VT, reset; *else if* $d \neq$ PendSV, RCS.push(s)

	Non-PendSV ISR return (§4.4.4)	`BX LR`
		`POP {..., PC}`
		`LDM SP!, {..., PC}`
			2/4	The dereferenced instruction and (d_1 == `EXC_RETURN` $\wedge$ s_2 == `EXC_RETURN`)	*if* bare-metal and $d_2 \neq$ RCS.top(), reset; *else if* bare-metal and d_2 == RCS.top(), RCS.pop(); *else* go to PendSV ISR return handling

RTOS-only Cases	PendSV async. interrupt (§4.4.5)	N/A	N/A	s[A-bit]	*if* d == PendSV, $Y_{\mathcal{T}}$.add(s) and $Y_{\mathcal{T}}$.add(RCS.pop())

	PendSV ISR return (§4.4.6)	`BX LR`
		`POP {..., PC}`
		`LDM SP!, {..., PC}`
			2/4	The dereferenced instruction and (d_1 == `EXC_RETURN` $\wedge$ s_2 == `EXC_RETURN`)	*if* $d_2 \notin Y_{\mathcal{T}}$, reset; *if* d_2 is in a shared library, and assuming the next trace record is $\langle s_n, d_n \rangle$, and $d_n \notin Y_{\mathcal{T}}$, reset

the secure and non-secure memory and ensure that the protected system runs in the non-secure state while SHERLOC runs in the secure state. Figure 4 presents an example memory map of the entire system. In addition to the code and data sections of the protected system, the peripheral area is configured as non-secure to allow the protected system's direct access to peripherals. However, the system area, in which SAU and MTB reside, is set as secure to prevent a compromised non-secure system from reconfiguring the memory layout or disabling the tracing. The trace buffer is also set as secure to restrict access to only SHERLOC.

Next, SHERLOC sets a watermark for the trace buffer by configuring the `MTB_FLOW` register. To enable the watermark mechanism and capture a watermark hit event with the DebugMon exception, SHERLOC also needs to set the debug monitor enable bit (`MON_EN`) in the Debug Exception and Monitor Control Register (`DEMCR`). Setting a higher watermark reduces the frequency of entering into the SHERLOC runtime enforcement module, thereby resulting in higher run-time performance. On the other hand, setting a lower watermark decreases the detection latency. With this configuration, a DebugMon exception will be raised when the watermark is hit, and the microcontroller enters the secure state to execute the DebugMon ISR. The SHERLOC runtime enforcement module, which we will discuss in the next subsection, is implemented in the DebugMon ISR. Even if the DebugMon ISR can immediately suspend tracing, several non-sequential control-flow transfer records generated by housekeeping instructions will be saved in the trace buffer. For instance, the state switch from non-secure to secure state will include several non-sequential control-flow transfers. To prevent these records from overwriting the protected system's trace, the watermark value should be smaller than the trace buffer size. We have empirically determined that a 32-byte slack (i.e., 4 non-sequential transfers) is safe for this purpose.

Afterwards, SHERLOC sets the `EN` bit in `MTB_MASTER` register to enable tracing and yields the control to the protected system in the non-secure state by calling its reset handler.

4.4 Holistic Runtime Enforcement

Once the trace buffer reaches the watermark, MTB automatically sets the `HALTREQ` bit of the `MTB_MASTER` register, which causes a DebugMon exception. Subsequently, the microcontroller executes the secure state DebugMon ISR, in which SHERLOC suspends the MTB tracing by clearing the `EN` bit in `MTB_MASTER`. Then, SHERLOC examines each record in the trace to detect any control-flow violations. To determine whether a transfer originated from an interrupt or an exception, SHERLOC first checks the A-bit of the source address (as discussed in §3.2). It then dereferences the instruction in the non-secure state and processes accordingly based on the type. Table 2 summarizes how SHERLOC handles each type of dereferenced instruction. It should be noted that the trace will include a small number of records from the transfers that occur across states (e.g., interrupting non-secure execution and entering DebugMon ISR),

as well as the transfers within the secure state (e.g., in DebugMon ISR). SHERLOC ignores such records as they are the results of secure transfers and thus require no further action.

4.4.1 Handling Calls and Branches. As direct branches are considered secure, SHERLOC skips them. For indirect branches and calls, SHERLOC verifies whether the source and destination address pair is in the IBT. If the pair is not in the IBT, a CFI violation is detected. Additionally, for direct and indirect calls, SHERLOC reconstructs call stacks, namely reconstructed call stacks (RCS), from the trace, which are used for the violation detection of function returns. For bare-metal systems, one RCS is enough, whereas RCSs are task- and kernel-specific for RTOSs. An RCS comprises return addresses, which are the source address plus 4 for direct calls and source address plus 2 for indirect calls. To identify which task a trace record belongs to, SHERLOC compares the source address with the module address ranges identified in offline analysis (§4.2) and pushes return addresses onto the corresponding RCS.

4.4.2 Handling Function Returns. By maintaining an RCS for bare-metal systems and RCSs for RTOSs-based systems, SHERLOC provides the same level of precision as shadow stacks for backward edges [20, 29, 38, 57]. For example, consider task C in Figure 1, where either c_2 or c_3 is a legitimate destination of the indirect callsite c_1. Without RCS, the consecutive record sequence $(\langle c_1, c_3\rangle, \langle c_4, c_5\rangle)$ would be considered legitimate. However, RCS increases the precision to only allow either the record sequence of $(\langle c_1, c_2\rangle, \langle c_4, c_5\rangle)$ or $(\langle c_1, c_3\rangle, \langle c_6, c_5\rangle)$.

4.4.3 Handling Non-PendSV Interrupts and Exceptions. Both bare-metal systems and RTOSs use non-PendSV interrupts and exceptions, and they can occur at any time. SHERLOC handles non-PendSV interrupts and exceptions differently from the PendSV interrupts (which will be discussed in §4.4.5). If a trace record originates from a non-PendSV interrupt or exception (i.e., the A-bit in the source address is set), SHERLOC first compares the destination address d in the record with the addresses in the non-secure state vector table. If no match is found, a CFI violation is detected. If a match is found, SHERLOC considers that the interrupt or exception is legitimate. To ensure the return to the interrupted code address from an interrupt or exception, SHERLOC pushes the interrupted code address s onto the current task's RCS. If the interrupt or exception occurs before any task is scheduled, the interrupted address is pushed onto the kernel RCS.

4.4.4 Handling Non-PendSV Interrupt and Exception Returns. Handling non-PendSV interrupt and exception returns is similar to handling regular function returns, with the difference that MTB generates two trace records $(\langle s_1, d_1\rangle, \langle s_2, d_2\rangle)$ for interrupt returns. SHERLOC verifies whether the destination address of the first record d_1 and the source address of the second record s_2 both equal to `EXC_RETURN`. It then peeks the top of the current RCS and checks whether the destination address of the second record d_2 matches the top of the current RCS. If there is a match, it indicates a legitimate non-PendSV return, and SHERLOC pops the top value from the current RCS. Otherwise, for a bare-metal system where there is only one RCS, it means a CFI violation is detected. For an RTOS, SHERLOC further verifies whether the record is a legitimate PendSV interrupt return, as will be described in §4.4.6.

4.4.5 Handling PendSV Interrupts. As discussed in §3.5, while bare-metal systems do not use PendSV interrupts, RTOSs adopt PendSV ISR to perform context switches, which can be triggered by either preemptive scheduling (i.e., a SysTick event) or a task yielding control (i.e., a task calling the `taskYIELD()` function in the shared FreeRTOS library). Therefore, a PendSV interrupt could occur at any code address of a task or a shared library. If the interrupted address is in a shared library, when PendSV ISR returns, it is critical to determine which task- or kernel-specific RCS to use for validation. To facilitate the identification of the appropriate RCS, in addition to the interrupted code address s, SHERLOC adds the current RCS's top item to the entry and re-entry list $Y_{\mathcal{T}}$.

4.4.6 Handling PendSV and Scheduler Returns. For a PendSV interrupt return record, SHERLOC checks whether d_2 is in the list of entry and re-entry task addresses $Y_{\mathcal{T}}$. If not, a CFI violation is detected. If d_2 is in $Y_{\mathcal{T}}$, SHERLOC checks whether d_2 belongs to a task or the kernel code, by comparing it to their address ranges and determines which RCS to use next. However, if d_2 is in the range of a shared library, SHERLOC needs to figure out which task or kernel called that library. As discussed in §4.4.5, the triggering of PendSV interrupts in FreeRTOS can occur either inside a SysTick handler or the `taskYIELD()` function in the shared FreeRTOS library. In the SysTick case, when a PendSV ISR returns, it returns to the interrupted code address in the task directly, and SHERLOC can easily identify the corresponding RCS to use next. In the task yielding control case, the PendSV ISR returns to the `taskYIELD()` function first, which in turn returns to the interrupted address in a task immediately with a regular function return. Since `taskYIELD()` returns immediately, and it does not introduce any extra non-sequential control-flow transfers except the function return. However, for the trace record $\langle s_n, d_n\rangle$ that originated from the `taskYIELD()` returns, SHERLOC does not know which task RCS it should check against. Therefore, it checks whether the return address d_n is in $Y_{\mathcal{T}}$, which should have been added there when a legitimate PendSV interrupt record was handled before, as discussed in §4.4.5.

4.4.7 Resume Tracing. If no violation is found after analyzing all of the trace records, SHERLOC resumes MTB tracing by setting the `EN` bit and clearing the `HALTREQ` bit in the `MTB_MASTER` register and resetting the `POINTER` field in the `MTB_POSITION` register to 0.

4.5 Event-triggered Runtime Enforcement

Besides holistic runtime enforcement, SHERLOC also supports event-triggered runtime enforcement, providing a trade-off between security and performance. Similar to the existing implementations of application-oriented CFVD [23, 34, 47, 51, 53], the enforcement can be triggered by some sensitive events, such as calling a certain kernel API (similar to system calls on desktop) and modifying a particular system register (e.g., changing memory permissions). To this end, SHERLOC utilizes DWT and does not rely on the MTB watermark mechanism. SHERLOC configures the `DWT_COMP` registers to monitor code or data addresses and the `DWT_FUNCTION` register to trigger a DebugMon exception upon operations at the monitored addresses. SHERLOC enables the DWT comparators by setting the `TRCENA` bit of the `DEMCR` register. When a monitored event occurs, DWT automatically sets the `MATCHED` field in the `DWT_FUNCTION`

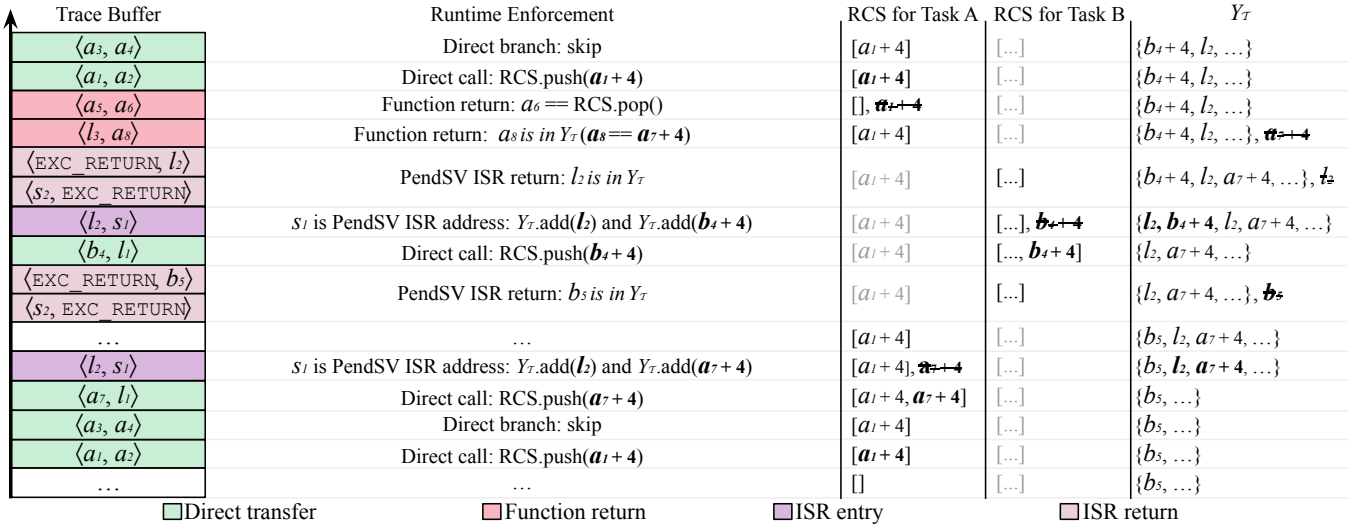


Figure 5: A running example showing how SHERLOC verifies the trace from FreeRTOS with two tasks. Both tasks call `taskYIELD()` in the shared library. At the beginning of this trace, task B has been suspended and task A is about to execute. [] represents RCS with the top on the right-hand side. Black [] represents the active RCS, and gray [] represents an inactive RCS.

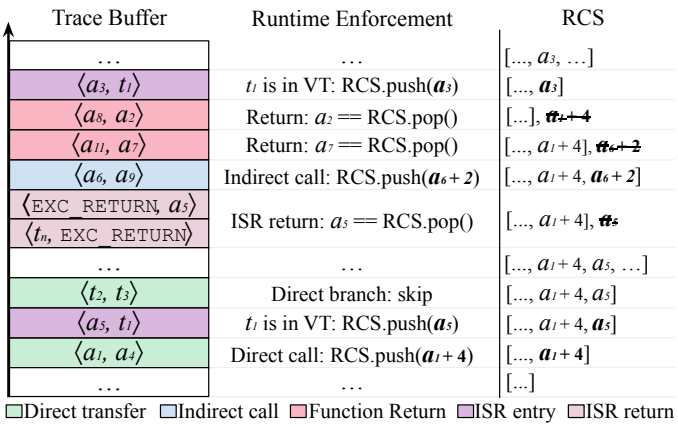


Figure 6: A running example showing how SHERLOC verifies the trace from a bare-metal system. [] represents RCS with the top on the right-hand side.

register and triggers the DebugMon exception. Upon entering the DebugMon ISR, SHERLOC suspends the MTB tracing, clears the `MATCHED` field of `DWT_FUNCTION`, and examines the trace records.

Since the MTB overwrites the trace buffer in a circular manner, in the event-triggered mechanism, SHERLOC only has access to the most recent records, not the full trace. The number of available records depends on the size of the trace buffer, so this mechanism cannot provide the same level of precision as the holistic mechanism. Specifically, the event-triggered mechanism only handles indirect function calls and returns, not interrupts and exceptions. The IBT for this mechanism includes all possible indirect control-flow transfers, including function returns.

4.6 Running Examples for Holistic Runtime Enforcement

4.6.1 A Bare-metal System Example. Figure 6 illustrates how SHERLOC handles different trace records for a bare-metal system. Since the bare-metal system only adopts one call stack, SHERLOC also only maintains one RCS. After identifying the instruction that a record originated from, SHERLOC performs a corresponding action, including updating the RCS accordingly. The example trace starts with a direct call at a_1, so the address of the next instruction $a_1 + 4$ is pushed onto the RCS. SHERLOC verifies that the instruction at a_5 was legally interrupted and pushes the interrupted code address a_5 onto the RCS. When seeing an interrupt return record, SHERLOC compares the destination a_5 with the top item of the RCS. Indirect calls are handled similarly, with the address of the next instruction being pushed onto the RCS.

4.6.2 An FreeRTOS Example. Figure 5 illustrates how SHERLOC handles the trace from FreeRTOS with two running tasks. Both tasks call `taskYIELD()` in the shared library to yield control. At the beginning of this trace, task B has been suspended, so a re-entry address b_5 of task B is already in the task entry and re-entry list $Y_{\mathcal{T}}$. As task A is about to execute, SHERLOC uses task A's RCS as the active RCS. After executing for a while, task A calls `taskYIELD()` at l_1 to yield control. `taskYIELD()` triggers the PendSV interrupt, which generates the record for the PendSV ISR. SHERLOC adds both the interrupted address l_2 of `taskYield()` and the top item $a_7 + 4$ of the active RCS to $Y_{\mathcal{T}}$. Afterwards, the records show that task B executes, so SHERLOC switches the active RCS to task B's RCS. Later, task B yields control. When seeing a PendSV ISR return record, SHERLOC verifies whether l_2 and the destination part of the next record a_8 are both in $Y_{\mathcal{T}}$.

5 IMPLEMENTATION AND EVALUATION

In this section, we first discuss the implementation of SHERLOC for the ARMv8-M architecture. Then, we evaluate the security and effectiveness of SHERLOC, including an analysis of its detection latency. We present the performance evaluation results of SHERLOC on BEEBS benchmark suite, bare-metal systems, and FreeRTOS. Finally, we compare SHERLOC with prior approaches.

5.1 Implementation

We developed a prototype of SHERLOC for the ARMv8-M architecture. The offline analysis module is composed of 948 lines of Python code, utilizing angr [43] and capstone [1] for binary disassembly and CFG analysis. For the runtime modules, we added them to the Reset and DebugMon handler of secure state code in the Keil IDE TrustZone example project [2], which includes system initialization code. The runtime configuration module includes 425 lines of C code, with 34 lines for the event-trigger installation. The runtime enforcement module is made up of 1,020 lines of C code. SHERLOC reserves 1 KB for each task's RCS and 3 KB for the task entry and re-entry list ($Y_{\mathcal{T}}$). We compiled the SHERLOC runtime modules with armclang [8] O3 optimization, and the generated secure state image is 13.1 KB in .text, 4 KB in .data, and 16.45 KB in .bss sections.

5.2 Evaluation Environment

We evaluated SHERLOC on the ARM Versatile Express Cortex-M prototyping system (V2M-MPS2+) [10], which includes peripherals, such as Ethernet, UART, LCD touch screen, LED, etc. We configured this system as a Cortex-M33 microcontroller running at 20MHz using the AN505 FPGA image [7]. The configured system has a 4 MB code/flash region and 4 MB SRAM, including 4 KB allocated to the MTB trace buffer. In addition, it supports 8 configurable SAU regions and 4 DWT comparators.

5.3 Security Analysis and Evaluation

5.3.1 Detection Latency Analysis. Like all other CFVD mechanisms, SHERLOC introduces detection latency, which can be measured by the number of instructions the attacker executes between the first hijacked control-flow transfer and its detection. Assuming that the watermark is set at the N-th byte, this is constrained by the size of the trace buffer and the records originated from SHERLOC itself, which consumes a small and bounded bytes in the trace (e.g., 32 bytes). A pair of interrupt or exception return records consumes 16 bytes, and all other non-sequential control-flow transfer records consume 8 bytes. Therefore, there will be at most $\frac{N}{8}$ records in the trace buffer. Assuming that the control-flow hijacking attack occurs at the i-th record (following a uniform distribution between the first and $\frac{N}{8}$-th record), then the delay is at most $\frac{N}{8} - i$ non-sequential transfers, considering that asynchronous interrupts will also generate trace records. Therefore, the average delay is at most $\frac{N}{16}$ non-sequential transfers. If we denote the average number of instructions before a non-sequential control-flow transfer instruction as S, the expected average detection latency of SHERLOC can be approximated as $\frac{N \times S}{16}$ instructions.

[2] https://www.keil.com/dd2/arm/iotkit_cm33/

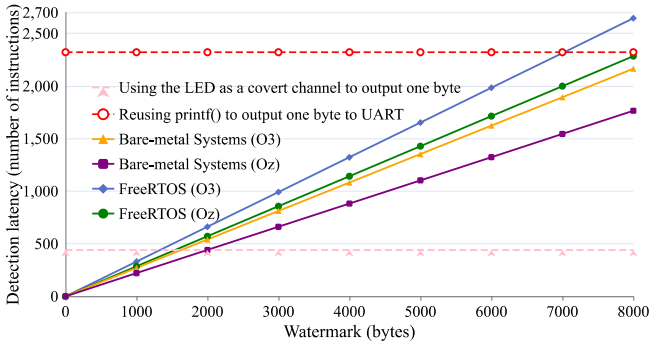


Figure 7: The expected detection latency with different watermark settings and average number of instructions before a non-sequential transfer. Most ARMv8-M systems only have a trace buffer of 4KB or smaller.

We conducted an empirical measurement of S in FreeRTOS and bare-metal systems. The results show that for armclang O3 optimization, FreeRTOS has an S value of 5.29, and for Oz optimization, it has an S value of 4.57. On the other hand, for O3 optimization, bare-metal systems have an S value of 4.33, and for Oz optimization, they have an S value of 3.53. Figure 7 shows the expected detection latency with different N and S.

To determine whether an attacker can successfully execute meaningful attacks before SHERLOC detects them, we consider a scenario where the attacker attempts to exfiltrate sensitive data from the system by conducting a ROP attack to read the data and reuse functions for exfiltration, such as (i) using the C library function `printf()` to output to stdout or using `sendto()` to send a UDP packet, and (ii) toggling the GPIO-based LED as a covert channel to output data. In our experiments, we make the assumption that the attacker has already gained permissions to access peripherals such as UART and GPIO. However, it is important to note that if the attacker does not possess the required privileges, e.g., control-flow hijacked an unprivileged task on FreeRTOS, they will need to execute a significantly larger number of instructions to escalate their privilege level first. This additional step of privilege escalation further increases the chances for SHERLOC to detect attacks.

Our experimental results show that it takes more than 2,300 instructions to reuse `printf()` to print out just one byte to the UART interface. It is worth noting that while `printf()` has a simple implementation and the UART interface has an uncomplicated driver, more complex exfiltration methods, such as `sendto()`, and drivers, such as those used for WiFi, Ethernet or Bluetooth, will require significantly more instructions to complete a similar task. In the second experiment, we leverage a GPIO toggling operation to alternate the LED status, thereby conveying one bit of information. This operation requires 55 instructions (including checking current GPIO status and reusing library function to turn on/off LED, but excluding any delays that the attacker may need to introduce between toggles). Thus, leaking a byte of data necessitates at least 440 instructions. As illustrated in Figure 7, the results indicate that the detection latency of SHERLOC is not a significant issue.

Table 3: The evaluated programs/systems, i.e., BEEBS programs, Blinky bare-metal system, and FreeRTOS, along with the number of their indirect call (IC), indirect branch (IB), direct call (DC), direct branch (DB), return (RET) instructions, and IBT entries (IBT). The two numbers in each entry of the table represent the number of instructions for O3/Oz optimization, respectively. It is important to note that the number of instructions does not necessarily correspond to the number of times those instructions are executed at run-time. For the BEEBS programs, we only measure the performance overhead of the programs alone. For Blinky and FreeRTOS, we measure the overhead of the entire systems, including system initialization and the handling of non-PendSV and PendSV interrupts and exceptions.

	bubblesort	crc32	dijkstra	edn	fasta	frac	levenshtein	nbody	ndes	nettle_aes	picojpeg	qrduino	rijndael	arraybinsearch	dllist	hashtable	listsort	queue	rbtree	st	whetstone	Blinky	FreeRTOS
IC	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/3	0/0	0/15	0/14	0/0	0/0	0/2	0/1	0/0	0/0	1/1	0/0	0/6	0/110	8/3
IB	0/0	0/0	0/0	0/0	0/0	1/1	0/0	1/1	0/0	0/0	5/5	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	1/2	1/2	5/5
DC	10/0	5/0	10/2	7/0	5/4	71/45	32/3	121/18	21/4	7/4	88/60	44/50	10/19	5/0	5/1	6/7	5/0	6/0	13/9	302/33	387/297	70/59	103/112
DB	33/12	7/7	49/24	20/29	32/9	113/110	52/26	93/94	17/32	30/25	532/206	456/235	90/60	23/12	82/3	154/28	55/29	45/20	99/90	121/35	226/236	2040/67	389/239
RET	3/0	1/0	2/1	3/0	1/2	68/66	6/5	40/36	2/1	2/2	17/19	17/25	7/10	1/0	1/1	3/3	2/0	1/0	7/7	69/44	107/105	19/25	30/41
IBT	0/0	0/0	0/0	0/0	0/0	1/1	0/0	1/1	0/3	0/0	8/24	9/22	0/0	0/0	0/8	0/4	0/0	0/0	2/2	1/1	1/7	4/121	11/8

However, the attacker can still successfully toggle a bit by utilizing ROP directly into a library function. In this scenario, the protected system will rapidly encounter a hardware fault due to the imbalanced stack. The default handler for this hardware fault is an infinite loop, causing it to fill up the MTB buffer and eventually trigger SHERLOC's holistic enforcement. As a result, this attack will eventually be detected but not prevented. To address this concern, SHERLOC's event-triggered enforcement can effectively prevent such attacks by setting the DWT to monitor the write behaviors of the memory-mapped register.

5.3.2 Effectiveness Evaluation. To evaluate the effectiveness of SHERLOC, we conducted two control-flow hijacking attacks on FreeRTOS and checked whether SHERLOC could detect them. The source code and detection traces for these attacks are available in our anonymized code repository. In the first attack scenario, a task running on FreeRTOS has a stack-based buffer overflow vulnerability that can be exploited to hijack the backward-edge control-flow transfer. In this case, SHERLOC discovered that the return address did not match the top item on the task's RCS. In the second attack scenario, we assumed and exploited an arbitrary write bug in the FreeRTOS kernel that allowed attackers to modify any task's re-entry address in the saved contexts. SHERLOC was able to detect that the tampered re-entry address was not present in the $Y_{\mathcal{T}}$ set and successfully stopped the attack.

5.4 Performance Evaluation

5.4.1 Experiment Setup. In the performance evaluation, all of the protected programs/systems were compiled with armclang using O3 and Oz compile-time and link-time optimizations. In general, the binaries generated from O3 optimization run faster and have fewer indirect branches/calls but larger binary sizes, while binaries from Oz optimization run slower and have more indirect branches/calls but reduced binary sizes. Table 3 presents all the evaluated programs/systems along with the number of their indirect call, indirect branch, direct call, and return (i.e., the three instructions shown in the row of *function return* in Table 2) instructions in the binaries.

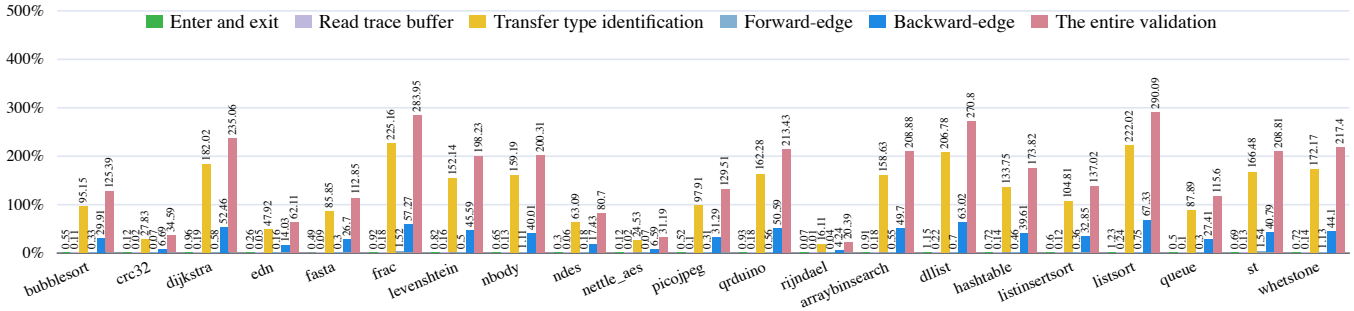
To evaluate the holistic enforcement performance of SHERLOC, we configured the watermark at 4032 bytes and conducted experiments on the following three sets of programs/systems.

We utilized the BEEBS benchmark (commit number `049ded9` [39] to evaluate SHERLOC's performance in handling calls, branches, and function returns. As discussed in Silhouette [57], some BEEBS programs are very small and would be excessively optimized during compile-time. Therefore, we selected 22 programs with relatively complex execution times. Each chosen BEEBS program was incorporated into the main function of the non-secure state code in the Keil IDE TrustZone example project. We solely measured the CPU cycles spent by these applications with or without SHERLOC enabled, excluding the CPU cycles of the system initialization.

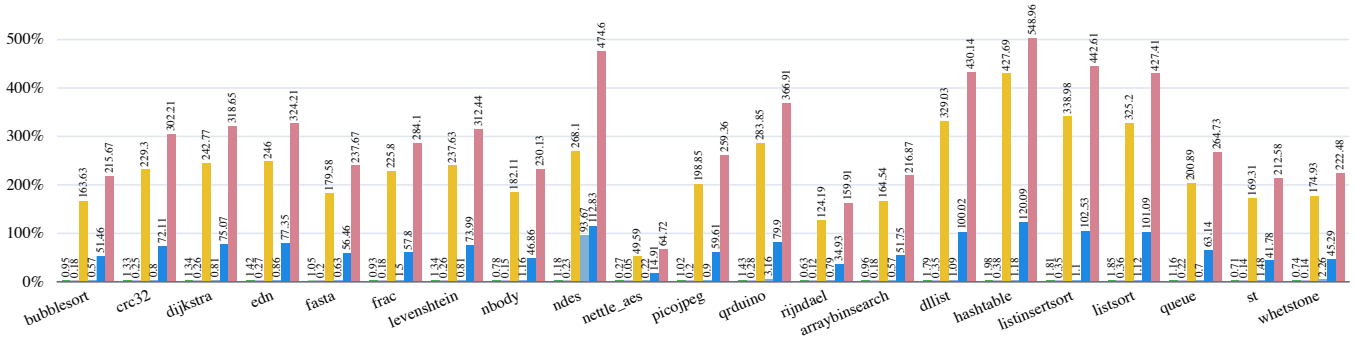
We employed the Blinky bare-metal system [9] that displays animations on the LCD touchscreen and toggles a LED at a predetermined time interval, such as 10 ms or 100 ms. As the Blinky application includes the drivers for those peripherals, it has the largest binary size among all the evaluated programs/systems, with 126,592 bytes for O3 and 49,324 bytes for Oz. The Blinky application sets up the SysTick timer and handles the SysTick interrupt periodically. This experiment examines SHERLOC's performance to handle non-PendSV interrupts, exceptions, and returns across various timer interrupt frequencies. We measured the CPU cycles from non-secure state system boots to Blinky finishing 10 toggles.

We used a FreeRTOS setup with memory protection unit (MPU) enabled (v202112) to evaluate SHERLOC's performance for handling PendSV interrupts and scheduler returns. This FreeRTOS setup only implements the ISRs for reset, hardware fault, SVC, SysTick timer, and PendSV interrupts and exceptions, and it excludes drivers for most peripherals. In these experiments, FreeRTOS runs two tasks: one is an unprivileged crc32 task from the BEEBS benchmark, and the other is a privileged idle loop task. The generated FreeRTOS binary is 74,452 bytes for O3 optimization and 65,596 bytes for Oz optimization. We measured the CPU cycles from non-secure state system boots to the crc32 task finishing five executions.

To evaluate the event-triggered enforcement performance of SHERLOC, we carried out experiments with two different triggers



(a) Performance overhead (%) of BEEBS programs compiled with O3 optimization.



(b) Performance overhead (%) of BEEBS programs compiled with Oz optimization.

Figure 8: Performance overhead (%) of BEEBS programs introduced by SHERLOC's holistic enforcement on a single-core Cortex-M33 microcontroller system.

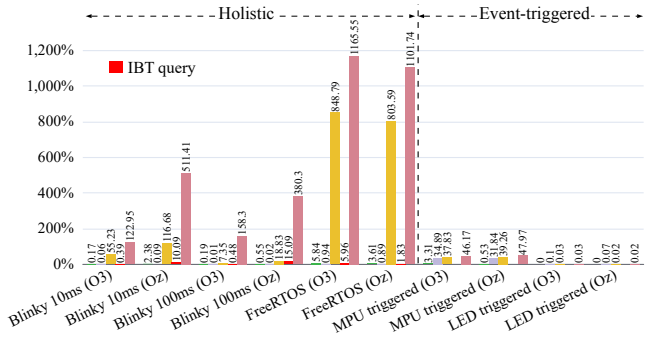


Figure 9: Performance overhead (%) of Blinky (10 ms and 100 ms toggling intervals) and FreeRTOS introduced by SHERLOC's holistic enforcement and MPU/LED-based event-triggered enforcement on a single-core Cortex-M33 microcontroller system. As for the forward-edge CFVD, we evaluated the performance introduced by the IBT query for Blinky and FreeRTOS.

on FreeRTOS: (i) FreeRTOS employs the ARMv8-M memory protection unit (MPU) to configure memory permissions of tasks and the kernel. Therefore, every context switch involves saving and restoring task and kernel-specific MPU settings. In the first experiment, SHERLOC is triggered by a write to any of the MPU registers; (ii) in the second experiment, we simulated less frequent scenarios where an attacker may attempt to access a sensitive peripheral, such as a LED. Thus, SHERLOC is only triggered by a write to the LED peripheral registers.

5.4.2 Results. We evaluated the number of CPU cycles that SHERLOC spends on (i) entering and exiting the secure state, (ii) reading through the trace buffer, (iii) transfer type identification, (iii) forward-edge CFVD or IBT query, (iv) backward-edge CFVD for BEEBS, and (v) the entire validation process, including (iii) to (iv). Figure 8 and Figure 9 illustrate the performance overhead compared to the unmonitored programs/systems.

As the results demonstrate, entering and exiting SHERLOC introduces negligible overhead due to the fast state switch mechanism of ARMv8-M. SHERLOC on BEEBS programs with O3 optimization only introduces 0.07% - 1.23% performance overhead for this step, while SHERLOC on BEEBS programs under Oz has a slightly higher overhead of 0.27% - 1.98%. Even for cases with a large number of task context switches, such as FreeRTOS O3 and Oz in Figure 9, entering and exiting SHERLOC only add 5.84% and 3.61% overhead, respectively. However, a higher entering and exiting overhead means that there are more trace records and the watermark has been hit more often. For instance, although bubblesort Oz has fewer non-sequential control-flow transfer instructions than bubblesort O3 as shown in Table 3, there are more run-time executions of those instructions (e.g., more loops). Hence, the entering and exiting overhead of bubblesort O3 is higher than that of bubblesort Oz.

Similarly to the overhead of entering and exiting, the overhead of reading through the trace buffer is highly dependent on the trace

size. In general, reading through the trace buffer is very fast. For BEEBS programs with O3 optimization, SHERLOC only introduces 0.01% - 0.24% performance overhead for this step, while for BEEBS programs with Oz optimization, there is a slightly higher overhead of 0.05% - 0.38%. Even in the worst-case scenario, FreeRTOS O3 and Oz only introduce 0.94% and 0.89% overhead, respectively.

Most of the SHERLOC overhead can be attributed to the validation stage, for which we will discuss optimization approaches in §7. The validation stage includes identifying the instruction type that originated a trace record, dispatching a record for handling based on the instruction type (e.g., query the IBT to validate address pairs if it is an indirect branch) and maintaining the RCS and task entry and re-entry list. We noted that the process of identifying the type of transfers constitutes over 75% of the overall time expended in the validation stage. For BEEBS programs with O3 optimization, SHERLOC only introduces 20.39% - 290.09% performance overhead for the entire validation step, while for BEEBS programs with Oz optimization, there is a higher overhead of 64.72% - 548.96%. The validation overhead of SHERLOC on the Blinky program with 10 ms toggling interval is 122.95% and 511.41% for O3 and Oz, respectively. The overhead of Blinky program with 100 ms toggling interval is 158.3% and 380.3% for O3 and Oz, respectively. For FreeRTOS with frequent context switches, the validation overhead is 1165.55% and 1101.74% for O3 and Oz, respectively. These results indicate that the number of interrupts and exceptions in complicated systems can have a significant impact on the performance of SHERLOC.

Diving deeper into the validation stage, for BEEBS (no interrupts) we interpret the IBT query as the forward-edge, while the updates and checks on the RCS form the bulk of the backward-edge. For Blinky and FreeRTOS, however, the forward-edges include IBT query, interrupt and exception handling, and task schedule handling. As O3 optimization generates faster code, it tends to use fewer indirect calls but more direct branches. As shown in Table 3, the O3 version of picojpeg, qrduino, and Blinky do not use any indirect calls, but the Oz version of them have 15, 14, and 110 indirect calls, respectively. On the other hand, the number of direct branches reduces significantly, with Blinky reducing from 2,040 direct branches to 67. More indirect transfers mean a larger IBT and hence higher run-time overhead of searching in the IBT. In general, we see Oz has a higher performance overhead than its O3 counterparts. The geometric means of forward-edge validation overheads for BEEBS under O3 and Oz are 0.38% and 1.11%, respectively. Meanwhile, the backward-edge validation overheads for the same benchmarks are higher, with geometric means of 29.61% and 64.81% for O3 and Oz, respectively. The IBT query overhead of Blinky program with 10 ms toggling interval is 0.39% and 10.09% for O3 and Oz, respectively. However, the IBT size is not the sole factor influencing run-time overhead. Despite ndes having fewer IBT entries than picojpeg, it performs indirect transfers more frequently, leading to a higher overhead. Additionally, for FreeRTOS which has a smaller IBT size than Blinky, the overhead incurred due to the IBT query is 5.96% and 1.83% for O3 and Oz, respectively.

As expected, we observed that SHERLOC's event-triggered enforcement results in much higher performance than the holistic enforcement approach. As aforementioned, current CFVD solutions that only monitor desktop applications all adopt this event-triggered approach. As shown in Figure 9, the event-triggered enforcement on FreeRTOS with frequent MPU updates only introduced a 46.17% overhead for O3 and a 47.97% for Oz, compared to the 1165.55% overhead for the holistic approach with O3 and 1101.74% with Oz. With even less frequent triggers, such as the LED-based trigger, SHERLOC only introduces a 0.03% overhead for O3 and 0.02% for Oz.

5.5 Comparison with Prior Approaches

Table 4 provides a comprehensive comparison of SHERLOC with prior approaches. The table evaluates various aspects, including targeted CPU architecture, protection targets, required hardware features, security guarantees (such as the ability to monitor privileged code and the entire system), backward- and forward-edge policies and precision, CFVD triggers, and self-reported run-time average or geometric mean performance overheads for each approach. Note that evaluation results are not solely dependent on the approach itself but also on the experimental setup, which can vary significantly from one project to another. For example, some approaches claim to work for RTOSs, but they were only evaluated on bare-metal systems. As a result, directly comparing the numerical values across different approaches may be misleading.

5.5.1 Comparison with Inlined Instrumentation Approaches for Embedded Systems. Prior works, such as RECFISH [48], μRAI [5], CaRE [38], TzmCFI [29], Silhouette [57], and Kage [20] have attempted to enforce control-flow integrity for embedded systems through inlined instrumentation. All of them, except CaRE, change the memory layout of the protection target. Neither RECFISH nor CaRE requires source code, while the others rely on source code.

RECFISH only protects unprivileged tasks on FreeRTOS and makes shadow stacks only accessible at the privileged level. μRAI achieves superior performance, i.e., 0.1%, and eliminates the need to spill return addresses to memory by using jump instructions, a reserved general-purpose register, and statically computed return address lookup tables to determine the correct return location at runtime. However, when calling a function from an uninstrumented library, μRAI needs to switch privilege levels and saves/loads reserved registers to/from a safe region, which is expensive.

Silhouette and Kage adopt a shadow stack combined with coarse-grained label-based CFI protections for bare-metal systems and RTOSs, respectively. To protect the shadow stack, they utilize unprivileged load or store instructions, necessitating instrumentation and instruction transformation. While both Silhouette and Kage exhibit superior performance than SHERLOC, they have certain limitations. Notably, they do not provide protection for startup code or libraries, unlike SHERLOC, which can safeguard the entire bare-metal system or RTOS. Moreover, Silhouette and Kage utilize a parallel shadow stack, leading to significant memory overhead and supporting only a limited number of tasks.

Both CaRE and TzmCFI use TrustZone and keep shadow stacks inside the secure state. CaRE targets bare-metal systems, while TzmCFI works on RTOSs. CaRE intercepts all function calls and return instructions to trap them into the secure state, while TzmCFI uses the updated exception trampoline to push the current and parent

Table 4: Comparison with prior approaches

		Targeted CPU	Protection targets	Required hardware features	Does not require source code?	Does not require instrumentation?	Can monitor privileged code?	Can monitor the entire system?	Backward-edge policy and precision	Forward-edge policy and precision	CFVD triggers	Run-time overhead on Bare-metal or Benchmark (%)	Run-time overhead on FreeRTOS (%)	Event-triggered overhead (%)
II	RECFISH [48]	R	Unprivileged task	MPU	✓	✗	✗	✗	● shadow stack	◑ coarse-grained	-	21	-	-
	μRAI [5]	M	Bare-metal		✗	✗	✓	✗	● return address integrity	◑ coarse-grained	-	0.1	-	-
	Silhouette [57]	M	Bare-metal	ULSI/MPU	✗	✗	✓	✗	● shadow stack	◑ coarse-grained	-	3.4	-	-
	Kage [20]	M	RTOS	ULSI/MPU	✗	✗	✓	✗	● shadow stack	◑ coarse-grained	-	-	5.2	-
	CaRE [38]	M	Bare-metal	TZ	✓	✗	✓	✓	● shadow stack	● fine-grained	-	513	-	-
	TzmCFI [29]	M	RTOS	TZ	✗	✗	✓	✓	● shadow stack	✗	-	84	-	-
CFVD	CFIMon [51]	I	Windows app	BTS	✓	✓	✗	✗	◔ call-proceded targets	◑ coarse-grained	critical syscalls	-	-	6.1
	PathArmor [47]	I	Linux app	LBR	✓	✗	✗	✗	◕ call/return matching	● fine-grained	critical syscalls	-	-	8.5
	CFIGuard [53]	I	Linux app	LBR/PMU	✗	✓	✗	✗	◑ whitelist targets	● fine-grained	PMU interrupts	-	-	2.9
	FlowGuard [34]	I	Linux app	PT	✓	✓	✗	✗	◑ whitelist targets	● fine-grained	critical syscalls	-	-	3.8
	GRIFFIN [23]	I	Linux app	PT	✓	✓	✗	✗	● shadow stack	● fine-grained	critical syscalls/watermark	-	-	11.9
	PT-CFI [27]	I	Linux app	PT	✓	✓	✗	✗	● shadow stack	✗	critical syscalls	-	-	21
	SHERLOC	M	RTOS/Bare-metal	TZ/MTB	✓	✓	✓	✓	● RCS	● fine-grained	critical operations/watermark	123.2	1106.2	0.09

II: inlined instrumentation; R: Cortex-R; M: Cortex-M; I: Intel; ULSI: unprivileged load or store instructions; TZ: TrustZone; LBR: last branch recording; PMU: performance monitor unit; BTS: branch trace store; PT: intel processor trace; coarse-grained: check only if the destination of an indirect control transfer is legitimate [27]; fine-grained: constrain the indirect transfer into a source-destination pair [27]; ● - ◔: precision in descending order; ✓: support; ✗: not support; −: not applicable. Note that evaluation results are not solely dependent on the approach itself but also on the experimental setup, which can vary significantly from one project to another and directly comparing the numerical values across different approaches may be misleading.

exception stack frame into the secure state shadow stack. In comparison, SHERLOC outperforms CaRE in protecting bare-metal systems. While TzmCFI demonstrates better performance than SHERLOC in benchmarks, it lacks explicit protection for forward edges, relying instead on the built-in LLVM CFI implementation [46]. TzmCFI did not report the performance overhead of protecting an entire RTOS.

5.5.2 Comparison with CFVD Approaches for Desktop Systems. None of the existing CFVD approaches for desktop systems, including CFIGuard [53], PathArmor [47], CFIMon [51], FlowGuard [34], GRIFFIN [23], and PT-CFI [27], are capable of monitoring privileged code. This underscores one of the key strengths of SHERLOC, which extends CFVD to cover privileged code and adeptly handles asynchronous interrupts.

Furthermore, most existing CFVD approaches enforce less precise backward- and forward-edge policies. For instance, CFIMon uses Intel's branch trace store (BTS) to collect branch information and checks if the executed path is in the pre-collected control transfer sets. PathArmor uses Intel's last branch record (LBR) for tracing, hooks sensitive system calls, and triggers the inspection when those system calls are invoked. It uses call/return matching to verify returns. CFIGuard utilizes the performance monitoring unit (PMU) to trigger verification. It uses a fine-grained CFG to get a strict target table and validates the indirect branches from LBR at run-time. However, LBR is vulnerable to history-flushing attacks [41, 42]. FlowGuard, GRIFFIN, and PT-CFI utilizes Intel's processor trace (PT), which records indirect calls/jumps and returns. FlowGuard first constructs an indirect branch Graph by running the target application and marks the covered execution paths with more vital credits and labels the non-covered edges with fewer credits. At run-time, FlowGuard compares the trace records with these credit edges. GRIFFIN and PT-CFI utilize shadow stack, which provides the same level of precision as SHERLOC's RCS mechanism. While GRIFFIN enforces fine-grained policy on forward edges, PT-CFI does not protect forward edges. Note that all of the aforementioned systems provide evaluation results on multicore processors.

6 RELATED WORK

6.1 Other Control-Flow Protections on Embedded Systems

Besides the inlined instrumentation approaches discussed in §5.5.1, other projects offer various levels of control-flow protections for embedded systems. Ret2ns [35] reports the fast state switch mechanism of Cortex-M TrustZone can be exploited for cross-state control-flow hijacking. To address this vulnerability, ret2ns proposes software-fault isolation (SFI) based mechanisms to govern cross-state control flows. uSFI [11], ACES [16], and MINION [31] compartmentalizes embedded software functions and peripherals into separate domains for better isolation. These approaches enforce control-flow validations across compartments, offering limited control-flow protections. However, they come with additional compartment context switch overhead. An alternate approach is to use memory-safe languages, such as Rust, to program embedded systems, as demonstrated in Tock [32]. However, microcontroller-based embedded systems often require direct low-level hardware access, such as through memory-mapped registers. For tasks like these, the use of `unsafe` blocks in Rust is necessary, which can potentially introduce vulnerabilities to control-flow hijacking.

6.2 Other Trace-based Security Approaches

Besides the CFVD approaches discussed in §5.5.2, other trace-based mechanisms were proposed for heuristic-based CFVD and various other purposes. For instance, kBouncer [40] and ROPecker [15] are security mechanisms developed to combat ROP attacks. kBouncer operates by comparing the executed gadget chain in LBR against heuristic ROP attack patterns. ROPecker discerns valid gadget chains via records in the LBR obtained through emulating the execution of a legitimate program. Bunkerbuster [52] employs PT and introduces a method for bug hunting that relies on symbolically reconstructing states based on execution traces. μAFL [33] leverages the Embedded Trace Macrocell (ETM) and fuzzing techniques to discover vulnerabilities in embedded systems.

7 LIMITATIONS AND FUTURE OPTIMIZATION

7.1 Limitations

The holistic enforcement approach of SHERLOC may not be applicable to time-sensitive tasks. Specifically, the non-secure interrupts can be blocked and deferred for later handling if they occur during the execution of trace validation. One potential solution to overcome this limitation is to set the priority of the secure DebugMon to the lowest level, ensuring that non-secure interrupts take precedence and are served immediately. However, implementing this approach would involve incorporating tracing enabling instructions within non-secure ISRs, which goes against our objective of achieving non-instrumentation in this paper.

Another limitation of SHERLOC is that it relies on task entry addresses generated through heuristics during offline analysis. To determine the heuristics used in generating such addresses, manual analysis of a specific RTOS is required since different RTOSs employ distinct strategies for task management and scheduling. Future work should aim to automate this process.

7.2 Future Optimization

The performance of SHERLOC's holistic enforcement can be significantly improved with the following approaches: (i) On multicore microcontrollers, such as the NXP LPC55S6x which has dual Cortex-M33 cores and TrustZone support [37], SHERLOC could execute concurrently with the protected system, eliminating the need to pause it during analysis. In fact, current application-oriented CFVD solutions [23, 27, 34, 47, 51, 53] have reported better performance than SHERLOC mainly because they are evaluated on multicore CPUs where the analysis does not block the monitored application's execution; (ii) In our prototype implementation, SHERLOC directly dereferences the original instruction from flash in the non-secure state. While this results in a simpler implementation, accessing from flash is much slower than accessing from SRAM. An optimization approach would be to encode the type of every instruction, such as its opcode, during offline analysis and load such information into the SRAM for faster run-time access.

8 CONCLUSION

Microcontroller-based embedded systems are interrupt-driven, with the majority of processing taking place in the privileged mode, and they are vulnerable to control-flow hijacking attacks. However, current CFVD mechanisms are limited to monitoring unprivileged applications, making it essential to provide a system-oriented CFVD solution that monitors control-flow transfers between privileged and unprivileged components. In this paper, we formalized the problem and designed SHERLOC, a secure and holistic CFVD system for the ARMv8-M architecture. SHERLOC monitors forward and backward edges of unprivileged and privileged programs, as well as control-flow transfers among unprivileged and privileged components. To ensure security, SHERLOC utilizes the trusted execution environment, preventing privileged programs from bypassing monitoring or tampering with the trace. In addition, SHERLOC supports event-triggered analysis as current CFVD approaches. Our experiments demonstrated the effectiveness of SHERLOC, and performance evaluations showed its efficiency.

ACKNOWLEDGMENT

Many thanks to our shepherd and anonymous reviewers for their thoughtful reviews. This material is based upon work supported in part by National Science Foundation (NSF) grants (2237238 and 2037798) and a National Centers of Academic Excellence in Cybersecurity grant (H98230-22-1-0307).

REFERENCES

[1] 2014. Capstone disassembly/disassembler framework. https://www.capstone-engine.org/. (2014).
[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity principles, implementations, and applications. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[3] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. A theory of secure control flow. In *International Conference on Formal Engineering Methods and Software Engineering (ICFEM)*.
[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*.
[5] Naif Saleh Almakhdhub, Abraham A Clements, Saurabh Bagchi, and Mathias Payer. 2020. μRAI: Securing Embedded Systems with Return Address Integrity. In *Network and Distributed Systems Security (NDSS) Symposium*.

[6] Arm. 2015. Armv8-M Architecture Reference Manual. https://developer.arm.com/documentation/ddi0553/bm/. (2015).

[7] Arm. 2017. AN505: Cortex™-M33 with IoT kit FPGA for MPS2+ Version 2.0. (2017). https://developer.arm.com/tools-and-software/development-boards/fpga-prototyping-boards/download-fpga-images.

[8] Arm. 2017. Arm Compiler armclang Reference Guide. (2017). https://developer.arm.com/documentation/100067/0609/.

[9] Arm. 2017. ARM IOT-Kit CM33 Secure/Non-Secure Blinky project. (2017). https://www.keil.com/dd2/arm/iotkit_cm33/.

[10] Arm. 2017. Arm MPS2+ FPGA prototyping board. (2017). https://developer.arm.com/tools-and-software/development-boards/fpga-prototyping-boards/mps2.

[11] Zelalem Birhanu Aweke and Todd Austin. 2018. uSFI: Ultra-lightweight software fault isolation for IoT-class devices. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[12] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. 2014. Hacking blind. In *IEEE Symposium on Security and Privacy*.

[13] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*.

[14] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *IEEE Symposium on Security and Privacy (SP)*.

[15] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attack. In *Network and Distributed Systems Security Symposium (NDSS)*.

[16] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. ACES: Automatic Compartments for Embedded Systems. In *USENIX Security Symposium*.

[17] Thurston HY Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*.

[18] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. 2010. Return-oriented programming without returns on ARM. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[19] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*.

[20] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J Walls, and John Criswell. 2022. Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage. In *USENIX Security Symposium*.

[21] Tommaso Frassetto, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi. 2022. CfInsight: A comprehensive metric for CFI policies. In *Network and Distributed System Security Symposium (NDSS)*.

[22] FreeRTOS. 2023. FreeRTOS. https://www.freertos.org/. (2023).

[23] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. Griffin: Guarding control flows using intel processor trace. *ACM SIGPLAN Notices*.

[24] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained control-flow integrity for kernel software. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.

[25] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*.

[26] Jens Grossklags and Claudia Eckert. 2018. τcfi: Type-assisted control flow integrity for x86-64 binaries. In *Research in Attacks, Intrusions, and Defenses (RAID)*.

[27] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent backward-edge CFVD using intel processor trace. In *ACM on Conference on Data and Application Security and Privacy (CODASPY)*.

[28] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf. (2016).

[29] Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. 2020. TZmCFI: RTOS-Aware Control-Flow Integrity Using TrustZone for Armv8-M. *International Journal of Parallel Programming*.

[30] Mustakimur Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-sensitive Control Flow Integrity. In *USENIX Security Symposium*.

[31] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. 2018. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In *Network and Distributed System Security Symposium (NDSS)*.

[32] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kb computer safely and efficiently. In *ACM SIGOPS symposium on Operating systems principles (SOSP)*.

[33] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. μAFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware. In *International Conference on Software Engineering (ICSE)*.

[34] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and efficient cfi enforcement with intel processor trace. In *IEEE International Symposium on High performance computer architecture (HPCA)*.

[35] Zheyuan Ma, Xi Tan, Lukasz Ziarek, Ning Zhang, Hongxin Hu, and Ziming Zhao. 2023. Return-to-Non-Secure Vulnerabilities on ARM Cortex-M TrustZone: Attack and Defense. In *ACM/IEEE Design Automation Conference (DAC)*.

[36] Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. 2019. Analyzing control flow integrity with LLVM-CFI. In *Annual Computer Security Applications Conference (ACSAC)*.

[37] NXP. 2023. LPC55S6x. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc5500-arm-cortex-m33/high-efficiency-arm-cortex-m33-based-microcontroller-family:LPC55S6x. (2023). Accessed: 05-01-2023.

[38] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N Asokan. 2017. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*.

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

[40] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2013. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security*.

[41] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. 2014. Evaluating the effectiveness of current anti-ROP defenses. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*.

[42] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit hardening made easy. In *USENIX Security Symposium*.

[43] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[44] Philip Sparks. 2021. The route to a trillion devices. https://community.arm.com/iot/b/blog/posts/whitepaper-the-route-to-a-trillion-devices. (2021).

[45] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*.

[46] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC and LLVM. In *USENIX Security Symposium*.

[47] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[48] Robert J Walls, Nicholas F Brown, Thomas Le Baron, Craig A Shue, Hamed Okhravi, and Bryan C Ward. 2019. Control-flow integrity for real-time embedded systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*.

[49] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *Network and Distributed Systems Security (NDSS) Symposium*.

[50] Nathanael R Weidler, Dane Brown, Samuel A Mitchel, Joel Anderson, Jonathan R Williams, Austin Costley, Chase Kunz, Christopher Wilkinson, Remy Wehbe, and Ryan Gerdes. 2017. Return-oriented programming on a cortex-m processor. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*.

[51] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[52] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. 2021. Automated bug hunting with data-driven symbolic root cause analysis. In *ACM SIGSAC Conference on Computer and Communications Security*.

[53] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. 2015. Hardware-assisted fine-grained code-reuse attack detection. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*.

[54] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy*.

[55] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. 2014. A platform for secure static binary instrumentation. In *ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*.

[56] Mingwei Zhang and R Sekar. 2013. Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks. In *USENIX Security Symposium*.

[57] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J Walls. 2020. Silhouette: Efficient protected shadow stacks for embedded systems. In *USENIX Security Symposium*.

[58] Lipeng Zhu, Xiaotong Fu, Yao Yao, Yuqing Zhang, and He Wang. 2019. FIoT: Detecting the memory corruption in lightweight IoT device firmware. In *IEEE International Conference On Trust, Security And Privacy (TrustCom)*.