Bitmap-Based Security Monitoring for Deeply Embedded Systems

ANNI PENG, National Computer Network Intrusion Protection Center, UCAS, Beijing, China
DONGLIANG FANG, Institute of Information Engineering, CAS, UCAS, Beijing, China
LE GUAN, University of Georgia, Athens, GA, USA
ERIK VAN DER KOUWE, Vrije Universiteit Amsterdam, Amsterdam, Netherland
YIN LI, National Computer Network Intrusion Protection Center, UCAS, Beijing, China
WENWEN WANG, University of Georgia, Athens, GA, USA
LIMIN SUN, Institute of Information Engineering, CAS, UCAS, Beijing, China
YUQING ZHANG, National Computer Network Intrusion Protection Center, UCAS, Beijing, China

Deeply embedded systems powered by microcontrollers are becoming popular with the emergence of Internet-of-Things (IoT) technology. However, these devices primarily run C/C++ code and are susceptible to memory bugs, which can potentially lead to both control data attacks and non-control data attacks. Existing defense mechanisms (such as control-flow integrity (CFI), dataflow integrity (DFI) and write integrity testing (WIT), etc.) consume a massive amount of resources, making them less practical in real products. To make it lightweight, we design a bitmap-based allowlist mechanism to unify the storage of the runtime data for protecting both control data and non-control data. The memory requirements are constant and small, regardless of the number of deployed defense mechanisms. We store the allowlist in the TrustZone to ensure its integrity and confidentiality. Meanwhile, we perform an offline analysis to detect potential collisions and make corresponding adjustments when it happens. We have implemented our idea on an ARM Cortex-M-based development board. Our evaluation results show a substantial reduction in memory consumption when deploying the proposed CFI and DFI mechanisms, without compromising runtime performance. Specifically, our prototype enforces CFI and DFI at a cost of just 2.09% performance overhead and 32.56% memory overhead on average.

CCS Concepts: • **Computer systems organization → Embedded and cyber-physical systems**; • **Security and privacy → Systems security**;

Additional Key Words and Phrases: CFI, DFI, microcontroller, TEE

This work was supported in part by National Key Research and Development Program of China (Nos. 2023YFB3106400 and 2023QY1202), the National Natural Science Foundation (NSF) of China (U2336203 and U1836210), the Key Research and Development Science and Technology of Hainan Province (GHYF2022010), Beijing NSF (4242031), Beijing NSF (Grant No. L234033), the Netherlands Organization for Scientific Research (NWO) (Grant No. VI.Veni.202.212 VENI "Vulcan"), and U.S. NSF (Grant No. 2238264). Authors' Contact Information: Anni Peng, National Computer Network Intrusion Protection Center, UCAS, Beijing, China; e-mail: pengan@nipc.org.cn; Dongliang Fang, Institute of Information Engineering, CAS, UCAS, Beijing, China; e-mail: fangdongliang@iie.ac.cn; Le Guan, University of Georgia, Athens, GA, USA; e-mail: leguan@cs.uga.edu; Erik van der Kouwe, Vrije Universiteit Amsterdam, Amsterdam, Netherland; e-mail: vdkouwe@cs.vu.nl; Yin Li, National Computer Network Intrusion Protection Center, UCAS, Beijing, China; e-mail: liyin@nipc.org.cn; Wenwen Wang, University of Georgia, Athens, GA, USA; e-mail: wenwen@cs.uga.edu.cn; Limin Sun, Institute of Information Engineering, CAS, UCAS, Beijing, China; e-mail: sunlimin@iie.ac.cn; Yuqing Zhang (Corresponding author), National Computer Network Intrusion Protection Center, UCAS, Beijing, China; e-mail: zhangyq@nipc.org.cn.

ACM 1557-7392/2024/9-ART192
https://doi.org/10.1145/3672460

ACM Reference format:
Anni Peng, Dongliang Fang, Le Guan, Erik van der Kouwe, Yin Li, Wenwen Wang, Limin Sun, and Yuqing Zhang. 2024. Bitmap-Based Security Monitoring for Deeply Embedded Systems. *ACM Trans. Softw. Eng. Methodol.* 33, 7, Article 192 (September 2024), 31 pages.
https://doi.org/10.1145/3672460

1 Introduction

Deeply embedded systems powered by **Microcontroller Units (MCUs)** increasingly affect our daily lives, as exemplified in health care, autonomous driving, home security, and so on. Following the **Internet-of-Things (IoT)** trend, devices are more and more connected, which enlarges the attack surface, making it challenging to safeguard the security of these devices. For example, attackers were able to exploit firmware vulnerabilities to remotely hijack the target device or launch Denial of Service attacks [10, 32, 43, 45, 73]. They have successfully compromised a variety of devices, including smart lights [64], smart cars [36, 40, 48], and smart medical devices [30, 57].

Since it is almost impossible to eliminate all software bugs, mitigating the impact of exploitation becomes critical. Over the years, **Control-Flow Integrity (CFI)** [55, 74] and **Dataflow Integrity (DFI)** [16, 51] have established themselves as two important security properties we want to enforce in a computing system. Specifically, CFI is violated when the control data (e.g., function pointers and return addresses) are corrupted. DFI is violated whenever a variable is illegally accessed. Existing work primarily focuses on enforcing CFI on embedded devices, as demonstrated by a recent survey [54] that reported the absence of data integrity protection in these systems. Current approaches struggle to prevent data-only attacks, and enforcing DFI online typically necessitates heavy instrumentation for dataflow tracking. While existing CFI/DFI solutions have shown encouraging results in protecting commodity OSs, they consume considerable memory and computation resources [5, 13, 15, 16], rendering them inviable in resource-constrained MCU-based embedded systems (e.g., those with only 256 kB RAM). For example, forward CFI stores an allowlist of legitimate branching targets for every indirect call site. Backward CFI maintains a separate shadow memory to detect return address corruption. DFI needs to instrument every memory access instruction and check the allowlist to detect unauthorized memory accesses, thus incurring high overhead. Moreover, to provide comprehensive protection, we need to simultaneously enforce these properties, stacking up the pressure they place on device resources. For example, if we enforce the DFI proposed in [16] and the CFI proposed in [42], assuming they do not conflict with each other, the overall memory overhead would be 50% plus 7.5%.

In this work, we propose a novel approach named **Bitmap-Based Security Monitoring (BSM)**. Our solution is partly inspired by the remote attestation-based solutions [4, 70]. This previous work, however, works in offline mode, only detecting attacks after the fact. Attestation-based solutions generally involve the collection and transmission of runtime data to a remote backend server, which subsequently performs an "offline" verification to ascertain the legitimacy of the execution. *BSM*, on the other hand, is online and does not require any backend server. It implements *online* security monitoring mechanisms based on the design of a compact, bitmap-based runtime data structure, which is a unified whitelist for both CFI and critical DFI. *BSM* enforces CFI by performing checks at indirect control-flow transfers and enforces DFI protection based on a data watchpoint mechanism, which performs checks *only when the write operation occurs*, avoiding the expense of checking all potentially risky writes. Each bit in the bitmap represents a valid operation, whose index is determined by an address tuple (A, B). Here, a hash function is applied to A and B to distribute the tuple to a random index of the bitmap. This bit indicates whether this address tuple should be allowed or not. We will later show that such a tuple is sufficient to

represent both control-flow transition and dataflow, thus covering the aforementioned CFI and DFI proposals. Depending on the firmware, the size of the bitmap is adjustable to minimize collisions. This novel data structure is compact compared with existing solutions, especially when CFI and DFI are deployed simultaneously, which is critical on deeply embedded devices, which have strict resource limitations.

The allowlist has to be isolated in a secure place where a memory corruption cannot touch it. This can help to ensure its integrity (cannot be modified by vulnerable code) and confidentiality (cannot be leaked to vulnerable code). To achieve this goal, we leverage the hardware-enabled **Trusted Execution Environment (TEE)** for MCU devices, such as MultiZone [38] on **Reduced Instruction Set Computer (RISC)**-V MCUs or TrustZone [7, 52] on **Advanced RISC Machine (ARM)** Cortex-M MCUs. By placing the allowlist in the secure world, we can update it directly by utilizing a secure API in the normal world. The protected firmware, while running in the normal world, reports security-critical operations to the TEE secure world, where security checks are performed.

Due to our limited resources, the size of the bitmap is constrained. This constraint, combined with the use of a hash algorithm, introduces the possibility of collisions. To detect potential collisions, we perform offline analysis. If collisions are inevitable, our tool automatically extends the bitmap. Specifically, for potential CFI-induced collisions, we treat all return instruction addresses and indirect calls in the program as possible A values in the tuple (A, B) and all addresses of attack-useful functions and **Return-oriented Programming (ROP)** gadgets as possible B values in the tuple (A, B). We then generate a series of illegal tuples by computing the Cartesian product of these two sets. For potential DFI-induced collisions, we generate illegal tuples for each critical variable using all other critical store instructions (not allowed to write the chosen variable). For each element in the illegal tuples for both CFI and DFI, we compute the illegal index using the same hash function. Finally, we check these illegal indexes against the allowlist record. If a collision is found, we increase the bitmap size, regenerate the allowlist, and repeat the previous step. Otherwise, the allowlist is collision-free for both CFI and DFI.

We implemented the proposed system on an ARM Cortex-M-based MCU with TrustZone support. Using BSM to deploy both CFI and DFI, our prototype incurs 32.56%, 10.79%, and 2.09% overhead in terms of RAM, FLASH, and performance. We also measured the power consumption with and without *BSM*. The results show that the average overhead of power consumption is only 0.59%.

In summary, we made the following contributions:

- —We propose a novel bitmap-based framework, which offers online enforcement for both CFI and critical data integrity while keeping the overhead acceptable for deeply embedded devices.
- —We have an offline analysis process for the bitmap storage structure to avoid conflicts, and we store the bitmap in TrustZone to ensure its integrity and confidentiality.
- —We have implemented our idea on an ARM Cortex-M-based MCU. Our evaluation results demonstrate a significantly lower overhead when compared to traditional solutions, highlighting the efficiency and effectiveness of our approach.

2 Background

Deeply Embedded Systems. Deeply embedded systems are designed to perform a determined set of specific tasks, usually on low-c-cost, high-reliability platforms with severe resource limitations [34]. For example, some battery-powered embedded systems can get a fresh battery charge daily, but others must last months or years on a single battery [46]. Following the categorization proposed by Abbasi et al. [3], we can recognize deeply embedded systems as a subset of embedded systems having the following characteristics: (1) they are powered by an MCU, which integrates a processor,

memory, and other peripheral devices onto a single chip; (2) they have a minimal OS (such as **Free Real-Time Operating System (FreeRTOS))** or no OS at all (i.e., baremetal); and (3) the application lack user interface or have uncommon ones to fit the constrained memory size, computation power, and energy consumption. Consequently, in such systems, all resources that are deemed irrelevant to the tasks they need to perform are eliminated to minimize production costs.

Typically, they consist of microcontrollers that are low-end processors with integrated memory, performing specific operations in a deterministic manner and collaborating to accomplish complex tasks. For example, modern vehicles can have more than a hundred individual computing units (consists of microcontrollers, also called electronic control units), which are individually customized to control different functionalities of the vehicle [47]. Moreover, such systems are broadly embedded in various application scenarios, such as small home [44], **Industrial Control Systems (ICSs)** [24], automotive [47], medical devices [69], and other fields.

Since these systems closely interact with the physical world, attacking them can cause not only software errors but also physical damages. For example, in smart cars, embedded devices control the vehicle's behavior, ranging from non-critical infotainment systems to extremely critical driver assistance systems [47]. If these systems fail, they may endanger people's lives. In addition, the famous "Stuxnet" attack infected surveillance and data acquisition systems [29] of nuclear centrifuges, causing significant damage to Iran's nuclear program. Moreover, researchers have demonstrated various exploitation methods against such deeply embedded systems in industrial control applications [2, 33, 35].

Data Watchpoints. Data watchpoints are a very common debugging feature. With this feature, whenever an interesting variable is accessed, the debugger halts the execution and notifies the user. This function is widely supported by mainstream MCUs (e.g., ARM [52], **Microprocessor without Interlocked Pipeline Stages (MIPS)** [50], RISC-V [67]) at the hardware level. For ARM Cortex-M MCUs, data watchpoint is implemented by the **Data Watchpoint and Trace Unit (DWT)**, an ARM intellectual property for data watchpoint, PC tracing, system profiling, and many helpful debugging functions.

DWT monitors a contiguous memory region based on the access types (e.g., read, write, or execute). The region is configured via a pair of comparators. In our experiment board, 16 pairs of comparators are supported, indicating that 16 memory regions can be monitored simultaneously. If the monitored memory is accessed, the hardware halts the execution or generates a debug monitor exception, depending on the configuration of the **Debug Exception and Monitor Control Register (DEMCR)**. We leverage the debug monitor exception mode in this work. On taking a debug monitor exception, the hardware stores contextual information onto the pre-exception stack. This operation is referred to as *stacking*, and the structure is referred to as a *stack frame*. The stored contextual information includes `R0–R3`, `R12`, `R14`, `xPSR` registers, and the return address, along with the floating point registers (if the floating point unit is active). Since the location of the stack frame is known, this contextual information can be obtained and manipulated in the exception handler (i.e., `DebugMon_Handler`).

ARM TrustZone. ARM TrustZone provides a hardware-based TEE for ARM processors. The feature divides the system into two execution environments: the secure world and the non-secure (or normal) world. The processor is time shared by the two environments. Correspondingly, the system's resources (such as memory and peripherals) are split into secure ones and non-secure ones. When the CPU is in the normal world, the CPU can only access non-secure resources. When the CPU is in the secure world, the CPU can access all the resources.

The transition between the secure world and normal world is realized in two ways: secure calls or exceptions. Secure calls allow the normal world firmware to invoke functions implemented in

the secure world via well-defined entry points. The entry is guarded by a special instruction called **Secure Gateway (SG)**. Exceptions can also cause cross-world transitions depending on whether they are banked or not. Banked exceptions have separate resources in both worlds to handle their own exceptions, so world transition is unnecessary. Non-banked exceptions such as debug monitor exception can only target one security state. As such, by configuring the DEMCR [52], we can also force the debug monitor exception to always target the secure world, even if the processor is currently running in the normal world.

ARM TrustZone ensures that code and data in the secure world cannot be affected by the normal world, even if vulnerabilities allow an attacker to achieve arbitrary execution there. Therefore, we store the allowlist needed for security monitoring in the secure world, where it cannot be tampered with even if memory protection is compromised.

3 Overview

We first describe the threat model we consider in this work. Then, we provide an overview of the design of the proposed system.

Threat Model. We consider an attacker who can send data over all Input/Output interfaces (e.g., network and **Universal Asynchronous Receiver/Transmitter (UART)**) of the target device. Therefore, they could craft and send manipulated input to trigger vulnerabilities in the firmware. In this work, we specifically consider vulnerabilities caused by memory errors that can be leveraged to violate the control flow or dataflow of the firmware execution. For example, the attacker can use write-what-where style vulnerabilities to overwrite a function pointer to hijack the control flow. They can also launch data-only attacks by overwriting a critical variable to indirectly influence the program logic. This is aligned with existing research on CFI and DFI [6, 56, 70].

We assume the hardware is equipped with a data watchpoint unit and a TEE environment. The former is a common debugging unit available in many MCU chips (such as ARM [52], RISC-V [67], and MIPS [50]), while the latter has already been deployed in the current generation of ARM-based and RISC-V-based chips. We also assume the trusted software in the TEE is bug-free and isolated from the normal world firmware, as important metadata, such as the allowlist, is stored there. Considering the small code base of the TEE-side software and the limited attack surface, we believe this is a reasonable assumption well accepted by existing TEE-based work [4, 56, 70]. We do not consider physical attacks, such as connecting to a Joint Test Action Group debugger to re-program the firmware. Finally, we assume our compiler passes are free of bugs.

System Overview. BSM, a bitmap-based security mechanism, is designed for deeply embedded devices. It leverages the TEE and the data watchpoint feature to efficiently enforce CFI and DFI for critical variables. While the concept is applicable to any chip with a TEE and data watchpoint unit, our implementation focuses on ARM Cortex-M-based devices due to their widespread usage.

BSM comprises a compile-time phase and a runtime phase, as illustrated in Figure 1. During the compile-time phase, our custom **Low Level Virtual Machine (LLVM)** pass performs static analysis (Sections 4.1 and 4.2) to extract tuples that form an allowlist for control-flow transitions and data accesses. These tuples are then encoded and stored in a compact bitmap-based allowlist in TEE. Furthermore, to streamline the collection of these tuples, BSM instruments the firmware to enable the gathering of control-flow transition tuples. Additionally, BSM utilizes the DWT hardware feature to automatically monitor critical data access and collect corresponding tuples. To achieve this, our LLVM pass performs an automatic analysis of user-defined critical variables, expands the set of critical variables to include those that propagate into them, and remaps them to a memory range monitored by the DWT. During the runtime, when the program encounters an indirect branch, it carries the collected tuple (consisting of the source instruction and destination address)

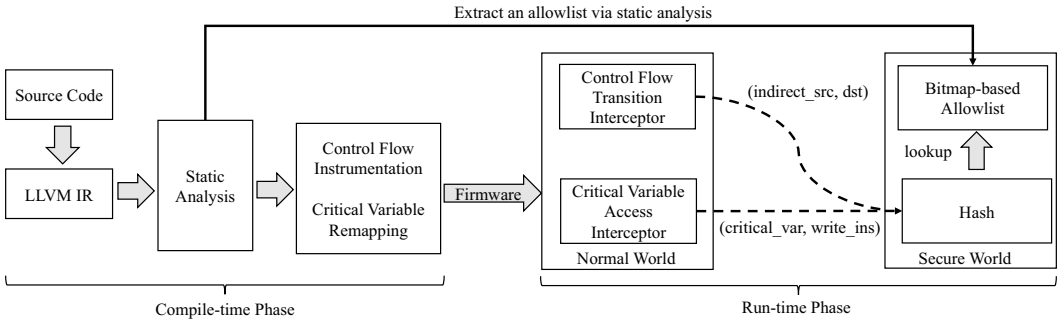

Fig. 1. *BSM* overview. IR, intermediate representation.

and transitions to the secure world for further validation (Section 4.1). When the program accesses a critical variable, a debug monitor exception is triggered, prompting the handler to dynamically extract the tuple (Section 4.2). Both CFI and DFI tuples are then passed from the normal world to the secure world, where the same hash algorithm is applied to obtain an index (mapping it into a bit) for each tuple. If the corresponding bit in the bitmap is set, the operation is considered legitimate. Otherwise, it is deemed illegal, and the context is recorded for postmortem analysis.

4 Design

In this section, we first elaborate on how to collect the runtime CFI tuples and DFI tuples, respectively. Then, we explain the TEE-side design that uses a bitmap-based allowlist to perform online security checking based on the obtained tuples.

4.1 CFI

There are generally three types of control-flow transfers in a program: direct jumps/calls, indirect jumps/calls, and function returns. Among them, attackers are particularly interested in indirect jumps/calls and function returns since overwriting the target address leads to the control-flow hijacking of the program. This is because the target addresses of these transfers are not hardcoded but determined at runtime, making them susceptible to manipulation.

According to the ARM instruction set, indirect jumps/calls are implemented by the `BLX Rx` instruction where the register `Rx` specifies the target address. The function returns use the `POP {..., PC}` instruction to pop a previously stored return address on the stack to the program counter `PC`. The attackers can leverage the `BLX` or `POP` instruction to redirect the control flow to an attacker-chosen target by manipulating the content of related registers or memory. We mark these exploitable instructions as *critical instructions*. To mitigate such control-flow hijacking attacks, our CFI enforcement scheme does a legitimacy check before each *critical instruction* to ensure that the firmware can only transfer to a legitimate target.

Extracting Valid Tuples. To verify the legitimacy of a control-flow transfer at runtime, we must know which targets are valid. For example, before executing the `BLX R0` instruction, we must know the set of legitimate values of *R*0. We call the address of the indirect jump site (i.e., the address of the `BLX R0` instruction) the *source*, and its target (i.e., the value of `R0`) the *destination*. Therefore, each valid control-flow transition is encoded as a tuple comprised of a source and a destination. We use this information to check the legitimacy of indirect control-flow transfer at runtime.

To obtain such information, our approach begins by constructing a call graph, which extracts a list of callers and callees for each function. However, constructing an accurate call graph poses

challenges, particularly in the presence of indirect function calls. To address this issue, our call graph building process leverages type-based alias analysis, which aids in resolving potential targets. More specifically, when encountering a `BLX Rx` instruction, we utilize the function type, similar to the forward CFI implementation in LLVM [49], to identify all possible targets. Here, we consider two function types to be identical if they have the same number of parameters, parameter types (such as structure or pointer), and return value types. As a result, for forward indirect branch instructions, their targets can be directly extracted from the constructed call graph. For backward indirect instructions such as return instructions, their potential return targets are determined to be the corresponding call site in the respective callers.

Instrumentation. To validate each indirect jump/call or function return at runtime, we need to insert instrumentation before the relevant *BLX* and *POP* instructions to pass the tuples to the CFI validation module. To protect the validation module and the allowlist, we place them in the TrustZone secure world so that they can only be accessed by the corresponding trampolines.

4.2 Critical DFI

Data integrity properties can be broadly categorized into two distinct aspects: read integrity and write integrity. While read integrity safeguards against unauthorized memory read operations, write integrity prevents unauthorized alterations to memory locations. The concept of write integrity was first introduced in **Write Integrity Testing (WIT)** [5]. In the context of our approach, write integrity is of greater significance than read integrity for several reasons: (1) DOP attacks [39] rely on violations of write integrity. (2) Read accesses occur more frequently than write accesses [15], making their protection more costly and less practical. (3) Prior research on general platforms has demonstrated that enforcing write integrity incurs substantial overhead. For instance, WIT [5] reported a performance overhead of approximately 5–25% for the SPEC benchmark. By minimizing this overhead, our approach enhances the likelihood of practical deployment. However, reducing overhead through selective write integrity enforcement is not a non-trivial task. For a given critical variable, its attack surface encompasses all write instructions apart from the legitimate instructions that are authorized to modify it. Consequently, a solution must ensure that modifications made by all write operations are scrutinized. This stems from the inherent nature of the C/C++ programming language, which allows any memory write instruction to potentially alter the value of any variable. This poses challenges in designing defense mechanisms, as we must monitor all memory write instructions even if we only protect critical variables.

Our solution attempts to perform security checks only when necessary to minimize the instrumentation overhead. By relocating all critical variables to a contiguous memory region, we can leverage a common debug feature to centrally monitor all semantically critical variables. As a result, there is no overhead until a critical variable is being written at runtime. When this occurs, the hardware automatically traps the execution, and the legitimacy of the trapped write instruction is verified. In essence, our approach avoids expensive dataflow tracking by involving hardware to passively monitor access to critical variables.

At a high level, our first step is to extract the critical variables. Determining which variables should be selected generally requires domain knowledge of specific application semantics (see Section 4.2.1). Next, we extract all legal write instructions for each critical variable, and this information will serve as allowlist in the bitmap. Then, we remap the critical variables in a continuous range of memory, which is write-monitored by data watchpoint (i.e., debug monitor mode of ARM DWT mechanism). In our design, we split monitored memory into three sub-regions: meaning that a critical variable can appear in any of the stack, heap, or global memory. Finally, we re-organize the extracted information into tuples (A, B), where A represents the target critical variables to write to

and B represents the its allowable write instructions. These tuples will be encoded and stored in the compact bitmap-based allowlist in TEE. As a result, once the program writes to a critical variable, it will trigger a debug_monitor event and lead to the legitimacy checks against the bitmap.

4.2.1 Critical Variable Extraction. Full DFI is expensive and not viable on resource-constrained embedded systems. To enable efficient data integrity, our approach is selective. Selective protection of semantically critical variables can raise the bar for successful exploitation, e.g., a variable indicating whether or not the user has been authenticated, or a variable in the syringe pump program that controls the number of injections. Generally, selecting variables heavily depends on application specific semantics and protection goals [15, 18, 70]. In our design, we assume critical variables have been provided as such, either manually by the programmer using explicit annotations or using an automated system. Determining which variables is critical and should be selected is orthogonal to our work.

Our implementation relies on programmers to annotate an initial set of semantically critical variables. This depends on domain knowledge of specific program semantics, which is not available in a compiler. Based on these annotations, our compiler automatically extracts critical variables using dataflow dependency analysis. This analysis has two steps: (1) expand the set of critical variables and (2) associate critical variables with a specific memory allocation site. We use the following expansion rules:

- $var1$ is a critical variable if it is annotated manually or identified as such by dataflow analysis. For example, using $\_\_attribute\_\_((annotate(\text{“}critical\text{”})))$ in the source code.
- $p = \&var1$, then any dereference of the pointer p is a critical variable. For example, $p = \&var2$, then $var2$ is a critical variable. A pointer pointing to a critical variable is critical, and any variable written to the pointer also becomes a critical variable itself.
- $var3 = var1$, then $var3$ is a critical variable.
- $var4.member = var1$, then the structure variable $var4$ is a critical variable, and all the member of $var4$ are also regarded as critical variables.

The process of expanding critical variables runs recursively as long as it finds new critical variables. Due to the limitations of the data watchpoint mechanism, what we can obtain at runtime is just a specific memory address (*NOT* equal to the specific critical variable). Therefore, we need a mapping relationship between critical variables and memory allocations. However, a memory allocation could lead to many variables in the LLVM **Intermediate Representation (IR)** code. This is because the LLVM IR uses single static assignment format to represent instructions, where each variable is assigned only once. For example, a piece of source code is shown in Listing 1.

```
int a[2];
a[0] = 1;
a[1] = 2;
```

Listing 1. Example of Source Code

After transforming it to the LLVM IR codes, it can be shown in Listing 2.

If we annotate the variable “a” as critical, then three critical variables can be expanded through dataflow analysis, which are respectively {%a, %1, %2}. However, only %a is actually allocated in memory, and the other two variables %1 and %2 share the same memory with the variable %a. Therefore, we can monitor all three variables as long as we monitor the memory of variable %a. We regard memory-allocated critical variables as the *root* variables. Based on the dataflow dependency

```
%a = alloca[2*i32], align4
%1 = getelementptr inbounds[2*i32], [2*i32]*%a, i64 0, i64
store i32 0, i32*%1, align4
%2 = getelementptr inbounds[2*i32], [2*i32]*%a, i64 0, i64
store i32 1, i32*%2, align4
```

Listing 2. Example of Generated IR Codes

analysis, we can map all critical variables in the expanded set to the root variables. Note that one critical variable could be mapped to at least one root variable. For example, a critical variable is the parameter $*dst$ of the $memcpy()$ function, could map to multiple *root* variables considering the different calling contexts. In the end, the compiler has determined the full expanded set of critical variables and their memory allocations. The process only requires the programmer to focus on a small number of critical variables, reducing the difficulty and effort of the manual work, and making it less error-prone.

4.2.2 Legal Tuples Extraction. The key insight of our approach is that critical variables can only be modified by a "legal" write instructions. For each critical variable, we use dataflow analysis to extract a list of tuples (A, B) to represent legitimate operations. Inside the tuple, A represents the identity of the target variable (each critical variable has its own identify after analysis), and B represents the address of the *STR* instruction. Using the LLVM framework, we can obtain a list of tuples $(criticalVar, storeInst)$ for each critical variable. All memory operations on variables are ultimately reflected on *root* variables. Therefore, combined with the mapping relationship between critical variables and the root variables, our compiler records legal store instructions for all root variables. We represent it as $\{rootVar : storeInsList\}$. Considering the example given in Listing 2, we could obtain the operations on root variables as $\{\%a : [ins3, ins5]\}$, or get two tuples as $(\%a, ins3)$ and $(\%a, ins5)$. Although the obtained instructions are represented as IR format, we further get its corresponding "legal" machine instruction address by doing an analysis on the LLVM backend and obtaining its offset in a function.

4.2.3 Critical Variables Remapping. We configure the DWT mechanism to monitor a continuous memory space, where memory access initiated by store instructions (e.g., *STR* instructions) will trigger debug monitor exception. In the exception, we can perform the legitimacy check. To make this happen, we need to remap the dispersed memory allocations of the critical variables into the continuous monitored memory region. There are three types of critical variables to remap: global, stack, and heap variables. Specifically, we first reserve and configure a contiguous memory space whose starting address is $base\_addr$, the first M bytes are used to manage global variables, the next N bytes are used to manage stack variables, and the last K bytes are used to manage heap variables. The values of M, N, and K can be adjusted according to the requirements of the specific applications.

Critical Global Variables. We sequentially replace global memory allocations to move them to the monitored memory space and maintain 4-byte alignment. Meanwhile, we also record the size of each variable and the offset relative to the starting address $base\_addr$. Then, each global critical variable is recorded as $\langle varID : \{base\_addr, offset\}\rangle$. In ARM Cortex-M series devices, there is no **Address Space Layout Randomization (ASLR)** mechanism. Therefore, given a specific address at runtime, it is easy to calculate the offset relative to the $base\_addr$. This helps to identify which critical variable (*varID*) is accessed.

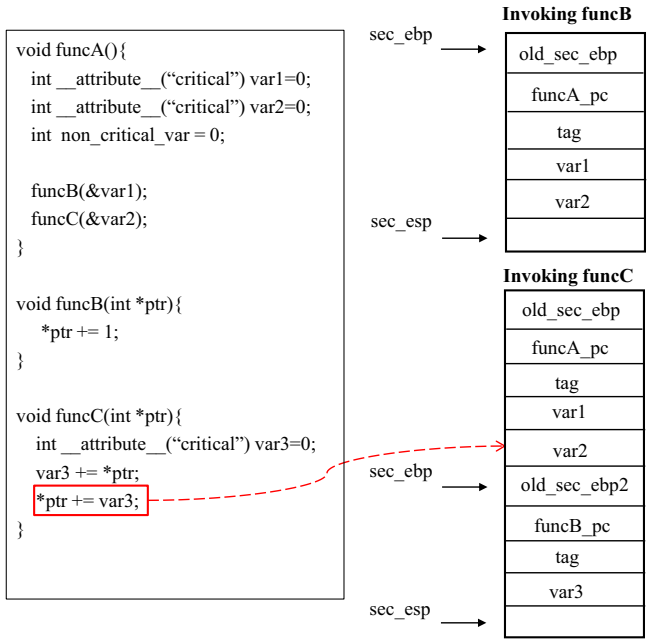

Fig. 2. Remapping stack variables to the secure stack. The secure stack only grows the function defines critical stack variables.

Critical Variables on the Stack. Stack allocations of critical variables are replaced by allocations in a memory block in the monitored memory. To identify and distinguish different stack variables, we build a secure stack in the monitored memory (see Figure 2). The secure stack is managed by a secure frame pointer (*sec_ebp*) and a secure stack pointer (*sec_esp*). To manage critical stack variables used in the function, we first save the *old_sec_ebp*, the function address $func\_pc$, and a *tag* on the secure stack. Specifically, critical stack variables are recorded as $\langle varID : \{func\_pc, offset\}\rangle$ after static analysis.

When a monitored stack variable is accessed at runtime, we need to identify which variable it refers to (i.e., its *varID*). However, the varID is not directly provided—instead, we can only get the accessed memory address at runtime. Therefore, we need to map the memory address to the corresponding *varID*. To achieve this mapping, we introduce a secure stack as shown in Figure 2. When a critical stack variable is accessed at runtime, and its address (*addr*) is obtained, we can determine its varID by obtaining its function and the offset. Specifically, we use the following three steps.

(1) If *addr* is greater than $base\_addr + M$, we know it refers to a stack variable. We then check if *addr* is greater than *sec_ebp*.
(2) If yes, we use $sec\_ebp + 4$ to obtain its function address (*func_addr*) and $addr - ebp - 12$ to get the *offset*.
(3) If no, we search for the "tag" field (a magic number) starting from *addr*. Once found, the corresponding *old_sec_ebp* can be easily located because the *offset* between "tag" and *old_sec_ebp* is fixed as 8. We then obtain *func_addr* and *offset* like in step 2. However, if an attacker can control the input and control the value of "tag," or the normal program input value equals the value of "tag," these situations may interfere with our analysis and cause errors in positioning

variables. Our design prevents this attack by two steps: (1) Since each write to the monitored memory will trigger an interrupt, in the interrupt logic, we extract the current write value and determine whether it is "tag." If so, we label the memory address in a global array. (2) During the "tag" search, if the value of the address is found to be "tag," we will compare whether the address has been marked in the global array. If so, we skip it and keep searching "tag." In fact, the probability that the value of the program equals the value of "tag" is very small, and in our experiment, we did not encounter such situation. Fortunately, even if we do encounter this situation, we can handle it well.

For example, the statement $*ptr+ = var3$ in $funcC$ marked by the red circle accesses a local variable at *offset* 4 of $funcA$. By running our prototype, we can correctly identify all its metadata ($funcA, offset$). Finally, we can match the obtained metadata to the *varID* based on the statically obtained record (i.e., $\langle varID : \{func\_pc, offset\}\rangle$).

Moreover, the design of the secure stack is memory-efficient in the sense that it does not use (much) more memory than the baseline. Only when there is a critical stack variable memory allocation in the function will it increase the secure stack. As shown in Figure 2, when function $funcB$ is called, the secure stack does not grow. This is because that the $funcB$ does not define any new critical stack variables, even though it manipulates the critical variable $var1$. Only when the function defines the critical stack variables the secure stack will grow. Note that when the critical variable is allocated in the secure stack, there is no need to allocate the memory on the original stack. When the function returns, the secure stack will also restore the secure stack frame.

Critical Variables on the Heap. We also remap the critical variables on the heap to the monitored addresses. When the program allocates an n-byte heap variable, we call a wrapper function for heap allocation and actually allocate $n + 4$ bytes. The additional 4 bytes are used to tag the identity of the variable. Then, we represent the heap variable as $\langle varID : tagID\rangle$ after static analysis (*tagID* is specific to the allocation site, which is generated after static analysis). It is easy to identify a heap variable access in the runtime. When the address of the variable is greater than the value of $base\_addr + M + N$, we can search the heap *tag* in the memory to obtain its identity.

4.2.4 Obtaining Runtime Tuples. The DWT feature supported by the ARMv8-M series devices can be used to monitor a continuous range of memory, and any writes to the monitored memory would trigger the execution of the interrupt handler, i.e., DFI_Validation_Module. In DFI_Validation_Module, we can obtain the stacked execution context through the *SP* register. The stacked context includes R0-R4, R12, R14, xPSR registers, and the *return address*. However, based on the stacked execution context, there are still two challenges in obtaining the exact tuple (A, B): the exact address of *STR* instruction and the exact address of memory accessed.

❶ The DWT interrupt is asynchronous, causing a delay between a memory access event and the execution context saving. For the following example, we assume the *STR* instruction at 0×04 triggered an interrupt. The hardware stacks the execution context and jumps into the DFI_Validation_Module. The *return address* obtained in the DFI_Validation_Module may not be 0×08. Instead, it could be $0 \times 0a$ or $0 \times 0c$ due to the delay. It poses a challenge to obtain the exact instruction that triggers the interrupt. Especially if there is a branch instruction, the trapped *return address* could be far away from the memory-accessing *STR* instruction. Based on our observation, there is a delay of at most two instructions. This also agrees with the pipeline design of the Cortex-M core; the manual [8] states explicitly that the maximum delay is two instructions (in Section 4.7). As such, the address of the fault write instruction is within three instructions backwards along the execution trace from the PC value obtained from the exception stack frame.

We address this challenge by injecting two NOP instructions (i.e., $(MOVR0, R0)$) after every critical write instruction. As a result, the secure runtime only needs to linearly search for the

```
0x04 STR R0, [MEM_ADDR]
0x08 MOV R0, R1
0x0a SUB R0, 4
0x0c LDR R1, [MEM_ADDR]
0x0e BL Function_XXX
```

Listing 3. Instruction delay in DWT trigger.

nearest write instruction (e.g., *STR*) backwards from the address obtained from the exception stack frame. Finally, the instruction address obtained can represent the original *STR* instruction. This can also help to detect illegal critical variable modification from non-critical stores, because if we do not find the NOP instruction by forward searching the *STR* instruction, we then know it is an illegal modification from non-critical stores.

❷ It is challenging to infer the accessed memory address *only* with the stacked context. Currently, the hardware does not support to explicitly provide the exact accessed memory address. As a solution, we instrument before the critical *STR* instruction that writes the critical variable. We first obtain its targeted memory address and save it in a TEE protected memory called *CriticalAddr*. Then, the address can be directly obtained in DFI_Validation_Module. Note that we do not instrument before all *STR* instructions. Instead, we *only* instrument the *define site* (we regard the instruction that modifies the critical variables as define site). There are two benefits: (1) Only relatively few define sites (*STR* instructions) need instrumentation, making it lightweight. (2) Transition to secure world is performed only when necessary, if no critical variables involved, there is no overhead.

4.3 Bitmap-Based Allowlist Construction and Storage

To accommodate the limited hardware resources on embedded devices, we propose a space efficient storage mechanism of legitimate operations (e.g., indirect branch, data store, and so on). This mechanism compresses and records legitimate operations into a bitmap structure, which we call allowlist. First, to extract the allowed tuples (A, B), we perform offline analysis to capture (1) the source address and destination address tuples for each legitimate indirect branch (i.e., forward indirect branch and backward indirect return) in the control-flow graph and (2) the *STR* instruction address and its allowable write targets tuples for each legitimate operation on critical variables. Then, each pair is mapped to a single bit in the allowlist using an efficient hash algorithm based on Cantor Pairing [17]. Finally, the complete allowlist of the program will be stored in the TEE, e.g., the ARM's TrustZone, to ensure confidentiality and integrity.

As a result, the protected program can generate the aforementioned tuple at runtime and jump into the corresponding validation module. Specifically, to enforce CFI, we instrument a secure function call (via SG) before indirect branch instructions in the normal world. The function takes two parameters, one is the current instruction address, the other is its destination address. It will then switch to the secure world, transferring the tuple generated in the normal world for further validation. To enforce DFI, we first configure the SDME bit of DEMCR register [52] so that the exception can be directly delivered to the secure-world handler. Specifically, the normal world debug monitor events (i.e., critical variable writes) are processed by the secure world handler DFI_Validation_Module. Then, in the validation module, it automatically extracts the tuple (i.e., the responsible `STR` instruction and its write target variable ID), which is further converted to the allowlist index for further validation. Finally, we check the index against the allowlist. If there is a record, then the operation (indirect control-flow jump or data store) is legal; otherwise, the operation is illegal.

To construct and verify against the allowlist, we take the Cantor Pairing algorithm as the initial step within our hashing mechanism. The Cantor Pairing algorithm is simple and relatively effective compared with traditional hash algorithms. It only contains two multiplications and three additions. However, it can uniquely map a pair of natural numbers (A, B) to another natural number N ($N = CantorPairing(A, B)$). To map the index within a specific range, it is required to do a modular operation on the mapped index (i.e., $index = N\%AllowList\_Size$). The simplicity of the algorithm can significantly reduce the runtime overhead, thereby reducing the impact on the normal function of the program.

Preventing Collisions in the Bitmap. Collisions are common in hash algorithms. If a collision happens, an invalid control-flow transition or memory access might result in an address tuple, which generates a valid bit in the bitmap when hashed. Attackers can potentially take advantage of the hash conflict to evade detection. It is clear that the larger the bitmap is, the less the chance of conflict would be. However, we have limited memory available and cannot grow the bitmap by much. Therefore, we do offline analysis to prevent collisions.

For potential collisions caused by a CFI attack, we regard all the addresses of *POP* {*.., PC*} *or POP* {*.., LR*} instructions and indirect calls in the program as possible A values and took all the addresses of functions useful in attacks and ROP gadgets as possible B values. Then, we compute the Cartesian product on these two sets to generate a series of illegal tuples (A, B). For potential collisions caused by DFI, consider an example: $memcpy(dst, src, n)$, where dst is the memory of critical variable, and the n is the dynamic size. Now suppose that due to a program bug, the value of n exceeds the size of dst and reaches another critical variable B. If an index collision exists in this situation, the critical variable B may be modified illegally without detection. To avoid the collision, for each critical variable, we then use all other critical store instructions (not allowed to write the chosen variable) to generate the illegal tuples (A', B').

For each element in the illegal tuples of both CFI and DFI, we calculate the illegal index with the same hash function. Finally, we check these illegal indexes with the record in the allowlist. If there exists a collision, we increase the bitmap size, regenerate the allowlist, and do the previous step again. Otherwise, the allowlist is collision-free for CFI and DFI.

4.4 Runtime Validation

After obtaining the legal tuples with the static analysis (from CFI and DFI), the next step involves encoding and storing them in a bitmap-based allowlist. This process entails applying a hash operation to the address tuples and mapping each tuple to an index in the allowlist, setting the corresponding bits to 1. During runtime, any security-sensitive operations, such as indirect branch transfers or writes to critical variables, will trigger legitimacy checks within the TEE. Within the TEE, we extract the tuple (A, B) either directly from the parameters of secure API (e.g., in the case of CFI) or infer it from the stacked context (e.g., in the case of DFI). Subsequently, we map the tuple to a bit using the same hash calculation and verify it against the constructed bitmap. If the corresponding bit is set to 1 in the allowlist, the transfer is considered legal. Conversely, if the bit is not set, the transfer or memory access is unexpected. We regard this as a potential attack and generate an alert. We also record the execution context of the stack and the registers, which can be used for offline analysis to locate the root cause. The user can also configure what to do depending on the specific scenario, including aborting the program, dropping privileges, or just recording the alert.

5 Implementation

We have implemented a prototype of *BSM* based on LLVM 10.0 [49] and OAT [70]. At a high level, *BSM* can be divided into four components: (i) Static analyzer: For CFI, we extract the function call graph and conduct a type-based indirect call target analysis. For DFI, we conduct a dataflow analysis to extract a set of critical variables. (ii) Allowlist extractor: We analyze the call graph and iterate the operation of critical variables to obtain legal tuples. Then, we map these tuples to form an allowlist. (iii) Instrumentation: We instrument critical instructions (insecure control-flow transfers) and remap critical variables to a contiguous monitored memory. (iv) Runtime validator: We obtain the runtime tuple, then calculate the index (hash) of the tuple, finally look up the hash value in the allowlist, and check whether the operation is legal.

5.1 Static Analyzer

For CFI, we first obtain the call graph through an LLVM pass. It identifies all possible targets for each function call statement. For each function, it generates a list to store callers and callees. For direct calls, we can directly determine the relationship between caller and callee; for indirect invocation, the possible target of the indirect invocation statement is determined through type-based analysis. That is, we assume that any type-matched function is a legal target. Therefore, we generate an over-approximated call graph.

For DFI, we first need to specify the initial set of critical variables using annotations as follows:

$$int\_\_attribute\_\_((annotate(\text{“}criticalg\text{”})))var = 1,$$

We start from these annotations and then automatically propagate and expand the set of critical variables.

5.2 Allowlist Extractor

For CFI, we extract the tuples from each call site and called function based on constructed call graph. Recall that building the call graph involves type-based alias analysis. Then, we transform the tuples with the Cantor Pairing function to form an allowlist, where each tuple will have a record in it.

For DFI, each critical variable is mapped to a varID, and we extract all the *define* instructions of the critical variable in the IR layer. Then, we locate the corresponding machine instructions (different from IR instructions) with a Machine IR layer pass. Specifically, we get the related machine instruction offset of the function. Finally, the varID and the address of *define/store* instruction are collected to form an allowlist. The results of this step make up a unified allowlist, which contains legal critical instruction transfers and legal critical variable operations.

5.3 Instrumentation

For CFI, we instrument the critical transfer instructions. The instrumentation is implemented in an LLVM backend pass. It can be divided into three steps: (1) collect all potential attack targets (i.e., indirect calls/jumps and function returns) in all functions, (2) instrument a trampoline (secure API call via SG) before each indirect call/jump and function return instruction, and (3) pass the source and destination address information as arguments to the SG function call, which would be directly used in the TrustZone. The *BSM* iterates through each instruction. Each time it encounters an indirect call or function return statement, it first backups registers R0 and R1 (because these registers are used to pass arguments). Next, pass the source address to R0 and the destination address (i.e., the return address on the stack or Rx in *BLX Rx* instruction) to R1 as the second argument. Then, call the SG function to switch to the TrustZone.

For DFI, we first annotate the critical variables manually and then use a dataflow analysis to obtain an expanded set of critical variables. Then, we remap all critical variables (stack, heap, and global variables) to a contiguous memory area configured with DWT. Besides, we instrument a trampoline function to obtain its memory address for each definition of critical variables. After the instrumentation step, we are able to capture the tuple (A, B) during the execution of the application.

5.4 Runtime Validator

For CFI, once the instrumented trampoline is called, the program switches to the secure world. In the secure world, it applies a Cantor Pairing algorithm with two parameters (representing the source and the destination) to calculate a hash (index) of the bitmap. Then, the validator module checks whether the allowlist has the record. For DFI, at runtime, any write to a critical variable would trigger a hardware interrupt event. Then, it leads to the execution of DebugMon_handler, which obtains the corresponding triggered instruction based on the saved execution context. Meanwhile, the value of the accessed memory is then directly obtained in DebugMon_handler if it is an critical store. Finally, we can get a tuple (A, B), representing the accessed memory address and the corresponding instruction. As a result, based on the obtained tuple, *BSM* can check its legitimacy with the prepared allowlist.

6 Evaluation

To evaluate our *BSM* mechanism, we chose the STM32L562E-DK, a widely used embedded development board, as our reference device. This board is representative of the systems we aim to protect. The STM32L562E-DK Discovery Kit is a complete demonstration and development platform for ARM Cortex-M33 with ARM TrustZone and ARMv8-M mainline security extension core-based microcontroller, with 512 kB of Flash memory and 256 kB of SRAM. Our evaluation aims to answer the following questions:

- —Q1: What is the performance overhead of *BSM*?
- —Q2: How does the performance overhead of *BSM* compare to the state-of-the-art solution — OAT?
- —Q3: What is the memory overhead of *BSM*?
- —Q4: What is the power consumption overhead of *BSM*?
- —Q5: What are the security benefits of *BSM*?

6.1 Firmware Samples

There are currently no bare-metal benchmark programs that have annotated semantically critical variables. Additionally, the device-specific nature of embedded programs necessitates substantial manual effort to port each program to a particular development board, such as configuring hardware interfaces. Note that this effort is not due to *BSM*; once the firmware runs on a particular board, the manual effort for implementing *BSM* is minor. Therefore, we selected some embedded programs with practical application scenarios for evaluation. For comparative consistency, we would have liked to migrate all five evaluation programs of OAT [70], because it is most similar to what we do. Despite a non-trivial manual effort, we were able to successfully port only two programs (namely Syringe Pump and Light Controller). The other three programs failed to be ported because they used *socket* API for network communication, while our device does not support the relevant API.

To increase the number of test objects to make the evaluation more comprehensive, we also tried to port test programs used as benchmark from other work [4, 6, 19, 25, 31, 81], and we successfully ported PinLock, PID Controller, and Steering Controller. In addition, we included

Table 1. Critical Variables (CVs) in Programs

Program	# Initial CVs	# Total CVs	Detailed List
Syringe Pump	11	19	mLBolus, mLBigBolus, mLBolusStepIdx, mLBolusStep,serialStr, serialStrLen, mLUsed, cmd, authenticated, motorDirPin, motorStepPin
Light Controller	7	11	method_start, method, memory_added_pattern, device_routing, data, new_item, switch_pattern
PID Controller	4	8	Temp, TPID, PIDOut, TempSetpoint
PinLock	5	8	key_in, key, failures, unlock_count, lock_status
Steering Controller	8	12	inSteer, inMotor, inMode, motorPin, steerPin, ledPin, command, LEDstatus
CRC_Data	5	6	buffer_size, CRC_POLYNOMIAL_16B, CRC_INIT_VALUE, uwCRCValue, uwExpectedCRCValue
RNG_MultiRNG	2	2	aRandom32bit, Counter
CRYP	4	10	AES_TEXT_Size, TIMEOUT_VALUE, AESIV, AESIV_CTR
OSPI	4	6	aTxBuffer, CmdCplt, address, step
RTC_Alarm	3	6	aShowTime, RTCStatus, stimestructureget

five more demo applications (namely **Cyclic Redundancy Check (CRC)**_Data, **Random Numbers Generator (RNG)**_MultiRNG, CRYP, OSPI, RTC_Alarm) from the official STM32CubeL5 repository [63]. Using demo applications for evaluation is common practice in related research in embedded devices [19, 20, 82] due to the limited availability of open source firmware to test with. In total, we evaluated BSM on 10 different test programs. These programs reflect the small size that is typical in real-life application scenarios, such as ICSs and smart homes. The business logic they provide includes semantically critical variables, as well as program control-flow transfers, which is appropriate to validate our prototype. Their source code uses common C/C++ programming features (e.g., arrays, aliases, pointers, indirect jumps, and so on) and also some complex algorithms such as SHA256. We compiled all the test programs using the default link time optimization options.

In the following, we provide detailed introductions to these 10 programs. We also present the number of critical variables we annotated, the total number of critical variables, and a detailed list of variable names annotated for each firmware in Table 1.

— *Syringe Pump* [70] used in the medical and production fields. It can automatically perform liquid injection without manual intervention, thus reducing manual burden. We manually annotated 11 critical variables related to authentication, motor status, and injection amount. Since they are critical to the injection action and the amount of injection. They were extended to 19 critical variables automatically by static analysis.
— *LightController* [70] used in smart home. The user can switch a light on or off by sending a command. We manually annotated seven critical variables related to operation commands, such as on–off action and mode switching action. They were extended to 11 critical variables automatically by static analysis.
— *PID Controller* [25] used in the industrial production field. It can adjust the input according to the historical data and expected value, so that the system is more accurate and stable. We manually annotated four critical variables related to historical data, which are useful for guiding input to the program. They were extended to eight critical variables automatically by static analysis.
— *PinLock* [19] used in the smart home. For example, it can be applied to the door lock. The door can be unlocked only after entering the correct PIN code. We manually annotated

five critical variables related to user authentication and lock status, which are used for controlling the lock. They were extended to eight critical variables automatically by static analysis.

— *Steering Controller* [31] used in autonomous driving. It receives commands from the computer to control the steering and moving/motoring of the autonomous vehicle. We manually annotated 8 critical variables related to vehicle control, which were extended to 12 critical variables automatically by static analysis.

— *CRC_Data* [58] used to calculate a CRC code based on HAL APIs (e.g., CRC peripheral configurations). It involves computing a 16-bit long CRC code, which is stored in the variable *uwCRCValue*. The computed CRC code is then compared to the expected CRC value, stored in the variable *uwExpectedCRCValue_reversed*. We manually annotated five critical variables related to CRC value and its configurations, which were extended to six critical variables automatically by static analysis.

— *RNG_MultiRNG* [61] uses an RNG to generate 32-bit long random numbers based on **Hardware Abstract Layer (HAL)** APIs. It uses an 8-entry array *aRandom32bit*, which will be filled up with 32-bit random numbers when a button is pressed by a user. The random numbers can be displayed on the debugger in the *aRandom32bit* variable. We manually annotated two critical variables associated with the storage of random numbers. No additional variables were identified through static analysis in this program.

— *CRYP* [59] uses the cryptographic peripheral to encrypt and decrypt input data using **Advanced Encryption Standard (AES)** in chaining modes. We manually annotated 4 critical variables related to encryption and decryption operations, which were extended to 10 critical variables automatically by static analysis.

— *OSPI* [60] used to demonstrate how to erase a part of an OSPI NOR memory, write data in memory-mapped mode and access to OSPI NOR memory in memory-mapped mode to check the data in an infinite loop. We manually annotated four critical variables related to steps of data processing, which were extended to six critical variables automatically by static analysis.

— *RTC_Alarm* [62] used to configure and generate an RTC alarm using the RTC HAL APIs, and the current time is updated and displayed on the debugger in the *aShowTime* variable. We manually annotated three critical variables related to time status, which were extended to six critical variables automatically by static analysis.

6.2 Performance Overhead

To ensure the deployability of defense mechanisms, it is crucial to make them lightweight [72]. This is especially important for MCUs, which are often equipped with limited resources. We measured the execution time for each test program in the two settings: baseline and BSM (i.e., CFI and DFI both enabled). We start the runtime measurement at the beginning of the main function and stop it when the application exits. The results are shown in the Table 2. The exact trigger counts of CFI validation and DFI validation are shown in the sub-columns "#CFI" and "#DFI." The sub-column "Overhead" shows the relative delay caused by *BSM* to the program executions. The geometric mean is 2.09% when CFI and DFI are both enabled. We can observe that the delay is acceptable even for programs running on resource-constrained embedded devices. Based on our observations, the number of DFI validations is much lower compared to the number of CFI checks. This can be attributed to the fact that (1) the legitimacy check for DFI is exclusively performed during *actual* critical data accesses, eliminating the need for instrumentation and legitimacy checks on all potentially risky write instructions; and (2) the legitimacy check for CFI is conducted dynamically at runtime for every forward indirect jump and backward function return, resulting in frequent checks.

Table 2. Performance Overhead

Program	#CFI	#DFI	Baseline Execution Time (ms)	CFI and DFI Enabled Execution Time (ms)	Overhead (%)
Syringe Pump	45,118	16	15,300	15,311	0.07
Light Controller	7,481	359	196	200	2.04
PID Controller	10,149	315	4,007	4,016	0.22
PinLock	3,347	786	69	74	7.25
Steering Controller	10,026	10	9,661	9,692	0.32
CRC_Data	4,411	7	445	455	2.25
RNG_MultiRNG	2,656	9	265	269	1.51
CRYP	39,823	42	7,879	8,038	2.02
OSPI	1,272	767	123	127	3.25
RTC_Alarm	4,979	6	410	419	2.20
Geometric Mean	7,130	53	907	926	2.09

In addition, we compare the performance overhead between OAT and *BSM*, as depicted in Figure 3. We applied both OAT and *BSM* to each individual application. *BSM* performs online legitimacy checks at indirect branches and critical data writes, whereas OAT relies on a backend server for remote verification of operation legitimacy. It is important to note that, for OAT, the remote server does not use the computation resources of the embedded device. Consequently, the overhead introduced by the remote server component is not taken into account. Even so, *BSM* reduces overhead compared to OAT, with an geometric mean of 2.09% in *BSM* versus 4.19% in OAT, while offering immediate detection, and therefore stronger security. There are several factors contributing to this improvement. Firstly, OAT must capture and record all branches to reconstruct control-flow execution traces remotely and conduct legitimacy checks within a program, whereas the number of branches (i.e., only indirect jumps) tracked in *BSM* is considerably smaller. Secondly, OAT requires checks to be performed at both read and write sites of critical variables, whereas *BSM* only performs legitimacy checks at write sites because it uses the DWT mechanism to implicitly detect reads to critical variables. Additionally, it is worth noting that both *BSM* and OAT incur the highest overhead on the PinLock project. This can be attributed to the complexity of the employed hash function, specifically SHA256. This function frequently writes to a critical variable (i.e., the memory holds the output hash value), necessitating the inclusion of a corresponding legitimacy check.

6.3 Memory Overhead

Memory overhead is measured in two parts: RAM overhead and FLASH overhead. RAM overhead refers to the extra memory needed to store the program state at runtime, while FLASH overhead refers to the space needed to store the instrumented program itself. The result can be seen in Table 3.

RAM Overhead. To enforce the security schemes on the embedded device, *BSM* requires additional RAM resources. The additional RAM overhead consists of two parts:

— *Allowlist. BSM* needs to deploy a bitmap in the program to store legal program behaviors, which are generated by static analysis. Each legal tuple (A and B) maps to a bit in the allowlist to represent a valid memory access (write_instruction, critical_variable) or a valid control-flow transfer (indirect_source_instruction, destination_instruction). In our experiments, the default allowlist size is set to 1 kB, which can represent 8,192 different legal behaviors. To avoid

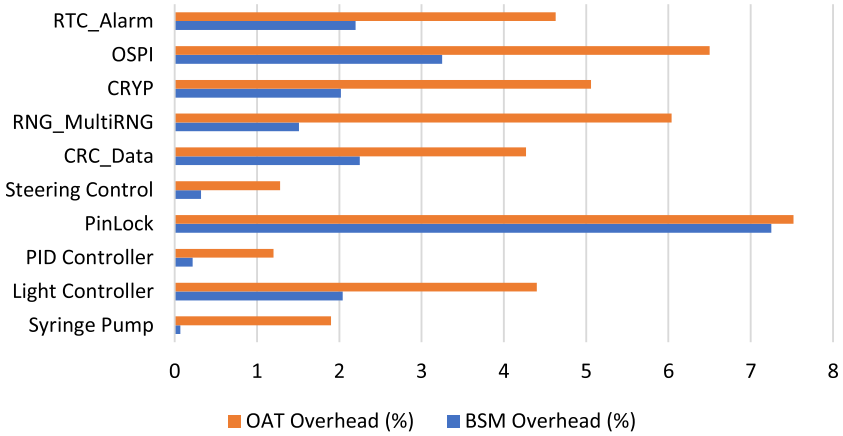

Fig. 3. Comparison of runtime overhead for OAT and *BSM*.

collisions, we adjusted the size of the allowlist for each program by increasing the size until no collisions were found.

— *Monitored memory.* *BSM* requires a piece of continuous memory to monitor the memory access of all critical variables. Essentially, our design does not incur the cost of critical variables in terms of memory overhead, because these variables in the original program will also occupy the corresponding memory. After memory remapping, the original memory allocation is no longer needed. However, we still include it in the memory overhead for its special usage.

The memory cost includes the size of all critical variables in the given application. We reserve 500 bytes by default because it is enough to handle the programs in our experiments. Our memory consumption is about 1.53 kB in total, occupying 0.60% of the total RAM (256 kB). On the other hand, compared with the existing method proposed by OAT (the whole memory footprint of the measurement engine is less than 10 kB for the real embedded applications used in the evaluation [70]), our method reduces runtime memory overhead by 84.7%, i.e., OAT costs 6.5 times as much memory as our solution. The key factor leading to the distinction is the choice of data structure employed by OAT and *BSM* for recording the allowlist. OAT utilizes a sparse hash map to store the allowlist, with each item occupying 4 bytes and indexed by the variable address. On the other hand, *BSM* adopts a compact bitmap representation, where each legal item is represented by a single bit.

FLASH Overhead. The increase in binary size caused by *BSM* instrumentation is shown in Table 3. It can be seen from the table that the binary size increases only 10.79% on average. The following four categories of instrumentation increase the code size:

— We instrument a trampoline function before control-flow transfers, to jump to CFI_Validation_Module.
— We instrument instructions to obtain the CriticalAddr before the *STR* instruction (define-sites) of critical variables.
— We instrument NOP ($MOV R0, R0$) instructions between the *STR* instruction (define-site of critical variable) and the Branch instruction, to help obtain the exact trigger instruction of the DWT event.
— The instructions that maintain the secure stack. These instructions will only be instrumented in the functions where critical stack variables are defined.

Table 3. Memory Overhead

Program	Baseline		CFI and DFI Enabled		Overhead	
	RAM (kB)	FLASH (kB)	RAM (kB)	FLASH (kB)	RAM (%)	FLASH (%)
Syringe Pump	4.20	33.07	5.89	36.67	40.24	10.89
Light Controller	4.72	19.42	6.22	21.38	31.78	10.09
PID Controller	5.88	21.53	7.39	24.04	25.68	11.66
PinLock	3.34	23.64	4.84	26.01	44.91	10.03
Steering Controller	4.61	20.63	6.12	23.01	32.75	11.54
CRC_Data	4.85	31.28	6.35	34.72	30.93	11.00
RNG_MultiRNG	4.89	31.90	6.40	35.12	30.88	10.09
CRYP	5.65	43.20	7.16	47.86	26.73	10.79
OSPI	5.15	35.61	6.68	39.48	29.71	10.87
RTC_Alarm	4.86	34.79	6.38	38.57	31.28	10.87
Geometric Mean	4.76	28.55	6.31	31.63	32.56	10.79

Table 4. Power Consumption Overhead

Program	Baseline (mW)	CFI and DFI Enabled (mW)	Overhead (%)
Syringe Pump	771.4	776.5	0.66
Light Controller	824.2	832.5	1.01
PID Controller	841.3	842.7	0.17
PinLock	778.9	782.1	0.41
Steering Controller	782.6	789.1	0.83
CRC_Data	713.4	718.2	0.67
RNG_MultiRNG	725.7	730.2	0.62
CRYP	737.4	741.9	0.61
OSPI	741.2	744.5	0.45
RTC_Alarm	726.9	730.4	0.48
Geometric Mean	763.2	767.7	0.59

In addition, the compiler will remap critical variables to the monitored memory, but this part only changes the operands of related instructions without adding any codes.

6.4 Power Consumption Overhead

In this section, we evaluate the power consumption overhead of our prototype. It is important to note that embedded devices are often designed to perform their tasks with low power consumption, as they may rely on battery power or other limited power sources. For example, smoke alarm devices are typically mounted on walls or ceilings and need to continuously perform their tasks while being on standby for a long time. Consequently, power is a highly precious resource for these devices, and any modifications must carefully consider its impact.

Power consumption would be influenced by various factors, including CPU frequency, workload, and connected peripheral devices, and so on [66]. In our evaluation, the primary source of changes in power consumption is the workload. We made efforts to keep other factors unchanged, such as maintaining the default CPU clock frequency of 110 MHz. Additionally, our development board, the STM32L562E-DK, is equipped with an "onboard energy meter" USB interface [9], which allows for convenient measurement of power consumption. On this basis, Table 4 shows the power

Table 5. Security Validation Testing

Vulnerability	Exploitation		Attack Prevention	
	No	Description	Unlock?	Attack Detected?
vul-1: Stack memory overflow	#1	Overwrite a local variable	No	No
	#2	Overwrite return address	No	Yes
vul-2: Global memory overflow	#3	Overwrite the pin hash (variable *key*)	No	Yes
vul-3: Arbitrary memory writes	#4	Modify return address	No	Yes
	#5	Modify lock_status variable	No	Yes
	#6	Modify DWT configurations	No	Yes

consumption of our prototype system on ten embedded programs. *BSM* is designed to focus on protecting only a limited set of critical variables, and the DFI validation would be triggered only when the monitored memory is accessed. Moreover, the hash function adopted in *BSM* is straightforward, which suppresses the increase in power consumption. The power consumption overhead associated with *BSM* is relatively small, with an geometric mean of 0.59%, translating to a negligible energy requirement for embedded systems, which is consistent with some existing work [71] in the field. This is likely due to the fact that our modifications do not entail an increase in CPU frequency, nor do they include high-frequency intensive computational processes (e.g., encryption/decryption), and they also avoid the utilization of excessive memory.

6.5 Attack Prevention

To show the security benefits of *BSM*, a common approach is to conduct an attack prevention test. However, there is no benchmark specifically designed for evaluating security issues associated with deeply embedded devices using bug and exploitation measures. Consequently, we align with the existing research [4, 6, 19, 82], which employs injected vulnerabilities and corresponding exploits for similar evaluation purposes. We also note that the target program is protected using the data execution prevention mechanism, meaning that data regions like stack or global are set non-executable based on MPU settings, but it does not have ASLR protection. To this end, we deliberately injected three vulnerabilities into the function *rx_from_uart* of PinLock project, introduced additional temporary variables, and crafted a number of related exploits. The final goal of these attacks is to execute the *unlock*() function without entering the correct pin.

The attack settings and results can be seen at Table 5, and the three vulnerabilities and six exploits are outlined below:

— *vul-1: Stack memory overflow.* Attackers can overwrite a local variable (Exploit No#1), and they can also overwrite return address of the function *rx_from_uart* with a value of their own choice—*unlock* function pointer (Exploit No#2).
— *vul-2: Global memory overflow.* Attackers can use this vulnerability to overwrite an important global variable (i.e., the variable *key* that holds the PIN's hash and is used in PIN code check) with a malicious value (Exploit No #3).
— *vul-3: Arbitrary memory writes.* Attackers can write the desired memory directly by inputting an address and the corresponding content. Specifically, these attackers can directly modify the function pointer of *rx_from_uart* (Exploit No#4), the DWT configurations (Exploit No#5), and the variable *lock_status* (Exploit No#6), respectively.

According to the table, none of the exploits successfully achieved the objective of executing the *unlock*() function. It is worth noting that the exploit No #1 went undetected by *BSM*. Although it tampered with a specific local temporary variable, it did not alter the control-flow graph (CFG)

or modify the critical variables that are monitored and protected by *BSM*. Consequently, the exploit remained undetected but failed to execute the *unlock*() function. Exploits No #2 and No #4 were detected by *BSM*. This is because, before the vulnerable function *rx_from_uart* returns, the instrumented trampoline function sends the tuple (A, B) (where B is the *unlock*() function pointer) to the secure world. As the control-flow transfer is not present in the allowlist, it is promptly identified and rejected by the CFI_Validation_Module. Exploits No #3 and No #5 were also detected by *BSM*. The variables *key* and *lock_status* are monitored by DWT mechanism. When these variables are modified, a debug event occurs, triggering the DebugMon_handler to analyze the stacked context and verify its legitimacy. Due to the absence of instrumentation and the lack of NOP instructions, it is quickly identified as a non-critical store instruction, allowing the attack to be detected without checking the allowlist. Exploit No #6 was also detected by *BSM*. We employed an additional pair of DWT comparators to monitor the memory range encompassing all DWT configurations. This range spans from the start address 0x0xE0001000 to the end address 0xE0001FFC, ensuring comprehensive monitoring of the memory associated with DWT configurations [52]. On this basis, any writes to this memory region (i.e., DWT configurations) would trigger an interrupt. Like the previous exploits (e.g., Exploit No #3 and No #5), this attack involved a non-critical store instruction, and it was swiftly detected and prevented by the DFI_Validation_Module. In summary, our security validation test experiments empirically demonstrate the effectiveness of BSM in preventing potential attacks.

6.6 Security Analysis

While *BSM* prevents potential attacks as demonstrated in Section 6.5, one also needs to consider potential negative security impact of *BSM*. In this section, we conduct a thorough examination of its potential attack surface. As outlined in Section 3, our threat model considers the possibility of attackers discovering memory corruption vulnerabilities within the normal world and exploiting them remotely through I/O interfaces (e.g., network and UART). For instance, they may seek to hijack control flow or manipulate critical variables using data-only attacks. Furthermore, we assume that the software within the TEE is free from bugs, which is also widely accepted in multiple existing research studies [4, 56, 70]. To bypass *BSM*, attackers would need to: ① discover and exploit vulnerabilities in the newly implemented code within the secure world; ② modify important metadata in secure world, such as bitmap-based allowlists; ③ tamper with DWT mechanisms, such as modifying DWT configurations; and ④ employ traditional attack methods, such as altering critical variables or hijacking control flow with desired function pointers.

Let us consider these risks one by one, starting with ①. Like existing work, we assume that the code in the secure world is bug-free. Even so, let us consider how realistic this assumption is. While in theory, the newly added code increases the attack surface, we find that the likelihood of exploitation is minimal. Firstly, the size of our added code base in TrustZone is only around 200 lines of code, which is relatively small. Secondly, the input data type of secure APIs is limited (e.g., the source and destination tuple (A, B)). Moreover, we ported the secure APIs to Linux and we subjected them to an extensive fuzzing campaign on Linux for 24 hours, where no bugs were found. Consequently, ① can be dismissed. ② can also be ruled out. Normal world code cannot directly access metadata in the secure world due to the protection provided by the TEE. The only way to modify metadata is by either abusing secure APIs or exploiting vulnerabilities within these APIs. However, since *BSM* incorporates CFI, any attempt to hijack control flow would be promptly detected. Moreover, we assume that secure APIs are free of bugs, eliminating the possibility of exploiting TEE vulnerabilities. Thus, ② is eliminated. As demonstrated in Section 6.5, ③ is impossible. Any modification to DWT configurations triggers security checks, and we quickly detect such attacks because no instruction is permitted to modify them. Therefore, ③ is not a viable threat. In Section 6.5, we showcased attack detection, where no exploits succeeded. However, this does not

imply that *BSM* is perfect. For instance, attackers can still modify other non-critical variables without detection. The overall impact on the application's security would depend on the protection goals and settings, such as determining which variables are deemed critical. Nonetheless, by deploying *BSM*, we believe that it would significantly raise the barrier for successful attacks.

7 Discussion

We design a hybrid protection scheme that overcomes two challenges associated with CFI and DFI. First, existing CFI schemes based on hashes need to calculate all jump paths in advance, including forward indirect jumps/calls and backward returns, which requires a large storage and calculation cost, especially for returns. Although a shadow stack can be used to reduce the memory and computation overhead [56], it is still vulnerable if its location cannot be properly be randomized. C-FLAT [4] only records a small program execution path; OAT [70] also focuses on only a subset of functions (critical operations). The existing solutions still do not fully protect the control flow.

Second, to check whether memory write operations are legal, existing DFI solutions need to instrument all the memory write instructions [16]. This is not affordable for embedded devices with limited resources. These schemes increase both code size and runtime overhead.

Our scheme achieves the verification of both while maintaining good performance on embedded devices. For CFI, *BSM* records all indirect forward and backward edges in a bitmap protected by TrustZone. The design of the bitmap allows the verifier to quickly check whether the device is under attack. For DFI attestation, instead of instrumenting all the write instructions, we simply need to remap the important variables/memory to DWT monitoring memory, which is supported by hardware. All writes to the memory of our concern are automatically captured, while other writes that are not security-relevant are not affected.

Handling Library Functions. When calling a library function (i.e., memcpy) in the source code, the compiler would link it with the pre-compiled library file by default. However, if the library function accesses the critical variables, we need to instrument the instructions that write to memory. The instrumentation will record the runtime memory address for further validation. This conflicts with the use of a pre-compiled library file. Therefore, in the LLVM IR pass, we will replace the related call instruction with the wrapped library function we prepared. For example, we replace the "call memcpy" with the "call_wrap_memcpy" instruction, and provide the source code of "_wrap_memcpy" function, then we instrument the source code of "_wrap_memcpy" function rather than "memcpy" function.

Handling Interrupts. Bare-metal programs for MCU-based embedded devices typically work in an interrupt-driven manner, where interrupts can occur at any stage of program execution. When an interrupt happens, the execution context of the interrupt (including the next pc instruction) will be saved on the stack. The return from interrupt is different: the processor loads the Exception Return Payload (EXC_RETURN) into the PC register either with a *BX* or a *POP* instruction. Note that EXC_RETURN is encoded to the form 0xFFXXXXXX and not interpreted as an instruction address. Instead, the processor will recognize the value of EXC_RETURN and know that there is an interrupt being handled. Then, it restores the saved context register, including restoring the *stored pc* to the PC register. Since an attack could modify the saved stack frame, and we do not know the value of *stored pc* in advance through static analysis. Therefore, no bitmap-based method is used here. Instead, we instrument the entry and exit point of the handler. Thus, we extract the *stored pc* to a shadow stack at the entry point and check it at the exit point.

Selecting Critical Variables. Generally, selecting critical variables heavily depends on application-specific semantics and protection goals. Programmers annotate variables that directly influence the

program's operational behavior as critical, based on program semantics and protection goals [15, 18, 70]. Variables with long lifecycles are also marked as critical since they are more likely to have a significant impact on the program. Determining which variables is critical and should be selected is orthogonal to our work, and this is not part of our contribution. In our design, we begin by manually annotating an initial set of critical variables then obtain a complete set of critical variables through propagation analysis. It would be interesting to explore methods that automatically infer semantic critical variables within an application for future research, which we leave as an open area for investigation. One promising direction for future research could involve leveraging machine learning-based approaches. Critical variables often exhibit application-specific semantics and distinct patterns, such as dataflow or control-flow patterns. Machine learning techniques are good at recognizing such patterns and understanding semantics across diverse domains, such as computer vision and natural language processing. We also observed that there are relevant works addressing this direction, such as the solution proposed by Wang et al. [77]. As machine learning techniques continue to advance, particularly in the areas of semantic learning and pattern recognition, we anticipate that they will significantly contribute to the domain of inferring critical variables, thereby enhancing the effective protection of embedded devices.

Supporting Other Development Boards. We implemented a prototype of the proposed enforcement system, *BSM*, on an STM32L562E-DK development board. However, our techniques are not limited to this specific board. They can be applied to any chip equipped with TEE and watchpoint debug feature, such as the STM32L552E-EK, NUCLEO-L552ZE-Q, and others.

Collision Prevention. To avoid collisions, we first considered a hashmap-based method, but found that it requires too many resources. For example, it will cause 300% memory overhead in extreme cases [72], which is unacceptable in resource-constrained deeply embedded devices. Our bitmap-based method can greatly reduce runtime and memory costs. However, it may lead to collisions, which seems to be a real attack and would influence the security of our solution. The attacker searches for illegal tuples with collisions to bypass the protection. A naive approach could potentially have collisions for CFI and DFI. For CFI, an attacker would modify the jump target for either forward or backward branches. Nevertheless, we could enumerate all potential ROP gadgets based on the ROPgadget tool [65]. If there are n indirect instructions and m gadgets, we can iterate to get $m * n$ combinations, then exclude all legal cases, and then get all the combinations that could potentially be attacked; For DFI, the potential attacks are limited to the instrumented instructions that operate on critical variables. Since CFI is guaranteed, the best way to create a collision is to overflow the critical variables. Nevertheless, we could also enumerate all the threat situations. Specifically, if there are n critical variables and m critical store instructions, we can iterate to get $m * n$ combinations, then exclude all legal cases, and then get all the combinations that could potentially be attacked. Note that we do not consider combinations of non-critical store instructions, because we can directly detect them by forward searching the NOP instruction in the exception handler and then reject it. Obviously, debug monitor interrupt is triggered by non-critical store instructions is illegal. After that, we can check the possibility of collisions and make sure these combinations do not overlap with our bitmap.

To avert this risk, we increase the size of the bitmap until it is sparse enough to eliminate the collisions. In practice, a 1-kB bitmap can map up to 8,192 different tuples, which is enough in bare-metal programs to ensure collision-free operation. Moreover, in the current implementation, a tuple (A, B) of a legal operation is stored in *only* one bit of the bitmap. We could store more information about the tuple to avoid collisions. For example, using multiple bits (e.g., 4 bit or 8 bit, similar to Bloom Filter [11]) to represent a tuple. In an extreme case, we can store an original tuple (A, B) directly in the bitmap, which is similar to the hashmap-based method. Nevertheless, we find

that the current approach works on the small devices, and we achieve low performance overhead and memory overhead even when we ensure there are no collisions.

8 Related Work

CFI. CFI enforcement has been a popular topic since it first came out. It aims to thwart attackers' attempts to manipulate software systems into performing unintended control transfers while executing code. Its primary focus is guaranteeing that the system code adheres to legitimate software control paths during system runtime. Numerous studies on CFI [1, 12, 14, 28, 53, 75, 76, 78, 80] have been proposed to protect programs from control-flow hijacking attacks. Abadi et al. [1] enforce CFI on the ×86 architecture, ensuring the control flow at runtime remains within a given CFG. Chao et al. [79] propose CCFIR, which collects all legal targets of indirect control-transfer instructions, puts them into a dedicated "Springboard section" in a random order, and then limits indirect transfers to flow only to them. Van der Veen et al. [75] present a binary-level context-sensitive CFI which tracks paths to sensitive program states and define the set of valid control edges within the state context to yield higher precision than existing CFI implementations. PIBE [28] offers comprehensive protection against control-flow hijacking at a fraction of the cost of existing solutions, by revisiting design choices in the compiler's optimization passes. While these schemes try to find a practical trade-off between runtime overhead and the level of precision, they are not built specifically for embedded systems.

With the emergence of the IoT and Cyber Physical Systems, the security of embedded devices becomes a concern for researchers and industry. However, embedded systems are typically subject to severe resource constraints, such as runtime, energy, and memory, it is very challenging to deploy general security protection schemes on them. Thus, some researchers systematically explore existing CFI protections targeting resource-cone-constrained embedded devices [54].

Buffer overflows are very common software flaws. An attacker can use a buffer overflow on the stack to overwrite the return address in the stack frame, thereby redirecting the program's control flow and executing arbitrary code. Backward control-flow edges in embedded programs are an attractive target for hackers and need to be protected. There has been a considerable amount of research to address this problem. Davi et al. [22] enforce that a return only targets a valid return landing pad of a function whose label is currently active. This requires the addition of new labels/instructions to the code. Almakhdhub et al. [6] suggest enforcing **Return Address Integrity (RAI)** where the return address cannot be modified by an attacker. They introduce a technique that moves return addresses from writable memory to readable and executable memory, preventing write access to the return address. Utilizing the **Memory Protection Unit (MPU)** and the unprivileged store instruction on embedded ARM architectures, Zhou et al. [81] present a compiler-based defense scheme, Silhouette, that efficiently guarantees the integrity of return addresses. Silhouette combines an incorruptible shadow stack for return addresses with checks on forward control flow and memory protection to ensure that all functions return to the correct dynamic caller. While Silhouette uses binary instrumentation to prevent a privileged attacker from modifying the MPU that hides the shadow stack, Nyman et al. [56] supports hardware-based shadow stack protection by TrustZone-M security extensions. The scheme they propose is the first interrupt-aware CFI scheme for low-end MCUs.

With the increasing popularity of backward edge defense mechanisms, attackers are beginning to consider other points of interest (indirect function calls and indirect jumps) to redirect the control flow. Hence, researchers have proposed defense schemes to counter attacks that exploit forward-edge control-flow manipulation as well. When it comes to deploying forward-edge CFI in resource-constrained embedded devices, the memory requirements of storing and enforcing the

CFG becomes an issue that needs to be addressed. Coarse-grained forward-edge CFI is occasionally applied in resource-constrained embedded systems because of its reduced memory and processing requirements. For example, Mingwei et al. [80] propose to check partial branches/jumps. Silhouette [81] utilizes a labeling mechanism to guarantee forward-edge CFI. However, such mechanisms are not precise enough and can still be exploited by attackers [23]. Therefore, Das et al. [21] present an approach to enforce fine-grained CFI at a basic block level for embedded devices, which is effective against runtime attacks on memory with acceptable performance overhead. Wenjian et al. [37] propose to employ basic block information inside the binary code and read-only data to enforce CFI. They identify a key binary-level property called basic block boundary, and based on it, they propose the code-inspired method where short code sequences can endorse a control-flow transition.

While these works can effectively protect embedded devices, they still need to fully protect the control flow, and their overhead could be further reduced. Meanwhile, they propose CFI enforcement schemes on the embedded devices but do not take dataflow integrity into consideration. Then, we propose *BSM*, which reduces the costs by using a compact bitmap-based allowlist to store and find information for forward and backward control flows. In addition, *BSM* also implements the dataflow integrity protection scheme.

Runtime Data Protection. Besides control-flow hijacking, which can cause unexpected execution paths at runtime, manipulation of non-control data that can affect the output of the embedded devices also causes significant damage because of the interaction between embedded devices and the real world. There are several works aiming to protect dataflow integrity. WIT [5] prevents memory error exploits by maintaining a color table that maps memory addresses to colors. At runtime, instructions are prevented from modifying objects that are not in the same color. Miguel et al. [16] present a simple technique that computes a dataflow graph using static analysis and instruments the program to ensure that the flow of data at runtime is allowed by the dataflow graph. HDFI [68] is hardware-assisted dataflow isolation that provides security protection. TMDFI [51] is a hardware dataflow integrity implementation to enable fine-grained DFI checks with reduced time and space overheads. DataShield [15] separates the memory of the embedded devices into two regions, a region for the sensitive data that need to be precisely protected and a region for non-sensitive data that only need to be coarsely protected. Dataflow between variables in different regions is forbidden. By doing so, DataShield can prevent sensitive data from being read or written.

Unlike for CFI, there are relatively few works to detect data-only attacks on embedded systems. Sun et al. [70] proposed OAT, first formulated a new security property for embedded devices, called "**Operation Execution Integrity (OEI)**." Based on this property, they designed a system that enables remote OEI attestation for ARM-based embedded devices. However, it does not work on low-end devices since no low-end ARM CPU with TrustZone was available at that time. Instead, it works on ARM Cortex-A53, a more powerful processor that is not suitable for deeply embedded devices. Although the overhead is decreased, OAT still requires significant resources (i.e., nearly 10 kB memory overhead and several seconds to verify the legitimacy of the operation). Due to the significant overhead in the verification, OAT does not support deploying online; instead, it resorts to remote attesting combined with a cloud backend. We have to note that frequent communication with the cloud could also increase the overhead for low-end devices, including memory, runtime, and power consumption.

In contrast, we propose a novel approach that uses the DWT and TrustZone features, as well as a bitmap-based allowlist mechanism. Among, DWT widely exists in low-end processors after ARMv7. It greatly reduces the amount of code for instrumentation (only need to instrument the limited define-sites). Moreover, only accesses to the monitored memory will automatically trigger the verification logic, while the others will not. Meanwhile, the use of the bitmap-based allowlist

Table 6. Comparison with Existing Researches

Research	CFI	DFI	Remote Attestation (Offline)	Online
C-FlAT [4]	✓	×	✓	×
Lo-FAT [27]	✓	×	✓	×
CFI CaRE [56]	✓	×	×	✓
Silhouette [81]	✓	×	×	✓
μRAI [6]	✓	×	×	✓
LiteHAX [26]	✓	✓	✓	×
OAT [70]	✓	✓	✓	×
BSM	✓	✓	×	✓

greatly reduces the memory overhead (1.56 kB compared with 10 kB of OAT) and can efficiently verify the legitimacy of the operation. In addition, when an illegal event occurs, we also support saving the relevant context for offline analysis.

Comparison with Existing Work. BSM differs from existing research as outlined in Table 6. The existing body of work in the context of deeply embedded systems has mainly focused on enforcing CFI, with relatively fewer efforts dedicated to enforcing DFI [54]. In the context of general platforms, enforcing dataflow integrity typically necessitates heavy instrumentation [5, 16], which can be costly and impractical for resource-constrained embedded devices. To reduce the overhead, we can draw lessons from CFI research, which has explored lightweight approaches by leveraging hardware features, as well by as offloading work to remote servers. For instance, C-FLAT [4] and Lo-FAT [27] utilize remote servers for legitimacy checks, albeit in an "offline" manner. Conversely, CFI CaRE [56], Silhouette [81] and μRAI [6] offer stronger "online" security by leveraging TrustZone, specific instructions (e.g., unprivileged store instructions) or dedicated registers (e.g., State Register), which are hardware features provided by the MCUs. There are also recent efforts in enforcing DFI in embedded devices that rely on remote servers for protection [26, 70]. However, these approaches are limited to "offline" protection, only capable of detecting attacks after the fact. Recognizing this limitation, Jakkamsetti [41] has highlighted the need for low-overhead online detection and prevention as a future research direction. Accordingly, we propose *BSM* as a step towards this direction, introducing a novel compact bitmap-based allowlist approach and exploring hardware features such as DWT and TrustZone to offer stronger "online" protection.

9 Conclusions

CFI and DFI are two essential invariants we want to enforce in a computing system. While existing CFI/DFI solutions have shown encouraging results in protecting commodity OSs, they consume excessive memory and computation resources, making them unaffordable in resource-constrained MCU-based embedded systems. Considering the limited resources in embedded systems, we propose a new framework to implement online security monitoring mechanisms based on a compact bitmap-based runtime data structure. In particular, we propose a hardware-assisted solution for DFI, which can reduce code instrumentation and improve performance. We have implemented our idea with a prototype named STM32L562E-DK for ARM Cortex-M series MCUs with TrustZone support. Our evaluation results show that *BSM* can successfully detect control-flow hijacking attacks and data-only attacks with minimal performance overhead.

Acknowledgment

We would like to thank all the editors and anonymous reviewers for their insightful comments and feedback.

References

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

[2] Ali Abbasi and Majid Hashemi. 2016. Ghost in the plc designing an undetectable programmable logic controller rootkit via pin control attack. *Black Hat Europe* 2016, 1–35.

[3] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. 2019. Challenges in designing exploit mitigations for deeply embedded systems. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS P'19)*. 31–46. DOI: https://doi.org/10.1109/EuroSP.2019.00013

[4] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: Control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 743–754.

[5] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing memory error exploits with WIT. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'08)*. IEEE, 263–277.

[6] Naif Saleh Almakhdhub, Abraham A. Clements, Saurabh Bagchi, and Mathias Payer. 2020. μRAI: Securing embedded systems with return address integrity. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20)*.

[7] ARM Ltd. 2021. Arm TrustZone Technology. Retrieved from https://developer.arm.com/ip-products/security-ip/trustzone/

[8] ARM Ltd. 2023. ARM Cortex-M Programming Guide to Memory Barrier Instructions. Retrieved from https://documentation-service.arm.com/static/5efefb97dbdee951c1cd5aaf

[9] ARM Ltd. 2024. Discovery Kit with STM32L562QE MCU. Retrieved from https://www.st.com/en/evaluation-tools/stm32l562e-dk.html

[10] Gal Beniamini. 2017. Over The Air: Exploiting Broadcom's Wi-Fi Stack. Google Project Zero Blog. Retrieved from https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html

[11] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (1970), 422–426.

[12] Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 340–353.

[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)* 50, 1 (2017), 1–33.

[14] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. {Control-Flow} bending: On the effectiveness of {Control-Flow} integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*. 161–176.

[15] Scott A. Carr and Mathias Payer. 2017. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 193–204.

[16] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. 147–160.

[17] Patrick Cégielski and Denis Richard. 2001. Decidability of the theory of the natural integers with the cantor pairing function and the successor. *Theoretical Computer Science* 257, 1–2 (2001), 51–77.

[18] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-control-data attacks are realistic threats. In *Proceedings of the USENIX Security Symposium*, Vol. 5.

[19] Abraham A. Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. {ACES}: Automatic compartments for embedded systems. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. 65–82.

[20] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. 1201–1218.

[21] Sanjeev Das, Wei Zhang, and Yang Liu. 2016. A fine-grained control flow integrity approach against runtime memory attacks for embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 11 (2016), 3193–3207.

[22] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. 2014a. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Proceedings of the 51st Annual Design Automation Conference*. 1–6.

[23] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014b. Stitching the gadgets: On the ineffectiveness of {Coarse-Grained}{Control-Flow} integrity protection. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*. 401–416.
[24] Enrico De Santis, Antonello Rizzi, and Alireza Sadeghian. 2017. A learning intelligent system for classification and characterization of localized faults in smart grids. In *Proceedings of the 2017 IEEE Congress on Evolutionary Computation (CEC'17)*. IEEE, 2669–2676.
[25] Majid Derhambakhsh. 2022. PID Controller Library for ARM CortexM (STM32). Retrieved from https://github.com/Majid-Derhambakhsh/PID-Library
[26] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. 2018. Litehax: Lightweight hardware-assisted attestation of program execution. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*. IEEE, 1–8.
[27] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, and Ahmad-Reza Sadeghi. 2017. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.
[28] Victor Duta, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. 2021. PIBE: Practical kernel control-flow hardening with profile-guided indirect branch elimination. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 743–757.
[29] Nicolas Falliere, Liam O. Murchu, and Eric Chien. 2011. W32. stuxnet dossier. *White paper, Symantec Corporation, Security Response* 5, 6 (2011), 29.
[30] FDA. 2017. Cybersecurity Vulnerabilities Identified in Implantable Cardiac Pacemaker.
[31] B. Feng, A. Mera, and L. Lu. 2020. {P2IM}: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the USENIX Security Symposium*. 1237–1254.
[32] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. 2020. SweynTooth: Unleashing mayhem over bluetooth low energy. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. USENIX Association, 911–925.
[33] Luis Garcia, Ferdinand Brasser, Mehmet Hazar Cintuglu, Ahmad-Reza Sadeghi, Osama A. Mohammed, and Saman A. Zonouz. 2017. Hey, my malware knows physics! Attacking PLCs with physical model aware rootkit. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*. 1–15.
[34] Giovani Gracioli and Antônio A. Fröhlich. 2008. An operating system infrastructure for remote code update in deeply embedded systems. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*. 1–5.
[35] Benjamin Green, Richard Derbyshire, Marina Krotofil, William Knowles, Daniel Prince, and Neeraj Suri. 2021. PCaaD: Towards automated determination and exploitation of industrial systems. *Computers & Security* 110, 102424.
[36] Andy Greenberg. 2016. Hacker Says He can Hijack a $35K Police Drone a Mile Away. Retrieved from https://www.wired.com/2016/03/hacker-says-can-hijack-35k-police-drone-mile-away/
[37] Wenjian He, Sanjeev Das, Wei Zhang, and Yang Liu. 2020. BBB-CFI: lightweight CFI approach against code-reuse attacks using basic block information. *ACM Transactions on Embedded Computing Systems (TECS)* 19, 1 (2020), 1–22.
[38] Hex Five Security, Inc. 2021. The First TEE For RISC-V. Retrieved from https://hex-five.com/multizone-security-sdk/
[39] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 969–986.
[40] Troy Hunt. 2016. Controlling Vehicle Features of Nissan Leafs Across the Globe Via Vulnerable Apis. Retrieved from https://www.troyhunt.com/controlling-vehicle-features-of-nissan/
[41] Sashidhar Jakkamsetti. 2023. *Root-of-Trust Architectures for Low-End Embedded Systems*. University of California, Irvine.
[42] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'14)*.
[43] Ori Karliner. 2018. FreeRTOS TCP/IP Stack Vulnerabilities–The Details. Retrieved from https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/
[44] Julie A. Kientz, Shwetak N. Patel, Brian Jones, E. D. Price, Elizabeth D. Mynatt, and Gregory D. Abowd. 2008. The georgia tech aware home. In *Proceedings of the CHI'08 Extended Abstracts on Human Factors in Computing Systems*. 3675–3680.
[45] M. Kol and S. Oberman. 2020. 19 Zero-Day Vulnerabilities Amplified by the Supply Chain. JSOF, White Paper. Retrieved from https://www.jsof-tech.com/disclosures/ripple20/
[46] Philip Koopman. 2004. Embedded system security. *Computer* 37, 7 (2004), 95–97.
[47] Sreenath Krishnadas. 2016. *Concept and Implementation of Autosar Compliant Automotive Ethernet Stack on Infineon Aurix Tricore Board*. Master's thesis, Universitätsbibliothek Chemnitz.

[48] Keen Security Lab. 2017. New Car Hacking Research: Tesla Motors. Retrieved from https://keenlab.tencent.com/en/2017/07/27/New-Car-Hacking-Research-2017-Remote-Attack-Tesla-Motors-Again/
[49] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization, 2004 (CGO'04)*. IEEE, 75–86.
[50] Lauterbach GmbH. 2022. MIPS Debugger and Trace. Retrieved from https://www2.lauterbach.com/pdf/debugger_mips.pdf
[51] Tong Liu, Gang Shi, Liwei Chen, Fei Zhang, Yaxuan Yang, and Jihu Zhang. 2018. TMDFI: Tagged memory assisted for fine-grained data-flow integrity towards embedded systems against software exploitation. In *Proceedings of the 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE'18)*. IEEE, 545–550.
[52] ARM Ltd. 2021. Armv8-M Architecture Reference Manual. Retrieved from https://developer.arm.com/documentation/ddi0553/
[53] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 941–951.
[54] Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. 2022. Survey of control-flow integrity techniques for real-time embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)* 21, 4 (2022), 1–32.
[55] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 914–926.
[56] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N. Asokan. 2017. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 259–284.
[57] Rapid7. 2016. Multiple Vulnerabilities in Animas Onetouch Ping Insulin Pump. Retrieved from https://blog.rapid7.com/2016/10/04/r7-2016-07-multiple-vulnerabilities-in-animas-onetouch-ping-insulin-pump/
[58] RJMSTM. 2020a. CRC Data. Retrieved from https://github.com/STMicroelectronics/STM32CubeL5/tree/master/Projects/STM32L562E-DK/Examples/CRC/CRC_Data_Reversing_16bit_CRC/
[59] RJMSTM. 2020b. CRYP. Retrieved from https://github.com/STMicroelectronics/STM32CubeL5/tree/master/Projects/STM32L562E-DK/Examples/CRYP/CRYP_AESModes/
[60] RJMSTM. 2020c. OSPI. Retrieved from https://github.com/STMicroelectronics/STM32CubeL5/tree/master/Projects/STM32L562E-DK/Examples/OCTOSPI/OSPI_NOR_MemoryMapped/
[61] RJMSTM. 2020d. RNG MultiRNG. Retrieved from https://github.com/STMicroelectronics/STM32CubeL5/tree/master/Projects/STM32L562E-DK/Examples/RNG/RNG_MultiRNG/
[62] RJMSTM. 2020e. RTC Alarm. Retrieved from https://github.com/STMicroelectronics/STM32CubeL5/tree/master/Projects/STM32L562E-DK/Examples/RTC/RTC_Alarm/
[63] RJMSTM. 2020f. STM32CubeL5. Retrieved from https://github.com/STMicroelectronics/STM32CubeL5/tree/master/Projects/STM32L562E-DK/
[64] Eyal Ronen, Colin O'Flynn, Adi Shamir, and Achi Or Weingarten. 2017. IoT goes nuclear: Creating a ZigBee chain reaction. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'17)*. IEEE, 195–212.
[65] Jonathan Salwan. 2023. ROPgadget Tool. Retrieved from https://github.com/JonathanSalwan/ROPgadget
[66] Vijayalakshmi Saravanan, Senthil Kumar Chandran, Sasikumar Punnekkat, and D. P. Kothari. 2011. A study on factors influencing power consumption in multithreaded and multicore cpus. *WSEAS Transactions on Computers* 10, 3 (2011), 93–103.
[67] SiFiv, Inc. 2022. RISC-V Debug Specification. Retrieved from https://github.com/riscv/riscv-debug-spec/blob/master/riscv-debug-stable.pdf
[68] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-assisted data-flow isolation. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 1–17.
[69] Yu Su, Fei Hou, Mingde Qi, Wanxuan Li, and Ying Ji. 2021. A data-enabled business model for a smart healthcare information service platform in the era of digital transformation. *Journal of Healthcare Engineering* 2021, 1–9.
[70] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. 2020. OAT: Attesting operation integrity of embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'20)*. IEEE, 1433–1449.
[71] Sebastian Surminski, Christian Niesler, Ferdinand Brasser, Lucas Davi, and Ahmad-Reza Sadeghi. 2021. Realswatt: Remote software-based attestation for embedded devices under realtime constraints. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2890–2905.
[72] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 48–62.

[73] MSRC Team. 2021. BadAlloc–Memory Allocation Vulnerabilities Could Affect Wide Range of IoT and OT Devices in Industrial, Medical, and Enterprise Networks. Retrieved from https://msrc-blog.microsoft.com/2021/04/29/badalloc-memory-allocation-vulnerabili-could-affect-wide-range-of-iot-and-ot-devices-in-industrial-medical-and-enterprise-networks/

[74] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium USENIX Security*. 941–955.

[75] Victor Van der Veen, Dennis Andriesse, Enes Göktas, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 927–940.

[76] Zhi Wang and Xuxian Jiang. 2010. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 380–395.

[77] Zhilong Wang, Haizhou Wang, Hong Hu, and Peng Liu. 2021. Identifying non-control security-critical data in program binaries with a deep neural model. arXiv:2108.12071. Retrived from https://arxiv.org/abs/2108.12071

[78] Bin Zeng, Gang Tan, and Greg Morrisett. 2011. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and Communications Security*. 29–40.

[79] 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 *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. IEEE, 559–573.

[80] Mingwei Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security'13)*. 337–352.

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

[82] Xia Zhou, Jiaqi Li, Wenlong Zhang, Yajin Zhou, Wenbo Shen, and Kui Ren. 2022. OPEC: Operation-based security isolation for bare-metal embedded systems. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 317–333.

Received 15 November 2023; revised 24 March 2024; accepted 26 May 2024