Survey of Control-flow Integrity Techniques for Real-time Embedded Systems

TANMAYA MISHRA, THIDAPAT CHANTEM, and RYAN GERDES, Virginia Polytechnic Institute and State University, USA

Computing systems, including real-time embedded systems, are becoming increasingly connected to allow for more advanced and safer operation. Such embedded systems are also often resource-constrained, for example, with lower processing capabilities compared to general-purpose computing systems like desktops or servers. With the advent of paradigms such as internet-of-things (IoT), embedded systems in both commercial and industrial contexts are being increasingly interconnected and exposed to the external networks to improve automation and efficiency of operation. However, allowing external interfaces to such embedded systems increases their exposure to attackers. With an increase in attacks against embedded systems ranging from home appliances to industrial control systems operating critical equipment that have real-time requirements, it is imperative that defense mechanisms be created that explicitly consider such resource and real-time constraints. Control-flow integrity (CFI) is a family of defense mechanisms that prevent attackers from modifying the flow of execution. We survey CFI techniques, ranging from the basic to state of the art, that are built for embedded systems and real-time embedded systems and find that there is a dearth, especially for real-time embedded systems, of CFI mechanisms. We then present open challenges to the community to help drive future research in this domain.

CCS Concepts: • **Security and privacy** → **Embedded systems security**; • **General and reference** → **Surveys and overviews**; • **Computer systems organization** → **Embedded software**; **Real-time system architecture;**

Additional Key Words and Phrases: Survey, control-flow integrity, real-time systems, embedded systems

ACM Reference format:
Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. 2022. Survey of Control-flow Integrity Techniques for Real-time Embedded Systems. *ACM Trans. Embedd. Comput. Syst.* 21, 4, Article 41 (October 2022), 32 pages.
https://doi.org/10.1145/3538275

1 INTRODUCTION

Today, computing systems communicate through a complex web of interconnections. For instance, the modern smartphone can simultaneously capture photographs and videos at quality rivaling

This work is supported by the National Science Foundation under grant no.: 2038726, and by the Commonwealth Cyber Initiative CCI is an investment in the advancement of cyber R&D, innovation and workforce development in Virginia. For more information about CCI, visit cyberinitiative.org.
Authors' address: T. Mishra, T. Chantem, and R. Gerdes, Virginia Polytechnic Institute and State University, USA, 22311; emails: {tanmayam, tchantem, rgerdes}@vt.edu.

1539-9087/2022/10-ART41 $15.00
https://doi.org/10.1145/3538275

that of movie cameras, upload gigabytes of information to the internet, turn on lamps and automatically control thermostats, stream high-fidelity music to the nearest speaker, and even unlock a car. We now live in the age of the **internet-of-things (IoT** [8]), where the physical world around us can be manipulated by the push of a button.

The convenience afforded by such interconnections is, unfortunately, countered by the inconvenience of dealing with malicious parties who try to take control of these connected devices to inflict monetarily and, in some cases, bodily harm. A simple smart bulb from a reputed company was exploited to launch a **distributed denial-of-service (DDOS)** attack [63]. While a DDOS attack may have at the most an economic impact on the victim, an attacker could reprogram the lights to blink so as to induce an epileptic attack in some individuals. Unfortunately, such instances of malicious behavior are not confined to small home appliances. Stuxnet [36] is a computer worm built to infect **supervisory control and data acquisition (SCADA)** systems. Infections of this worm were first uncovered in 2010 and by then it had already infected nuclear reactor control systems and caused significant damage to Iran's nuclear program. Malicious entities could, theoretically, cause the reactors to fail and cause catastrophic damage to both life and property. Interestingly, many Stuxnet systems were air-gapped, i.e., did not have a direct connection to external systems. Instead, the infection spread from physical drives inserted by human operators.

Over the years, a variety of system defense mechanisms have been proposed for a wide range of threat models and system configurations. These mechanisms can be hardware assisted [39]; entirely in software [91]; implemented in the system pre-deployment, such as compiler-based protections [71]; or detect attacks during system runtime [37]. Discussing the entire body of work of such defenses is beyond the scope of this survey. We therefore focus on a class of defense mechanisms, collectively called **control-flow integrity (CFI)**, which are designed to defend against a powerful set of attacks, called control-flow attacks, that can allow attackers to have arbitrary control over program execution. In this work, we discuss CFI for embedded and, particularly, real-time embedded systems.

Our major contributions are:

(1) We explore a number of recently proposed mechanisms targeting embedded systems, specifically those that are *resource-constrained*, such as reduced processing capabilities over general-purpose processing environment systems such as those found in desktop or server-grade equipment. Such embedded systems usually feature low-end processing environments such as microcontrollers (and their related underlying processor architecture). We also identify key techniques that could provide inspiration for more robust real-time system CFI design.
(2) We find that there are very few CFI mechanisms built specifically for real-time embedded systems. Our exploration of the work for embedded systems shows that there is an avenue to extend CFI techniques from general embedded systems to create powerful CFI mechanisms that uniquely leverage real-time properties.
(3) We consolidate our findings and present challenges and suggestions for future research in Section 6.

We give definitions for *embedded systems* and *real-time embedded systems* later in this section, which defines the scope of our survey. We will now provide an overview of the type of attacks that are countered by CFI and an overview of CFI itself.

1.1 Scope of Attacks and Defenses: Control-flow Attacks and CFI

To aid the discussion of CFI, which is the main focus of this survey, it is necessary to first describe the type and scope of attacks for which they are built. This family of attacks is collectively called control-flow attacks. We shall now discuss these types of attacks.

1.1.1 Control-flow Attacks. Control-flow attacks capture and modify the *flow of execution* of a program. These attacks attack *control information*, that is, information presented to a program during runtime that determines the path that a program takes to continue execution. A simple example of such information is the return address of a function call. See Figure 1(a) for an example of the stack frame of a function call on a generic ARM architecture-based microcontroller. Here, the value stored in the LR field of the stack frame is popped into the special `LR` or link register. ARM calling convention [33], which is implemented by all compilers that officially support this architecture, utilizes the `LR` to implement the return sequence of a function call. Return sequences are implemented by branching on the `LR` such as by using the `BX LR` instruction. Therefore, the contents of the `LR` effectively constitutes control information. Control-flow attacks aim to modify such information to redirect program execution for malicious purposes. The same figure showcases a sample attack where the attacker utilizes a buffer-overflow bug in the code that writes to a memory buffer in the stack frame, such that it uses the bug to overwrite the `LR` information, thereby tainting the return address with a desired target address. Therefore, when the function returns, the tainted value is popped and becomes the target of the branch statement. Note that since control-flow attacks redirect program execution, they are also sometimes called *code redirection* attacks.

Two broad categories exist in control-flow attacks. While each category is a large research domain by itself, we briefly describe them here for context:

(1) Code injection attacks - The sample attack we discuss above is a simple example of a code injection attack. As discussed above, the attack can be broken into two stages that are (a) injecting (writing) code into some form of executable memory, followed by (b) a redirection to the beginning of the injected code, such as by using the `LR` register. Due to code injection that takes place in stage (a), these attacks are termed code injection attacks. Code injection attacks have a large body of work [38, 42, 72]. However, such attacks have lost favor over time with advancements in software and hardware architecture. Note that an implicit assumption of the attack is that code is injected into executable memory; that is, the stack is executable. Therefore, to defeat such attacks, it is sufficient to introduce countermeasures that ensure that writeable memory addresses are not executable. A large body of research has been presented to counter code injection attacks with relatively inexpensive performance overheads [51, 58]. Even for lower-end processors, such as microcontrollers from the ARM Cortex-M family, prior work has implemented defenses [57]. Modern hardware now includes architectural features such as the **memory protection unit (MPU)** that make it trivial for system designers to implement writeable but non-executable memory, an important requirement for code injection attacks to propagate [22]. Since such attacks can be defended against relatively easily, such attacks are outside the scope of this survey.

(2) Code reuse attacks - With the addition of defenses against code injection attacks, a new class of attacks emerged that are collectively called code reuse attacks. These attacks are a logical extension of code injection attacks where attackers modify control information to reuse arbitrary sequences of code already present within the program binary to perform malicious operations. One of the most famous examples of code reuse attacks is the return-oriented programming or ROP [75, 81, 92] attack. We provide more details of ROP, and defenses against such attacks, in Section 3. Increasingly sophisticated variants of ROP [17], such as some that do not even require return sequences [18, 29], have been proposed over the past decade. A large set of defenses have also been proposed to counter ROP and related attacks [20, 52, 88], showcasing the relevance and danger such types of attacks represent for modern systems.

Since control-flow attacks modify the control flow of the program, it is necessary to maintain the *integrity* of the control flow by detecting malicious control-flow deviation when it occurs. Therefore, any defense mechanism that enforces this integrity is called control-flow integrity [4]. In this survey, we discuss CFI that is specifically designed to defend against code reuse attacks. Note that we alternatively refer to CFI as CFI enforcement, CFI mechanism, or CFI technique throughout this article.

1.1.2 Control-flow Integrity (CFI). CFI is the set of system security techniques built to prevent an attacker from forcing a software system to execute code in an unintended manner. CFI focuses on ensuring that system code does not deviate from known software control paths during system runtime. CFI mechanisms are built to address powerful threat models where it is assumed that the attacker can bypass all other defenses to infiltrate the system and force system software to execute in an arbitrary manner. There is a wealth of research in recent years that develops CFI mechanisms for increasingly complex and powerful attack scenarios [13, 31]. CFI mechanisms are also available in many commercial and production-grade software. For example, the Clang compiler implements control-flow violation detection mechanisms [1], and Microsoft has its own CFI implementation called Control Flow Guard [62] for its Windows operating system, which has been available since Windows 8.1.

While the literature concerning CFI mechanisms (and techniques to bypass them [16, 34]) is rich with studies regarding the non-negligible performance and/or memory overhead of the mechanisms, few are built specifically for embedded systems, and even fewer explicitly consider the real-time requirements of such systems. Therefore, we shall first look at CFI mechanisms for general embedded systems and then move toward mechanisms built explicitly for those with real-time constraints. We shall look at both software-based and hardware-assisted mechanisms, as well as a mechanism that takes advantage of the predictability of real-time systems. However, before we begin discussion of CFI techniques, we shall now define resource and real-time constraints.

1.2 Systems Considered: Embedded and Real-time Embedded Systems

There exists numerous prior works that are excellent surveys and compilations regarding CFI defenses for general systems [4, 13, 31, 79, 85]. However, none of these works explicitly considers system capabilities and constraints. We now define the types of systems that we consider for the rest of this work, and their constraints that influence the design of CFI for such systems.

1.2.1 Embedded Systems. As discussed earlier, the Stuxnet worm was built specifically to target and control SCADA systems. A SCADA system is usually composed of a number of embedded computing systems built for specific operations, such as data gathering and actuator control. However, embedded computing systems themselves can be found in a wide variety of operating environments, ranging from complex SCADA systems to robots used for medical procedures as well as small household appliances. These embedded systems are usually severely *resource-constrained* to minimize **size, weight, and power (SWaP)** and cost and/or simplify operations. Typically, they consist of microcontrollers that are low-end processors with integrated memory, executing software built to perform specific operations in a deterministic and predictable manner. For example, the modern vehicle can have over 100 individual computing units, called **Electronic Control Units (ECUs)**, that control different functionalities of the vehicle. These units usually consist of microcontrollers [56] that operate at a clock frequency an order of magnitude lower than the processors found in modern internet servers, and have similarly small amounts of memory for storage and operation. These computing units control vehicle operations ranging from non-critical infotainment systems to extremely critical **Advanced Driver Assistance Systems (ADASs)**, such as anti-lock braking systems, whose failure could result in passenger loss of life. Further, the software

for such systems may not be regularly updated due to the inaccessibility of their deployment locations. Therefore, once security vulnerabilities in the software are found, they may not be easily patched, making them lucrative targets for malicious entities. In addition, in the case of modern vehicles, increasing inter-vehicular connectivity to improve ADASs as well as the increasing number of interfaces such as WiFi and Bluetooth for passenger convenience has widened the attack surface that can be exploited by such entities [19]. Therefore, due to the wide range of applications of resource-constrained embedded systems and their increasing attack surface due to system inter-connectivity, it is imperative that such systems have built-in defense mechanisms to prevent their exploitation by attackers.

To summarize, our definition of embedded systems is in the broader sense. That is, our definition encompasses embedded systems with fixed system resources (memory, processor, peripherals, etc.) where processing elements are embedded off-the-shelf microcontroller architectures such as ARM Cortex-M and ARM Cortex-R [98] or bespoke architectures that evolve from those that could be utilized in similar systems. Such processing environments have fewer architectural features than desktop or server-grade processors and usually paired with slower/limited memory and peripherals for managing costs and/or special memory systems for redundancy and safety. Such systems are usually deployed in mission-specific applications in a wide range of domains, such as industrial, automotive, space, and medical systems or even IoT systems. Our definition of such systems is broad since it allows us some flexibility to look at CFI mechanisms that may work for a specific type of embedded system but could be applied to similar architectures with some modifications, giving us a broader field of view of the domain. For each mechanism we take a closer look at in later sections, we state the specific architectural considerations that informed its design. Note that for completeness we also briefly discuss some techniques that use external processing resources such in Section 4.5 and show their fundamental similarities with techniques that do not require external processing resources. However, we do not present in-depth information for these techniques since they utilize external processing resources that make it difficult to compare with techniques that do not require such external resources. Note that our definition of embedded systems assumes that such systems are resource-constrained and we interchangeably refer to embedded systems as *embedded systems* or *resource-constrained embedded systems* throughout this work.

1.2.2 Real-time Embedded Systems. Many resource-constrained embedded systems require real-time guarantees. In the case of ADASs such as anti-lock braking systems, for example, multiple control loops (including actuator control) must be completed per second to maintain safe vehicle operation. We term such embedded systems as *real-time embedded systems*. If such a system misses any deadline, regardless of the correctness of the computation, the consequences could include the loss of life. When such guarantees are required atop resource constraints, developing defense mechanisms for such systems becomes especially challenging.

We therefore focus on defense mechanisms that are built for embedded systems and real-time embedded systems. Since such systems have both *resource* and *real-time* constraints, considering systems that have a combination of these two types of constraints leads to a unique set of problems for designing useful CFI mechanisms for such systems. In general, some of the problems are:

(1) Weaker processing capabilities as compared to general-purpose desktop or server-grade systems constrain the complexity of the design and scope of the CFI mechanism that can be introduced in the system. Complex CFI would introduce unmanageable overheads that would break the real-time guarantees of the system. For example, most of the defenses we discuss specifically for embedded systems in Section 4 detect irregularities in branch source and targets individually for each branch. However, general-purpose architectures have more

complex mechanisms available [20, 25] since such systems are not constrained by real-time guarantees and can accept greater performance reductions for a higher degree of security.
(2) In addition, due to reduced hardware capabilities, certain defense mechanisms that are built for general-purpose systems may not be directly applicable to resource-constrained embedded systems. For example, some defenses [20] require advanced memory management features such as virtual memory that are not available on low-end microcontrollers. Therefore, defenses for such systems require hardware/software workarounds to maintain acceptable levels of defense without hampering real-time operation.
(3) Real-time systems require a study of the increased overhead due to CFI mechanisms and its impact on the schedulability of the system. Many CFI techniques designed for general-purpose and, in fact, as we see later in Section 4, resource-constrained embedded systems do not discuss schedulability, nor do they discuss possible security-schedulability tradeoffs that may be required to balance timing and security.
(4) Other system parameters, such as power consumption, are rarely considered when discussing CFI. Many resource-constrained embedded systems may have access to limited (such as battery-based) or intermittent (such as via renewable energy like solar) power supply. Such constraints are rarely discussed by prior work. We, in fact, realize a gap in knowledge with respect to impact of CFI and power consumption and suggest readers explore this domain in future work (see Section 6).

To the best of our knowledge, this survey is the first to identify a gap in research of CFI mechanisms for real-time embedded systems and propose future research avenues that could be considered by the real-time systems community. In this survey, we discuss CFI for real-time embedded systems and not general real-time systems that do not consider resource constraints (such as memory or low-end computation environments) typical to embedded systems. This is due to a lack of CFI literature that explicitly considers real-time constraints *without* considering resource constraints. On the other hand, we believe that our discussion of CFI for real-time embedded systems provides adequate coverage of possible techniques that can be utilized, without many modifications, for any general real-time system. We also believe there is ample opportunity to investigate the unique hardware-software constraints of resource-constrained embedded systems and utilize real-time execution characteristics to aid the development of CFI techniques that are equally applicable to real-time systems that do not suffer from resource constraints. Such timing-based co-design, as we show in later sections, is severely lacking and we present a few possible paths of investigation for the reader to follow for future work in Section 6.

2 ARTICLE ORGANIZATION

The rest of this work is divided into four major sections. These are:

(1) CFI Techniques for Backward and Forward Edges (Section 3) - We discuss different CFI designs, from both a theoretical and practical approach, for general-purpose systems. This section provides the reader a general overview of how state-of-the art CFI mechanisms, both basic and advanced, are usually designed and implemented.
(2) CFI for Embedded Systems (Section 4) - We discuss different types of CFI techniques built specifically for resource-constrained embedded systems. Please note that our definition of embedded systems is provided in Section 1.2. As stated in Section 1.2, the nomenclature "embedded systems" and "resource-constrained embedded systems" are synonymous and interchangeably used depending on context for clarity.
(3) CFI for Real-Time Embedded Systems (Section 5) - We then discuss how real-time considerations play into the design of CFI for embedded systems. Four specific techniques are

Table 1. Table of Contents of Advanced Forward-edge CFI Techniques Discussed in Section 3.2, CFI Techniques for Embedded Systems Discussed in Section 4, and CFI Techniques for Real-time Embedded Systems Discussed in Section 5

CFI Mechanisms	Forward-edge		Backward-edge	Mechanism Highlights
	Fine-grained	Coarse-grained		
Advanced forward-edge techniques for general systems (**Section 3.2**)				
BBB-CFI [48]		✓	✓	Block-based enforcement—binary-only approach without need for CFG
PathArmor [86]	✓	✓	✓	Context-sensitivity—requires architectural support
CFI for embedded systems (**Section 4**)				
Silhouette [97] (Section 4.1)		✓	✓	Uses shadow-stacks and labeling
Control-flow locking [12] (Section 4.2)	✓	✓	✓	Lazy + shadow-stack replacement.
μRAI [6], Zipper Stack [59], (Section 4.3) PACStack [60]			✓	Register-based CFI—shadow-stack replacement Interrupt-handling (μRAI)
CFI CaRE [67], (Section 4.4) TZmCFI [54]			✓	ARM TrustZone based shadow-stack, nested interrupts stronger threat models
HCFI [21] (Section 4.4)			✓	New ISA that integrates shadow-stack operations in processor pipeline
CFI for real-time embedded systems (**Section 5**)				
RECFISH [90] (Section 5.1.1)	✓		✓	Large-scale schedulability study of common CFI techniques applied to an RTOS
Improve schedulability by reducing security [45] (Section 5.1.2)			✓	Searching the number of task jobs that can have CFI turned on to improve schedulability
Timing-deviation [11] (Section 5.2.1)				Detects control-flow deviation by excess computation time
ECFI [5] (Section 5.2.2)	✓	✓	✓	CFI for hard-real time PLC code that detects abnormal increase in execution

Important highlights of each technique and degree of coarseness of forward-edge path deviations is discussed.

considered that explicitly consider real-time constraints and discuss schedulability-security trade offs and/or schedulability analyses.

(4) Summary and Open Challenges (Section 6) - We summarize our discussion of different CFI techniques and discuss some challenges from a real-time perspective and from an overall resource-constrained embedded system perspective.

Table 1 provides a brief overview of the relevant sections where we discuss specific CFI techniques, especially for Sections 3.2, 4, and 5.

3 CFI TECHNIQUES FOR BACKWARD- AND FORWARD-EDGES

We shall now look at some general techniques that are used in many CFI mechanisms. We will first look at techniques developed to prevent an attacker from modifying return sequences of function calls (*backward-edge*) or modifying other points of interest, such as indirect branches/function calls (*forward-edge*). Techniques for the former are well established and extensively utilized in mechanisms for embedded systems and real-time embedded systems (Sections 4 and 5). However, some recently proposed advanced techniques for forward-edge CFI have not yet been considered for real-time embedded systems and are highlighted in Table 1. Note that for this section and the rest of this article, "performance overhead" and "overhead," unless stated otherwise, are synonymous and refer to the increase in the CPU cycles required due to the addition of the CFI mechanism into the system. "Memory overhead" refers to the increase in the total memory (code and data) required to implement the mechanism, unless otherwise specified. Unfortunately, not all prior work discussed in this survey utilized the same benchmarking software and hardware. Nor did they always report memory overheads. We present the information regarding overheads as it was

presented in the original work. We only quantitatively compare different work if the overheads have been measured using the same combination of hardware and software. That said, we try to provide a qualitative discussion when possible to aid the reader in determining the pros/cons of the CFI technique based on the values reported.

3.1 Backward-edge CFI Techniques

The first step to any control-flow attack is infiltration. There must be some flaw in the system that can be exploited by an external attacker to begin a control-flow redirection. A very common software flaw is the buffer overflow. Due to the tight memory restrictions of embedded systems, and the flat memory model due to the lack of complex (and potentially expensive, from both economic and performance perspectives) memory management units, buffer overflow or stack overflow flaws are common in resource-constrained embedded systems since they are usually programmed using memory-unsafe languages such as C/C++ [82]. A simple example of such a flaw is a statically allocated array that is filled past its capacity. Imagine such a flaw exists within a function call of a driver code that handles user input from a keyboard. In the absence of proper memory management, such flaws can be easily exploited to overwrite adjacent locations within the function stack frame as seen in Figure 1(a). Of particular interest is the return address value in the stack frame. Overwriting the return address with a target address ensures that when the function returns, the code will continue execution at the target address, successfully redirecting the flow of the program. The target address could be a location either within the pre-existing code memory or to some other memory address. A simple use case for the latter technique is to first *inject* the malicious code into the stack memory using the overflow vulnerability, and then set the return address to the start of the injected code. When the function returns, the injected code executes. Code-injection attacks can be thwarted with the help of memory protection mechanisms that implement the W $\oplus$ X memory policy, i.e., prevent execution from writable memory. Such memory protections are now readily available in many **commercial-off-the-shelf (COTS)** low-end processors and microcontrollers. Therefore, the rest of our discussion will be focused on the consequences of the former technique of forcing the processor to continue execution at a target address in code memory.

Pointing the processor to an incorrect location by overwriting the return address is an example attack that serves as an entry point to a set of very powerful *code-reuse attacks*. For example, a well-studied sub-family of control-flow attacks is **Return-oriented Programming (ROP)** [75]. An ROP attack is where an attacker chains together arbitrary code sequences (also called *gadgets*) that are already present on the device to achieve their objective. After the seminal work by Shacham [81], ROP attacks have become increasingly popular and very sophisticated. It should be noted that using the return address to perform a control-flow diversion is also referred to as *backward-edge* control-flow attack. On the other hand, *forward-edge* control-flow attacks modify function pointers, or the targets of indirect function calls, to reuse code. An example is that by Checkoway et al. [18] that modifies the target of indirect function calls to create gadget chains. Forward-edge defenses are discussed in the next section and are slightly more ambiguous in nature. It is interesting to note that all these attacks require exploiting an initial vulnerability such as a simple buffer overflow bug.

Two simple mechanisms to deal with backward-edge control-flow attacks are stack canaries [24] and shadow stacks [15]. Both these mechanisms, especially the latter, feature heavily in more sophisticated realistic CFI mechanisms for resource-constrained embedded systems. Stack canaries are special values inserted into the stack frame and are located in between the return address and the local statically allocated variables as seen in Figure 1(b). The concept behind using stack canaries is that an attacker overwriting the stack using a buffer overflow will have to first overwrite the canary value before overwriting the return address. Checking the canary value in the stack

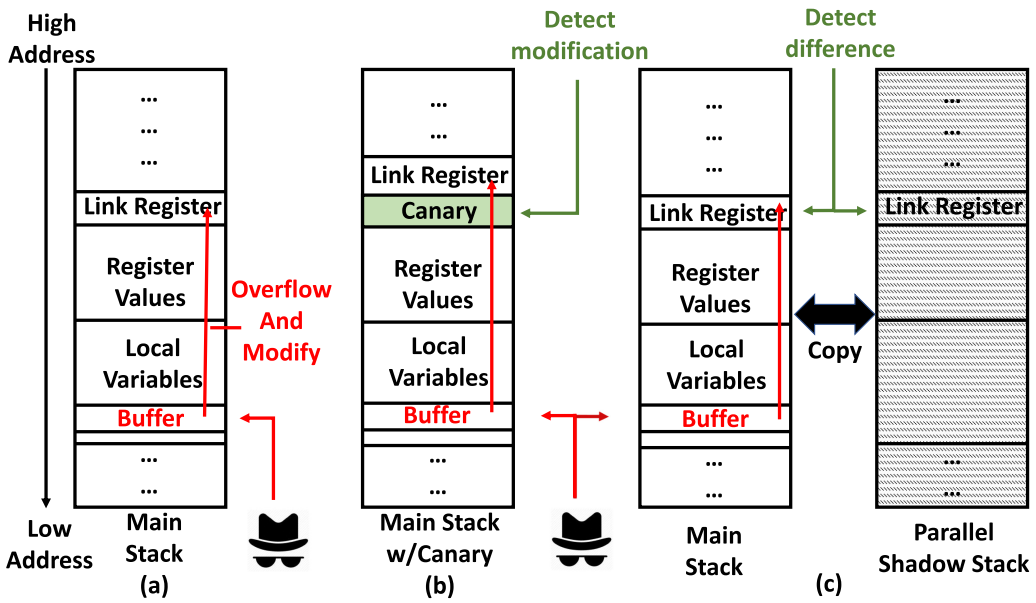

Fig. 1. (a) A function stack without any defenses. (b) Backward-edge CFI using an embedded canary. (c) A parallel shadow stack (Section 3.1).

frame before a return operation can help determine whether the return address can be trusted. However, stack canaries can be bypassed by a sophisticated attacker, especially if the canary value is known (not random) or if the value can be guessed (not random enough). Further, they do not stop the attacker from overwriting local variables located before the canary value. By doing so, the attacker can still influence the function call operation [74].

Shadow stacks are a more sophisticated defense mechanism. Under the assumption that the attacker cannot access or modify a portion of the memory, a copy of the stack frames, or at least return addresses, is kept in that memory portion. Figure 1(c) presents an example of a shadow stack. This copy is updated during the initial stages of a function call (such as in the function prologue), and the return address is checked just before the return instruction is executed. If a discrepancy exists between the stored and actual addresses, it can be indicative of an attack. Shadow stacks are essentially more sophisticated canaries since both mechanisms indicate an attack by checking for discrepancies in the contents of the stack, with the major difference being that the shadow stack keeps a copy of the correct value [27]. While these mechanisms are relatively simple, applying them comes at a cost.

Dang et al. [27] performed a study of the overheads caused by two different shadow stack implementations on the SPEC CPU2006 [2] standard suite of benchmarks on an x86 architecture processor. The first is a "traditional" shadow stack that has its own stack pointer and stores only the return addresses. The second is a "parallel" shadow stack that uses the same stack pointer as the main stack; however, the parallel shadow stack is stored at a different base address and records the return addresses while skipping over the other values in the stack frame (Figure 1(c)). Architecturally, this makes the parallel shadow stack faster than the traditional shadow stack since the same offset can be used for both the main and shadow stacks. The correct entry can be accessed by simply swapping out the contents of the stack base register, which can be achieved with a single instruction. On the other hand, a traditional shadow stack would require additional code to

maintain the stack as well as at least one extra instruction per operation to increment or decrement the shadow stack pointer for push and pop operations. Their measurements of the performance overhead show that traditional shadow stack implementation, on average, introduces a 9.69% overhead (over a system without shadow stacks), while the parallel shadow stack introduces a 3.51% overhead. Worst-case overheads of both were 52.5% and 19.6%, respectively. The cost of checking the return address was an additional 0.8%. On the other hand, stack canaries had an average performance overhead of 2.54%. At first glance the parallel shadow stack mechanism is clearly better suited to applications that are performance sensitive. As discussed above, the performance benefits of parallel shadow stacks are expected since accessing the relevant position in the parallel shadow stack only requires swapping the stack base register since both stacks share the same offset, whereas multiple operations are required for an equivalent operation on traditional shadow stacks. However, the traditional shadow stack has its merits for a resource-constrained system with a low amount of memory.

3.2 Forward-edge CFI Techniques

Forward-edge control-flow attacks are the logical extension to backward-edge attacks. The increasing popularity of backward-edge defense mechanisms forced attackers to consider other **points of interest (POI)** to redirect control flow. These POI include indirect branches and indirect function calls via pointers. By attacking the destination of these branches, the attacker could call any arbitrary location without the need for return instructions [18].

Forward-edge CFI is difficult and, in general, subtler than backward-edge CFI. This is simply because *looking at the past is easier than predicting the future*. Forward-edge CFI techniques that could theoretically predict all possible combinations of branch start and end points are called *fine-grained* [4] CFI. Valid combinations of start and end points, essentially valid control flow, can be represented as a ***control-flow graph* (CFG)**. For example, Abadi et al.'s [4] approach performs a binary static analysis using Vulcan [84] to generate a CFG and utilize said CFG to determine whether a branch is valid or not. A common mechanism to help enforce the valid control-flow paths in a CFG is *labeling*. Labeling is a process where all possible forward-edges that can be used by an attacker such as indirect branch locations, functions, and any other potential branch targets are labeled with unique IDs. Figure 2 is an example of a labeling scheme where indirect branches and function prologues are labeled and matched against a CFG. When a branch occurs, the source label (such as an indirect branch) is checked against the destination label (such as a function) via code that has been instrumented into the binary (such as checks in a function prologue). A simple example of such an approach is presented in Figure 2.

An obvious problem of this approach, especially in the resource-constrained embedded systems, is the amount of memory required to store and enforce a CFG. However, more subtle issues arise in real-world cases. Many real-time embedded systems are industrial control systems, robotics systems, and so forth. In many cases, these environments run proprietary legacy software whose source code is difficult to obtain for analysis or, due to licensing issues, does not allow instrumentation. Due to these reasons, fine-grained CFG may not be possible to obtain, or the performance overhead associated with checking every branch may be prohibitive, especially in a real-time context. Therefore, many *coarse-grained* CFIs [96] have been proposed that allow varying degrees of relaxation of which branches or jumps need to be checked and which can be ignored. Due to reduced memory and processing requirements when utilizing coarse-grained CFG, coarse-grained forward-edge CFIs are sometimes used for resource-constrained embedded systems. Due to the nature of coarse-grained CFI, such mechanisms may have blind spots that can be exploited by attackers [30]. A simple example is where a coarse-grained CFI allows any branch to any legal target, such as the start of a function, due to the unavailability of quality control-flow graphs. In such a

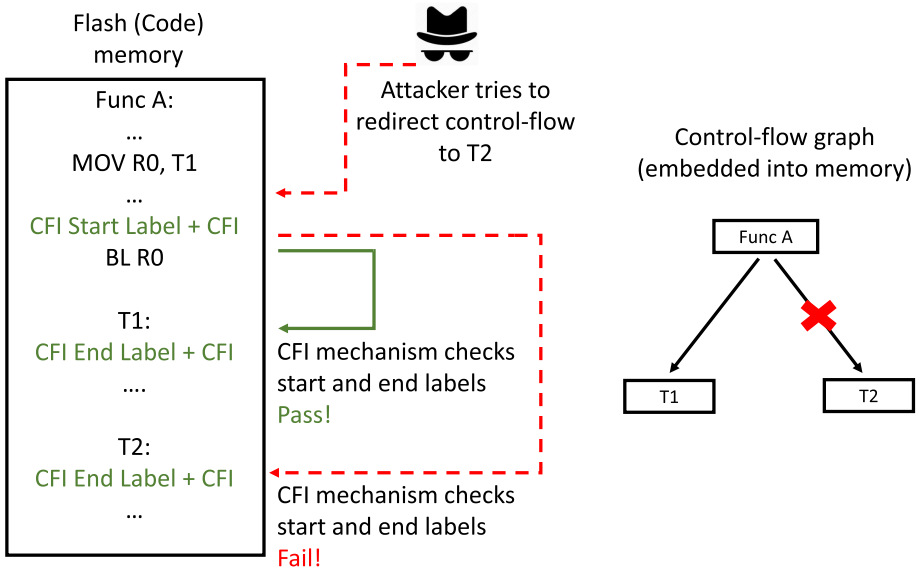

Fig. 2. Using labels and an embedded control-flow graph to enforce forward-edge CFI (Section 3.2).

case, the attacker could jump to targets that would have otherwise been identified as illegal by a fine-grained CFI. An interesting approach to overcome the need for a CFG, or the codebase to determine a CFG, is proposed by the authors of BBB-CFI [48], where the authors inspect the binary and divide it into *basic-blocks*, with each block having a single entry and exit point. A runtime mechanism prevents branches to the middle of a block, ensuring that the blocks are the smallest unit of code.

Interestingly, even fine-grained CFI can be defeated [16, 35], such as by exploiting the inability of current code static-analysis techniques to perfectly capture coding practices. Advanced forward-edge CFI techniques such as Van Der Veen et al.'s PathArmor [86] can defend against such attacks. PathArmor logs control-flow transfers and then performs path verification by having access to the program CFG and performs a depth-first comparison of the logged transfers with the CFG to determine if the path taken during runtime is legitimate. This allows checking if a legitimate pair of source and destination addresses of a control-flow transfer are also contextually correct with respect to neighboring transfer events. However, the requirement for architectural support to record control-flow transfers prevents its direct application to low-end microcontroller-based systems that lack such specialized hardware.

3.3 A Note on Control-flow Checking for Soft Errors

While CFI techniques are built considering an adversarial perspective, there exists a line of research that applies similar methodologies to detect erroneous control-flow redirection due to non-malicious soft errors [43, 73, 80, 95]. These works utilize very similar techniques, such as by creating signatures for each basic block (code blocks that are delineated by control-flow transfers but do not contain any transfers themselves) and comparing currently executing basic block against a predetermined graph of valid signature chains [68]. While such techniques utilize similar underlying principles to those discussed in prior sections, such as the forward-edge techniques in Section 3.2, soft errors are generally one-shot errors that arise due to environmental factors. Control-flow redirection that is caused due to these errors is not easily predictable. For example, a redirection could

take place due to reading the incorrect branch target from memory due to a bit-flip that took place in memory. However, control-flow redirection due to attacker control takes place under more predictable conditions (such as a buffer-overflow bug) and at a control-flow transfer point such as a branch/return statement. Further, advanced control-flow redirection, such as control-flow bending [16], where a control-flow transfer has valid start and end points but is incorrect only within the context of past control-flow transfers, cannot be detected by control-flow checking techniques since they are built to detect single-shot soft errors. For this survey, we will focus on techniques explicitly built for defending against various forms of control-flow redirection attacks.

4 CFI FOR EMBEDDED SYSTEMS

We now move toward more realistic CFI implementations in the context of resource-constrained embedded systems. The mechanisms presented here either combine techniques from Section 3 or propose entirely new techniques. Highlights of some of the mechanisms discussed in this section are presented in Table 1.

4.1 Implementation of Basic Techniques

We stated a pre-requisite in the prior section with respect to shadow stacks: *... Under the assumption that the attacker cannot access or modify a portion of the memory.* This assumption does not have a straightforward justification in the context of embedded systems. As previously noted, low-end embedded systems simply do not have complex memory management units to support well-known features such as virtual memory, which is now common in higher-end processors, let alone have special built-in mechanisms to support hiding shadow stacks from an attacker. Therefore, a successful CFI mechanism has to first wrangle the available hardware capabilities to support shadow stacks.

Zhou et al.'s Silhouette [97] is an attempt to support shadow stacks on ARMv7-M [49], the architecture underlying the ARM Cortex-M series of processors commonly found in embedded systems. It also supports forward-edge CFI checks. Silhouette is designed for bare-metal codebases that do not utilize an RTOS. It is, thus, an example of how a sophisticated CFI mechanism would look in the context of a resource-constrained embedded system with a bare-metal codebase.

The ARMv7-M architecture supports two privilege levels in hardware, privileged and unprivileged. The optional **memory protection unit (MPU)** allows a system designer to decide access rights to an address. A limitation of the ARMv7-M architecture is that the MPU can be controlled by any privileged code. For example, most RTOSs, such as FreeRTOS [10], by default, execute both the tasks and the operating system as privileged code to mitigate the overhead of switching privilege levels. This makes using the MPU to protect a shadow stack a moot point, simply because an attacker that has infiltrated the system could re-program the MPU since they would most likely already execute under the privileged execution context.

Silhouette ensures that the MPU access rights are adhered to by working around this limitation. It replaces all store instructions, other than those that are supposed to directly store to the shadow stack, or the **hardware abstraction layer (HAL)** code, with unprivileged store variants, at compile time, to ensure adherence to the memory access policies defined in the MPU for the target address, regardless of the processor's current execution privilege level. The shadow stack is implemented in a similar manner as the parallel shadow stack explained in Section 3.1. To ensure that the store instructions with higher privilege levels are not abused by an attacker, Silhouette implements forward-edge CFI checks. Silhouette utilizes a labeling mechanism (Section 3.2) to guarantee forward-edge CFI [14].

On the performance front, Silhouette is benchmarked using well-known embedded system benchmark suites, namely CoreMark-Pro [23] and BEEBS [69]. We will see these same benchmarks

being used in other approaches too in later sections, providing a common playing field. The maximum performance overhead reported for the two benchmark suites is 4.9% and 24.8%, respectively, and a code memory overhead of 8.9% and 2.3%, respectively. The geometric mean of the performance overheads for all the benchmarks in each test suite is 1.3% and 3.4%, respectively. The approach used by Silhouette, which they term as *store hardening*, basically utilizes a memory management technique to hide the shadow stack from the attacker.

Another mechanism that can be used to prevent access to the shadow stack is called **software fault isolation (SFI)** [64, 89]. SFI is a technique where the address space is partitioned into *fault domains*. Any code within a fault domain has unrestricted access to code or data within the same fault domain, but the partitioning scheme prevents the code from accessing any memory outside the fault domain. This is achieved by instrumenting load/store instructions during compile time to trigger the fault handler if the memory access takes place outside the fault domain. A variant of Silhouette is proposed that utilizes this technique by instrumenting store instructions to restrict them from writing to the shadow stack unless the store instruction is part of the shadow stack manipulation code. The authors note a higher performance overhead, with the geometric mean results being 2.2% and 10.2%, respectively, for the two benchmarks, which leads the authors to conclude that the store hardening approach is superior in performance. However, it would be interesting to note how the performance would vary if the shadow stack was protected using an approach similar to Aweke and Austin's [9] lightweight SFI for IoT systems that shows an overhead of just 1% on the MiBench [44] benchmarks. Their approach utilizes a small amount (150 lines) of trusted code that sets up the MPU to create the fault domains, trapping accesses outside the domain as memory access faults. Unfortunately, they do not present results using the CoreMark-Pro or BEEBS suites, making direct comparisons difficult.

While the Silhouette and its variant provide a good overview of the well-known techniques of shadow stacks and labels can be applied to a real low-end processor architecture, Kage [32] extends Silhouette to provide an implementation of CFI for an RTOS environment on microcontrollers based on ARMv7-M. Kage modifies FreeRTOS and introduces the concept of a *trusted kernel* and *untrusted tasks*. Untrusted code is passed through the store hardening compiler technique introduced in Silhouette and transformed into unprivileged store variants. This prevents their write access to the trusted portions of memory, which can only be accessed through privileged store instructions. Therefore, the trusted code such as the kernel and its associated data structures are maintained as privileged instructions so that they may access any portion of the privileged or unprivileged memory. Portions of the trusted kernel, such as common RTOS infrastructure that is expected of application tasks (locks, queues, etc.), are made available via a secure API that is designed to vet arguments from untrusted code such that they are unable to overwrite control information within the trusted kernel. The authors showcase that the Kage kernel incurs an average performance overhead of 5.2% over the baseline FreeRTOS kernel when running a multitasking workload of one to three benchmarking tasks from the CoreMark test suite.

Silhouette and Kage provide a good overview of how well-known techniques of shadow stacks and labels can be applied to a real low-end processor architecture. However, there are avenues to improve the operation of such systems. We shall now look at some of them.

4.2 Beyond the Basics

While the techniques discussed in Section 3 consider forward-edge and backward-edge separately, some effort has been applied in recent years to develop more holistic mechanisms that apply to backward- and forward-edges at the same time.

An example of such a mechanism is the **Control-Flow Locking (CFL)** technique [12]. This is also an example of a *lazy* CFI that trades off attack detection speed with performance overhead.

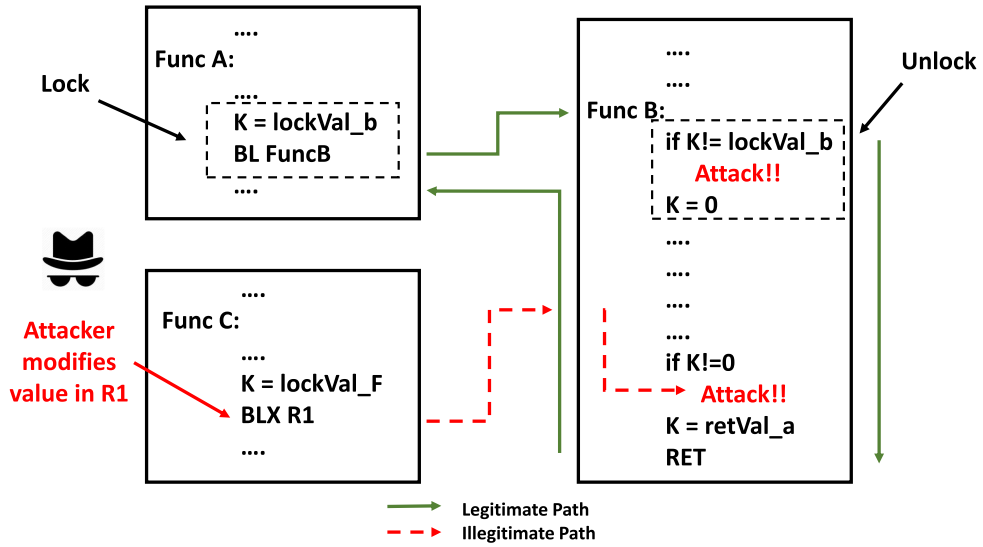

Fig. 3. Control-flow locking operation. Note the exclusive use of locks/unlocks for the entire operation (Section 4.2).

While CFL is not explicitly targeted at resource-constrained embedded systems, the mechanism can be implemented with similar memory and performance overhead as any general label-based CFI for detecting forward-edge control-flow attacks. CFL uses locks, instead of shadow stacks, to determine if an attacker has diverted control flow to an arbitrary location. An overview of the CFL operation is given in Figure 3. The idea behind the CFL approach is simple. Similar to how labels are generated based on the valid control-flow graph, *key values* are assigned to legitimate call/jump target locations. CFL targets indirect calls/jumps as well as return instructions (an x86 architecture-based processor was assumed). Once the unique key values, which essentially represent valid edges in the control-flow graph, are generated, the authors propose to then instrument the target binary with instructions to lock and unlock control-flow paths using these key values. Every legitimate control-flow redirection start point, which may be an indirect `call`, `jmp`, or `ret` instruction, is *preceded by a lock operation*; i.e., the key value is stored into a buffer. The assumption here is that the buffer is stored in a memory location such that it can be modified only by the lock and unlock subroutines, and not by attacker-controlled code. Once program control is redirected to a valid destination (such as a function entry point), it is *immediately succeeded by an unlock operation* where the key value is validated; i.e., it is checked against a list of key values that could end up at this target location. If the values match, the key is zeroed out (*unlocked*) and execution continues as before. When the next control-flow redirection operation must take place, the key buffer is first checked to see if it contains a non-zero value. If it does, an attack is detected since no legitimate transfer would allow the key buffer to have a non-zero value due to the paired lock-unlock operations. Depending on the quality of the available CFG, this pairing of lock-unlock operations could be coarse or fine.

The overall mechanism is interesting due to its simplicity and the introduction of laziness. Not only does it prevent an illegitimate jump to a *valid* control-flow transfer site, but also it automatically detects an illegitimate jump to an *invalid* control-flow site in recent history *without* requiring additional runtime memory such as using a shadow stack. Evaluations show that CFL can outperform fine-grained CFI mechanisms, with a maximum overhead of 21% vs. 31% overhead under

Abadi et al.'s [4] mechanism on the SPEC CPU2000 [2] benchmarks. However, as discussed earlier, the mechanism is lazy. This laziness can introduce blind spots that can be exploited by an attacker. For example, the attacker can redirect control and can remain undetected until it is caught by the next locking site. While laziness allows the mechanism to work with the time and memory overhead similar to a labeling scheme, it could have interesting security repercussions especially in the context of the real-time embedded systems, many of which are used in industrial environments, controlling actuators in critical processes. If an attacker is able to send out control commands to these actuators before they are detected, the attacker can still inflict catastrophic damage. However, laziness is not inherently flawed. There is therefore an avenue to leverage real-time requirements to enforce timing bounds on laziness.

While CFL is an example of a CFI technique that re-purposes control-flow labels to solve both forward and backward control-flow attack detection at the same time, it still uses a form of memory protection. All the techniques discussed up to this point attempt to work around hardware limitations to enforce memory protection and are conservative. However, they do not take full advantage of the processor architecture or require radical software/hardware changes to improve performance.

4.3 Register-based Shadow Stacks

We will now discuss two approaches that would require significant software modifications to allow them to work. We will first briefly look at Zipper Stack [59], which is the more radical of the two since it proposes CPU architecture modifications to forego shadow stacks. The other is μRAI [6], which is built for COTS embedded systems. It takes a more moderate approach by requiring reservation of parts of the CPU but can be implemented by recompiling the codebase with a modified compiler. Both implement backward-edge CFI.

Zipper stack aims to solve the problem of securing shadow stacks by replacing them with a set of processor architecture modifications. Shadow stacks, as discussed in Section 3, are inherently simple but require additional support to secure them from attacker manipulation. For example, Silhouette in Section 4.1 requires additional code instrumentation to secure the shadow stack. Zipper Stack aims to solve this problem by replacing the shadow stack with a single value stored in a special-purpose register called the *top register*. A separate register, the *key register*, holds a secret key. At the start of a new process, the key register and top register are initialized with random values. Each time a function call takes place, the top register is pushed onto the main stack alongside the actual return address. A **message authentication code (MAC)** algorithm, a cryptographic operation that is commonly used to authenticate messages from a known source, generates a new MAC from the top register value and the return address using the key in the key register. This newly created MAC is then stored in the top register. During a return sequence, the steps are reversed to authenticate the return address. First, the previous MAC value is popped from the stack and the MAC is recalculated using the return address and the popped MAC value. If the calculated MAC matches that currently in the top register, the return address is verified to be authentic. The processor replaces the top register with the popped value and continues execution at the return address. The purpose of the MAC-based design is to reduce the attack surface. By utilizing the top register and chaining the MAC values with each successive function call, an attacker can only modify the return address and evade detection if it first modifies the value present in the top register (which is inaccessible to application code and is automatically updated by the hardware) before modifying the other MACs. Therefore, the rest of the MACs can be kept in non-secure memory that may be accessible to the attacker, reducing the amount of overhead introduced by accessing the "zipper stack" of MAC addresses.

The operation shows that Zipper Stack is heavily dependent on (a) the efficacy of the MAC algorithm to ensure *collisions* (same MAC from different inputs) do not occur, (b) the speed of the algorithm since every function call would constitute running the algorithm at least twice, and (c) the attacker not being able to access the key register to forge MACs. For (a) the authors use a well-known MAC algorithm, for (b) the authors argue that a hardware implementation would allow MAC calculation in a single cycle, and for (c) the authors argue that even if the key is leaked, the top register can only be modified at a call or a return operation. Their custom implementation on an FPGA with a RISC-V CPU achieves a 1.86% overhead on the SPEC CINT 2000 [2] benchmark.

While Zipper Stack presents a very radical approach that may never see wide-scale commercial adoption due to its hardware modifications, it is still interesting since custom architectures for specific applications, such as defense, are not uncommon in the embedded system world. In such cases, a custom architecture designed with optimized built-in defense mechanisms is not hard to envision. Interestingly, the use of MACs for authenticating return addresses may become possible very soon on commodity hardware. For example, PACStack [60] re-purposes the ARM `pac` instruction to create a MAC chain of return addresses, very similar to Zipper Stack. As part of the ARMv8.3-A PA extension, and soon to be available on SoCs based on ARMv8.3-A and later architecture revisions, `pac` allows generating **pointer authentication codes (PACs)**, which are MACs generated on pointer values and stored alongside the pointer. Similar to Zipper Stacks, the authors use a *chain register* to store PAC values, which are generated from previous chain register values and the return address of a function call. When a return sequence takes place, similar to Zipper Stack, the reverse operation takes place. PACStack showed a geometric mean of 2.75% and 3.28% performance overhead on the SPECrate and SPECspeed (part of the SPEC CPU 2017 benchmark suite), respectively. PACStack provides a strong argument for MAC-based shadow stack replacement, especially since it depends on architecture extensions, which will soon be available in commodity hardware.

On the other hand, the authors of μRAI take a similar but more realistic approach, especially on current-generation hardware. μRAI is also concerned solely with the backward-edge, but instead of verifying the return address as is common with shadow stack approaches, μRAI enforces **Return Address Integrity (RAI)**, where the return address simply cannot be modified by an attacker. Their approach, in essence, is to prevent write access to the return address. μRAI has the same set of requirements as many of the schemes we have discussed in previous sections, such as **data execution prevention (DEP** or $W \oplus X$) and an MPU. Similar to Zipper Stack, it requires that one of the processor registers is wholly dedicated to its operation and should never spill. This is called the ***State Register*** **(SR)**. μRAI's operation requires that the attacker cannot modify the register.

μRAI works by instrumenting code before branches and at return points, similar to CFL. It works solely with direct branches, i.e., branches with encoded destinations, and converts all indirect branches into direct branches by matching all possible start and endpoints. Figure 4 provides a basic overview of how μRAI instrumented code looks and operates. Every function *call site* is assigned a unique **function key (FK)**. As is seen in the figure, Function A can have multiple call sites to another Function B. μRAI instruments code such that before every such call site, the value in the SR register is XOR'ed with the FK for the call site. This value is also called the **Function ID (FID)**. The call goes through and Function B operates. At the point where Function B returns, it checks what the authors call the **Function Lookup Table (FLT)**. This table has all the FIDs that could call this function. Based on which FID matches the value in the SR, the function returns to the corresponding location. Finally, the SR is XOR'ed with the same FK used before the branch, returning it to the original value before the function call. The authors tested their approach on an ARM Cortex-M4-based board and report a maximum performance overhead of 8.1% on the

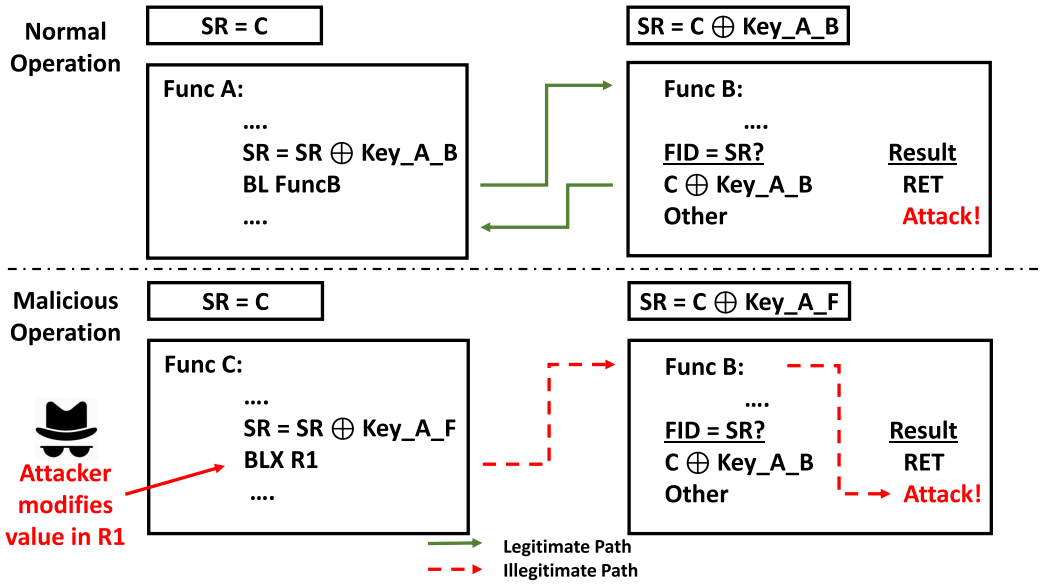

Fig. 4. μRAI operation. Shadow stack operation is implemented via SR register and FID table during return (Section 4.3).

CoreMark [41] (a lighter variant of CoreMark-Pro) benchmark with an average of just 0.1%, making it comparable with shadow stack mechanisms discussed previously. However, it requires on average 34.6% extra flash memory for instrumentation and FLT.

The reader may have noticed that the possible return addresses are encoded into the code memory under DEP restrictions that prevent an attacker from modifying the code memory. DEP is enforced using the MPU. μRAI, therefore, foregoes the return address that the processor may record in its stack, which is inherently writable memory, during a function call. Instead, it implements a function return mechanism that is implemented completely in code memory. This enforces μRAI's goal of return address integrity. μRAI is also the first mechanism that we have discussed in this survey that explicitly considers interrupts. Since interrupts can occur at any time and can potentially interfere with shadow stack operations, they require explicit consideration. μRAI instruments interrupt handler code to first save the return address that has been automatically stored on the stack by the hardware before the handler code is executed. μRAI saves the return address to a safe memory hidden behind the MPU. Here μRAI has to essentially create a shadow stack due to the limitation of the hardware. Supporting interrupts is a significant step to eventually supporting multi-threaded scheduling under a **real-time operating system (RTOS)**. However, dedicating a register to μRAI operations would require modifications to the compiler as well as incompatibility with embedded systems having a severely limited processing capacity, especially when the software requires a large number of registers for computational purposes.

Unfortunately, none of these techniques improve forward-edge CFI. For example, in the case of μRAI, the attacker could keep redirecting code execution using branch operations without allowing code to execute till an FID table. Therefore, such CFI mechanisms are helpful from only a performance or memory perspective over a regular shadow stack. That is, they do not provide any additional security guarantees, while requiring significant codebase changes or at least a modified compiler to support their operation.

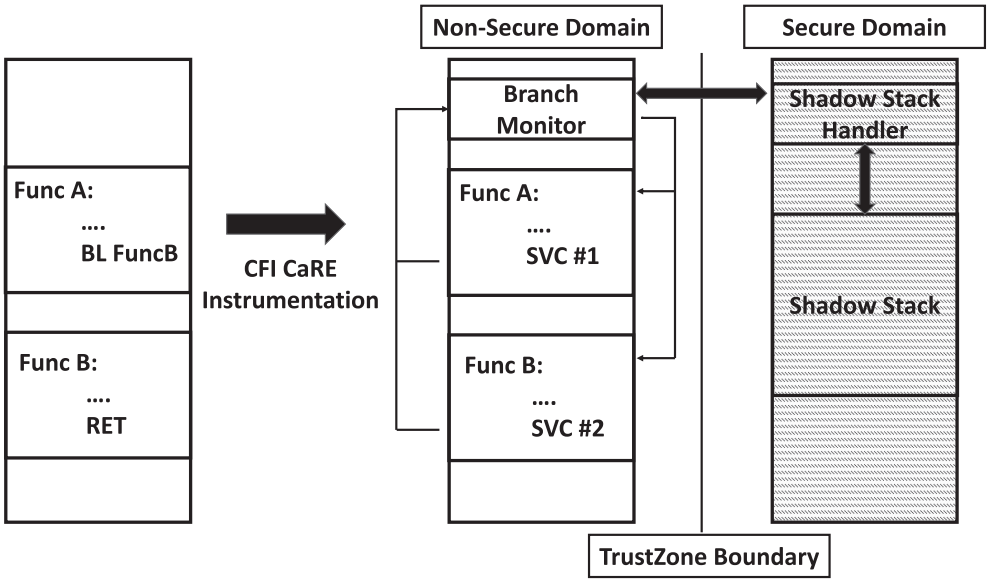

Fig. 5. CFI CaRE instrumented code. Branch and return instructions translate to SVCs (Section 4.4).

4.4 CFI Using Processor Architecture Extensions

Before we finally move toward real-time aware CFI mechanisms, we will look at two mechanisms that depend on very modern processor architecture extensions such as ARM TrustZone [70]. TrustZone allows a processor to support two execution domains, *secure* and *non-secure*, each with its own address space, with the secure domain having supervisory access to the non-secure domain. CFI designers have found creative ways to use it as part of their designs.

The first is Nyman et al.'s CFI CaRE [67], which presents an alternative approach to secure the shadow stack to that of Silhouette 4.1. An overview of its operation is given in Figure 5. While Silhouette uses binary instrumentation to prevent a privileged attacker from modifying the MPU that hides the shadow stack, CFI CaRE hides the shadow stack behind the TrustZone in the secure domain. CFI CaRE assumes that the original binary is only allowed to execute under the non-secure domain. It replaces all function calls with a ***supervisory call* (SVC)** that launches a special function called the *branch monitor*. The branch monitor runs in a privileged context, and based on the parameter passed to the SVC that launches it, the branch monitor is able to identify if the source of the SVC is a branch or a return. It then calls secure domain code, passing the source identifier as a parameter that updates the shadow stack. While the SVC ensures that all branches and returns are effectively trapped in the branch monitor, the TrustZone boundary ensures that non-secure domain code cannot view or modify the shadow stack. The authors used the Dhrystone (precursor to CoreMark) benchmarks to evaluate their work on an ARM Cortex-M23 processor. Performance overhead ranged between 13% and 513% with an overall 14.5% increase in flash memory consumption.

While CFI CaRE may seem like just a different implementation from previous approaches, it proposes a mechanism to address a crucial flaw in previous approaches with respect to embedded systems. The previously discussed approaches instrument binaries with no regard to the original layout. While this may be a non-issue for systems whose source code is available, many real-time embedded systems use proprietary legacy software and access to the source code may be limited. Further, due to memory and processor restrictions, these binaries are painstakingly built with strict

adherence to page limits, available flash memory, and so forth. Unchecked binary instrumentation may destroy compatibility with the hardware. CFI CaRE's usage of SVC simply overwrites the branch or return instructions, keeping the original binary layout intact. However, it does require extra space for the branch monitor.

CFI CaRE also supports interrupts and uses *trampolines,* which are short sequences of code at the start of interrupt that call the secure domain to store the return address in a shadow stack. However, it does not support nested interrupts. If an attacker-controlled higher-priority interrupt fires before the trampoline can store the return address in the shadow stack, the attacker-controlled interrupt code could rewrite the return address. When the lower-priority interrupt finally gets to run, its trampoline would store a modified return address. Furthermore, nested interrupts can occur on an RTOS-controlled system. For example, the timer tick could fire alongside interrupts from other peripherals. Kawada et al.'s [54] TZmCFI fills this gap. They too propose using the TrustZone to hide the shadow stack. However, they also extend the shadow stack concept to what they term as *exception shadow stacks* that support nested interrupts. They modify the trampolines such that every trampoline will complete all pending shadow stack transactions of lower-priority interrupts before the interrupt body is allowed to execute. This ensures that if an attacker controls the interrupt body, it cannot affect the shadow stack copy of the interrupt return address. TZmCFI showed a performance overhead of up to 84% when supporting FreeRTOS as compared to FreeRTOS without CFI. For nested interrupts, the instrumented interrupts (with the trampolines) increased interrupt execution time from 30 cycles (un-instrumented) to 132 to 236 cycles, i.e., up to a 550% increase in execution time.

Other work that involves extending the architecture of the processing environment includes Intel's **Control-Flow Enforcement (CET)** [53] architecture extensions in their recent Tiger Lake [87] processors. The CET extensions provide hardware support for shadow stacks and forward-edge CFI. Due to their recent introduction in production hardware, there is a lack of prior CFI work that builds upon CET. Further, the Tiger Lake processor family are powerful desktop-grade processors, which are outside the scope of this work, which focuses on embedded systems (see definition in Section 1.2.1). Similar in concept to CET, the authors of HCFI [21] suggest creating a new CFI-enabled **instruction set architecture (ISA)** by modifying an existing ISA such as SparcV8's Leon3 [40]. They do so by adding new stages in the CPU pipeline to perform CFI operations such as shadow stack operations and show that performance overhead with respect to an unmodified Leon3 core is less than 1% on their FPGA implementation for the SpecInt2000 benchmarks. While optimum performance can be achieved by extending the processor architecture and/or designing custom processor cores, it remains to be seen if such extensive hardware modifications are feasible for the more resource-constrained processing environments of embedded systems. Until such a time, the TrustZone-based approaches discussed earlier are more realistic.

4.5 CFI Using Separate Processing Environments

We wrap up our discussion of different CFI mechanisms for embedded systems with a brief note about CFI by utilizing off-chip processing environments since they behave very similarly to CFI achieved via TrustZone and utilize the same set of techniques presented in detail in Section 3. For example, techniques such as Abad et al.'s [3] use a separate monitoring module to track the program counter and detect deviation from the control flow. Similarly, SecMonQ [66] is designed for automotive systems and utilizes the **Hardware Security Module (HSM)** found in many commercial automotive ECUs to detect anomalous path behavior. In a more general sense, techniques such as RTTV [93] utilize the **Trusted Platform Module (TPM)**, a common co-processing environment used as a store for cryptographic keys and performing a limited and static set of cryptographic operations in many embedded systems, can be used to store the CFG and perform regular

measurements against the stored CFG. All these techniques inherit and apply the basic techniques presented in Section 3.

4.6 Section Summary

The techniques discussed in this section generally follow the basic techniques listed in Section 3. The proposed mechanisms either directly apply those basic techniques or have progressively complex hardware modifications, from special registers to reduce the cost of shadow stacks (Section 4.3) to novel ISA (Section 4.4). However, the techniques do not inherently change the underlying principles of CFI and can be *conventional* by their nature. That is, they all verify the source and target destination addresses without much variation. Another important observation is that each of the techniques presented is uniquely tied to the underlying hardware for both performance and enforcement of CFI, making it difficult to compare their individual overheads. However, on a qualitative note, it is clear that the most performant CFI requires radical hardware changes, such as integrating shadow stack operations into the pipeline of the processor [21].

A common theme in the techniques discussed, however, is the lack of any discussion regarding the implications of the overhead they introduce on systems where timing is critical, e.g., real-time systems. Real-time systems have certain characteristics that could be utilized to aid CFI and/or reduce the impact of the overhead introduced. We will now discuss these characteristics:

(1) In periodic real-time systems, work is performed in a temporally predictable manner. That is, tasks execute during defined periodic intervals. CFI could utilize this predictable periodic nature to determine if an application is misbehaving due to attacker control.
(2) The system is usually underutilized due to safety requirements. Since real-time systems are, in many cases, deployed in critical environments such as medical, industrial, or automotive systems, such systems are designed to not perform work all the time to reduce or eliminate the possibility of missing deadlines. For example, the system is usually provisioned with enough computing resources such that tasks do not need to consume 100% of the computing resource at all times to complete by their deadlines. Therefore, the system may have large periods of idle times. CFI could utilize the idle time, thereby reducing localized spikes in computational load and reducing the possibility of missing deadlines. Note that although these systems may be underutilized, they are still considered to be resource-constrained. The underutilization is intentional due to safety concerns and any addition in the computational requirements must be done judiciously.
(3) The total system utilization at any given point of time is usually well characterized and there exist schedulability tests to determine if the system may be successfully scheduled without missing deadlines under a given scheduling algorithm. These tests may differ for different types of real-time task models (periodic tasks, aperiodic tasks, etc.). None of the techniques discusses their applicability and/or changes that must be introduced to satisfy these schedulability tests.

None of the techniques discussed in Section 4 consider timeliness. We now discuss CFI works that are specific to real-time embedded systems.

5 CFI FOR REAL-TIME EMBEDDED SYSTEMS

We have discussed multiple CFI techniques in the previous section for embedded systems. In this section, we survey the state-of-the-art mechanisms that consider real-time requirements. Unfortunately, there is little prior work that explicitly considers real-time properties of the system's operation. Therefore, this section discusses a few available CFI mechanisms. We divide our discussion into two parts; the first part covers techniques that are built specifically with an RTOS

scheduler in mind, and the second discusses non-conventional CFI approaches. Highlights of the mechanisms discussed in this section are presented in Table 1.

5.1 CFI with an RTOS

5.1.1 An Analytical Approach for Common CFI Techniques. TZmCFI, presented in the previous section, is an example of CFI mechanisms for embedded systems that can work alongside an RTOS or, more specifically, a scheduler. A scheduler consists of supervisory code that decides when code that does actual work, i.e., complete the goal of the system, is able to run. A scheduler is critical to ensure system timeliness. While TZmCFI supports an RTOS, it lacks a study of system schedulability under different workloads. The recent work by Walls et al. [90] addresses this deficiency in research. Their approach, called *RECFISH*, is an RTOS-aware CFI scheme. Since RECFISH shares several similarities with techniques discussed in prior sections, we will briefly discuss the mechanism and take a closer look at the evaluation results.

RECFISH is designed for ARM Cortex-R [61] processors that are built specifically for critical real-time applications. Like the Cortex-M series, they forego memory management units and have special caching mechanisms to maintain predictability and support a small address space, but do not support TrustZone. RECFISH, instead, utilizes the MPU, like μRAI, to enforce DEP. It assumes that the task code executes in the unprivileged mode while the RTOS runs in privileged mode. This ensures that if an attacker infiltrates a task, it cannot override the MPU settings. RECFISH is designed to be used with FreeRTOS and modifies it to allow setting up a per-task shadow stack (which only privileged code, such as the RTOS, can modify since it is hidden by the MPU), and modifies the scheduler to update the shadow stack when switching between tasks. Finally, RECFISH also instruments the binary to add labels to function prologues, as well as enforce shadow stack operations before (and after, in the function epilogue) the function body can execute. The labeling mechanism is used for enforcing forward-edge schemes, while the shadow stack operations are enforced by calling privileged shadow stack handling code using SVC just like that seen in CFI CaRE.

While the operation of RECFISH may look similar to multiple ideas presented in previous sections, the authors are the first to present a study of their approach's effect on real-time workloads. They evaluate and note a 21% performance overhead for their approach on the CoreMark benchmarks. Microbenchmarks show that RECFISH increases scheduler context switching time from 120 CPU cycles to 159. Further, the label checking and shadow stack operations increase function prologue and epilogue overheads from 19 cycles (without any CFI operation) to 275 cycles. The authors then perform a large-scale schedulability study on simulated workloads. They randomly generated synthetic task sets with varying utilization values, task periods, and number of indirect branches. Utilization values ranging from 0.1% to 90% were considered. The overhead of task context switch (39 cycles) was incorporated into the task's **worst-case execution time (WCET)**. For incorporating the function prologue and epilogue overheads (label checking for forward-edge CFI), the authors considered a varying number of indirect branches per task that were either 0 or ranging from 1 every 10^3–10^5 cycles to 1 every 10^6–10^7 cycles. Multiplying the number of branches with the 256-cycle overhead for the task yielded the overhead for the label checking mechanism, which was then incorporated into the task WCET. RECFISH performs well for task sets where the number of tasks is few and each task has a high utilization, and when indirect branches are infrequent. However, the results show that up to 30% of the system utilization can become unusable for task sets with more frequent indirect branches and function calls and more tasks. Overall, RECFISH could schedule 85% of the 6 million task sets generated from 5,760 different parameter combinations. The results show that well-known CFI mechanisms such as shadow stack and labeling could be used with a wide range of multi-threaded real-time workloads.

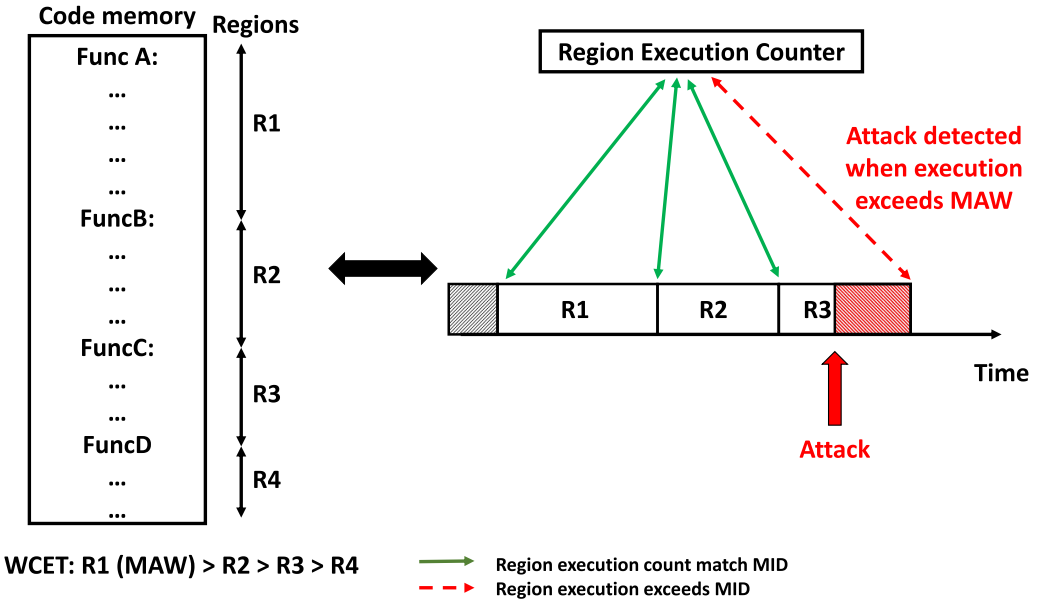

Fig. 6. Utilizing execution time (MID) as a metric to determine control-flow attacks (Section 5.2).

5.1.2 Trade off Security for Schedulability. While RECFISH provides a schedulability study of common CFI techniques, Hao et al. [45] provide a novel technique to improve the schedulability of a real-time system by trading off security with system schedulability. They focus on defending against ROP attacks (Section 1). They do so by selectively switching on CFI checks for a subset of instances (also called jobs) for each task in the system by exhaustively searching for the maximum set of jobs that can have CFI checks without hindering the schedulability of the system. The authors provide a comparison of an approximated scheduling algorithm that is designed to be faster to execute during runtime with respect to the exhaustive search algorithm, which is determined to be optimal. Experimental results show that their approximation approaches optimality at lower ($\leq$0.6) utilizations. A schedulability study shows that there is a sharp drop-off in schedulability of task sets if the CFI checks are added to task sets with utilization greater than 0.8. This observation echoes the results of the study of RECFISH that as task sets become "heavier," that is, have a higher utilization, schedulability sharply drops down to zero.

5.2 CFI Utilizing Timing Deviations

5.2.1 Utilizing WCET. While RECFISH implements well-known CFI techniques, Bellec et al.'s [11] proposal utilizes the predictability of real-time systems to detect control-flow violations. An overview of the approach is provided in Figure 6. Their approach is based on the simple idea that an attacker will cause a control-flow violation to perform some malicious action. This will undoubtedly cause an increase in execution time, over and above the execution time of the system's tasks. Since real-time systems have well-defined task timing parameters, it is within reason to expect that an attacker-controlled execution would show a marked increase in execution time. A monitoring mechanism could, theoretically, detect such an increase and expose an attacker. The authors are able to support such a mechanism by first splitting the code base, consisting of a single task, into *regions*. Regions are either non-overlapping or located entirely within another region. Since the WCET of the task is known, each region within the task code is assigned a WCET

of its own, called the **maximal inner duration (MID)**. The MID of a region does not include the MID of a sub-region. Therefore, the sum of MIDs of all regions covering a task's code would equal the task's WCET. The authors define another metric called the **maximal attack window (MAW)**. For a set of monitored regions, the MAW is the maximum MID of that set. Therefore, the goal is to find the best possible set of regions such that (a) the entire task code is covered and (b) the MAW is minimized. The authors perform a search, bounded by the available memory to store region boundaries as well as performance metrics during runtime, to find the best possible set of regions. To evaluate their approach, the authors propose a custom hardware architecture that can detect when code execution enters and exits a region, as well as keep constant track of the time the processor spends within a region. If the time spent exceeds the MAW, an attack would be detected. The authors utilized two benchmark suites, Mälarden, and Polybench. They found that their approach had a mean latency of 95% (maximum of 99%) of the MAW before it detected an attack, where the MAW sizes ranged from a few hundred up to over 160,000 CPU cycles. However, they found that their approach calculated MAWs of 600 or fewer CPU cycles for half of the benchmarks.

Due to the detection latency, this approach has similar issues as those that utilize laziness; specifically, the attacker could damage the system before it is detected. Further, it requires extensive modifications to the architecture to support it. However, it presents an interesting starting point for CFI mechanisms that effectively utilize the predictability of a real-time system to inform their approach.

5.2.2 Timing Code in Hard Real-time Context. We end our discussion of the state-of-the-art CFI for real-time systems with Abbasi et al.'s [5] ECFI. ECFI is built for **Programmable Logic Controllers (PLCs),** which are commonly found as the computing units for industrial-control systems. ECFI is a middle-ground approach, utilizing coarse-grained or fine-grained (depending on whether the code has pointer-based calls) CFI as well as exploiting the high predictability of the typical hard real-time system where PLCs serve as computational units, to detect if an attack causes a sudden increase in execution time to warrant the need to perform CFI checks. ECFI operates by capturing control-flow data in a global shadow stack during system execution and then checks the data in a low-priority process. ECFI presents an amalgamation of traditional CFI techniques and utilization of predictability of the time domain.

Note that there are related techniques to improve the schedulability of security mechanisms in general, such as Hasan et al.'s Contego framework [46] that introduces the concept of abstract security tasks into the system, but such techniques are not specifically designed for CFI and are not directly compatible with any of the work presented in this section.

5.3 Section Summary and Observations

Our discussion of CFI techniques for real-time embedded systems is summarized in Table 1. In general, we see a lack of techniques that consider timing constraints. While prior work has explored applying, with varying degrees, timing constraints to improving CFI schedulability, there is still clear room for exploring this domain. For example, none of the techniques presented considers overloaded system conditions or utilizes timing to amortize the cost of CFI in such situations. For example, a periodic real-time system has well-defined intervals of slack. By deferring CFI operations to these slack intervals, it would be possible to reduce the effective in-line overhead that the CFI operation introduces while executing the system application, an observation we also state in Section 4.6. In our survey of CFI techniques for real-time systems, we have not found any technique that capitalizes on system slack in this manner. On the other hand, Hao et al.'s technique in Section 5.1.2, while useful to reduce the cost of CFI to maintain schedulability, can be considered

Table 2. A Summary of Techniques Discussed in Depth in Sections 4 and 5

Category	Technique and Summary
Implementation: Standard CFI techniques on different architectures	1) Silhouette - Shadow stack and binary labeling on ARM Cortex-M 2) RECFISH - Shadow stack and binary labeling on ARM Cortex-R
Design Changes: Non-standard CFI techniques utilizing standard control-flow start and end points	1) Control-Flow Locking - Lazy control-flow evaluation. Single technique for forward- and backward-edge 2) uRAI - Collapse shadow stack into a single register using XOR operations. 3) Zipper Stack - Custom hardware to collapse shadow stack into a single register via HMAC operations
Modern hardware architecture: Techniques that utilize new processor architecture features	1) CFI Care - Shadow stack hidden by ARM TrustZone 2) TZmCFI - Nested interrupts (RTOS)-aware shadow stack in ARM TrustZone. 3) PACStack - ARM pointer authentication (ARMv8.3-A) utilized for collapsing shadow stack in single register
Underlying Principle: CFI techniques that detect control-flow deviations using non-standard principles	1) Timing deviation - Detect WCET violation of code segments using custom hardware 2) ECFI - Built for PLCs. Detects timing violations code during runtime

incomplete in terms of security since only a subset of the code executed at runtime is actually checked. This could be exploited by a smart attacker, especially one aware of the technique used to decide which jobs do not have CFI checks. Some mitigation could be provided by randomizing the schedule using techniques such as using Yoon et al.'s TaskShuffler [94], but even such works have been shown to be defeated by carefully crafting an attack [65] that defeats the randomization. Essentially we do not see novel techniques that successfully use real-time constraints to amortize the cost of a *complete implementation* of CFI for real-time systems. Bellec et al.'s approach could be considered as a good starting point for creatively using timing constraints; however, it has its own failings, which we discuss in Section 5.2.1.

6 SUMMARY AND OPEN CHALLENGES

For convenience, special terminology/mechanism names that have been discussed before are listed here alongside the relevant section in the article:

Silhouette - Section 4.1, **Lazy** - Section 4.2, **Timing deviation** - Section 5.2, BBB-CFI - Section 3.2, **RECFISH, ECFI** - Section 5.1.1, **Context-sensitive** - Section 3.2

A summary of our discussion in prior sections is presented in Table 2. Some common themes and omissions in the techniques presented are:

(1) Most prior CFI work utilizes some form of software-hardware bypass to accommodate hardware constraints present in resource-constrained embedded systems. The techniques trade off performance and security to create the best possible compromise for their target hardware architecture.
(2) The wide variety and heterogeneity of embedded system hardware make it difficult to all but qualitatively compare techniques in terms of memory and performance. Many require the use of custom/bespoke hardware architectures such as Zipper Stack (which requires a custom HMAC and special registers to speed up CFI). It is, therefore, difficult to judge if one technique is better. The applicability of any of the approaches we list for embedded and real-time embedded systems is dependent on the target application. We therefore only provide some qualitative discussion and summary, especially for the techniques discussed in depth for embedded systems in Section 4 to aid the reader.
(3) With the exception of Bellec et al.'s work [11], the design of *conventional* CFI for embedded and/or real-time embedded systems can primarily be viewed as *memory-based*, where CFI is performed by detecting deviations from expected instruction memory accesses. There is no fundamental difference in the detection methodology across all the presented techniques.

(4) Real-time CFI mechanisms, other than techniques such as that presented by Bellec et al.'s work [11], are *ad hoc* in design. None of the techniques seems to utilize the strict timing requirements of the system to aid CFI. CFI, in essence, is detecting deviations in system behavior, and timing critical systems depends heavily on being temporally correct. However, utilizing temporal guarantees exclusively to detect abnormal behavior can reduce the effectiveness of the mechanism as we discuss in Section 5.2. In fact, for a real-time embedded system, some assumptions can be made (we will discuss a possible approach later in this section) that can synergistically aid conventional CFI and improve its performance.

In this section we first present open challenges to the real-time community based on our understanding of the state of research in CFI for real-time embedded systems. We also present general consideration points that have not yet been incorporated into CFI designs.

6.1 Real-time Challenges

We believe there exist two broad avenues of research that could be undertaken immediately, considering the state of the art.

Bounded Laziness: CFI designs for real-time systems are few in number and do not seem to capitalize on system predictability. In particular, laziness, such as that introduced by control-flow locking, is a promising mechanism for hard real-time systems due to its ability to defer CFI checks. However, a drawback of their approach, and Bellec et al.'s timing deviation-based mechanism (Section 5.2), is the lack of expressiveness in the threat model, specifically, the time at which an attacker is able to affect the system. For example, the proposed mechanisms fail to consider that an attacker could modify and produce system outputs, such as sending messages via a network controller to other systems, before the CFI mechanism detects an attack. On the other hand, conventional CFI techniques have an unnecessary sense of *urgency* since CFI is performed as close to when the control-flow path changes as possible. For example, the mechanism presented in Silhouette adapts well-known CFI techniques that all perform CFI during a control-flow transfer event. We believe there is a middle ground that can improve performance and still maintain the *usefulness* of CFI. That is, the purpose of CFI, to detect an attacker before they are able to damage the system, is still maintained. This is because real-time systems inherently have discrete and well-known time instances where they must generate system output. For example, in a typical industrial control system, an I/O controller [26] may scan for sensor data periodically. In such a setting, there is no need to *urgently* perform CFI on the sensor task code execution, and the CFI work can be deferred. That is, any control-flow transfer events that may occur can be recorded and can be verified at a later stage before any actuator commands (or some other form of system output) are sent out. Since all current techniques introduce CFI in-line during execution, effectively inflating task WCET, which may cause overload situations rendering certain task sets unschedulable, deferring CFI could possibly avoid such WCET inflation and increase its acceptance in more real-time systems as more deadlines can potentially be met. However, such techniques would possibly require a record of control-flow events such as the addresses of the start and end points of a control-flow transfer, thereby increasing memory usage depending on the granularity of the CFI. For example, a fine-grained CFI would consume more memory to record every control-flow transfer event. Capturing and quantifying such memory-timing-security tradeoffs in real-time systems is an open problem and should be investigated.

Multi-thread/Core Scheduling: RECFISH and ECFI showcases the applicability of well-known CFI techniques to multi-threaded hard real-time systems. We believe there is an opportunity to extend the concept of bounded laziness to multi-threaded/multicore systems and utilize available multicore real-time scheduling theory for increased parallelization [78]. In the case of multi-core scheduling, a number of cores could be dedicated to performing CFI operations. Note

that there is prior work that considers arbitrary security operations as security tasks and explores their schedulability in multicore real-time systems [46, 47]. However, such works do not explicitly consider temporal bounds for completing security operations. From a security perspective, ECFI implicitly trusts the scheduler's integrity. However, in advanced threat models where an attacker could have the privilege to disrupt scheduler operations, such as modifying the system timer to warp the scheduler's sense of time, such defense mechanisms could fail. Prior work to secure time sources, such as TimeSeal [7], could provide some inspiration to solve this problem.

Determining CFI-related Workload Attributes: Addressing the previous challenges would also require determining the real-time properties of CFI operations, such as the WCET of CFI operations, or how CFI operations would be incorporated into other real-time models such as those that consider varying task periods [55] and systems with mixed real-time tasks (e.g., a system with both periodic and aperiodic tasks [83]), and so forth. The WCET of CFI too could be difficult to accurately determine especially if the mechanism operates on historical control-flow data, such as in context-sensitive CFI, where the amount of data can vary during system operation.

6.2 General Challenges

In addition to the real-time system-specific challenges listed above, there are some general considerations that should be incorporated into future designs. The following challenges are not just limited to CFI mechanisms but the system security research in general.

Power Consumption: An often overlooked component of embedded system development is power consumption. This is also evident in every CFI design reviewed in this article. None of the mechanisms considers power consumption, which is especially important in embedded systems operating off batteries and deployed in the field. Some designs such as that provided by Das et al. [28] provide power consumption measurements of their custom control-flow checking hardware design implemented on an FPGA. However, such measurements are an exception rather than the norm with respect to CFI research. Custom designs presented by other work such as Zipper Stack [59] do not provide information regarding power consumption, making it difficult to decide the applicability of such work to severely power-constrained and hard real-time environments such as heart pacemakers. Since CFI techniques such as shadow stacks have high memory access rates, the impacts on system power consumption of different techniques must be considered.

We believe that, alongside real-time scheduling theory, techniques such as **Dynamic Voltage Frequency Scaling (DVFS)**, backed by an extensive pool of scheduling algorithms that utilize DVFS [76, 77], could provide significant reduction in system power consumption and interesting schedulability issues. Interestingly, a logical correlation can be made between coarse-grained CFI and reduced power consumption, by virtue of reduction of CFI checks that are required due to the coarseness of the design. A study on the relation between coarse CFI and power consumption of design on commercial off-the-shelf hardware could have an immediate impact within the research community, providing researchers guidance on which type of CFI and what aspects of CFI design have the worst effects on power consumption. We also believe that a new class of schedulability-power co-design problems could arise from utilizing laziness in CFI to limit the peak power consumption of a system by carefully differing CFI to low-power consumption phases of the system.

The goal of CFI is similar to that of system reliability improvement techniques, i.e., to prevent incorrect execution and/or detect when incorrect execution occurs. There is a large amount of prior work that discusses mechanisms to implement recovery schemes with minimal impact to system power consumption. Such work could be used as inspiration to create energy-aware CFI mechanisms.

Portability: In general, CFI for resource-constrained embedded systems adapts well-known CFI techniques such as shadow stacks and labeling to such systems while working around their limitations. A primary observation is that many of these workarounds are very specific to the hardware platform that the authors target. For example, Silhouette targets ARMv7-M and therefore modifies store instructions to use the MPU on this architecture. Since these limitations are hardware-specific, designing realistic CFI mechanisms for such systems that are also portable is difficult. Unlike desktop or server-grade hardware, where commodity systems usually include processors with similar underlying architecture, embedded systems utilize architectures from ARM, RISC-V, MIPS, and so forth as well as application-specific designs. Designing a one-size-fits-all mechanism for such a wide range of target architectures is a difficult challenge. Further, architectures such as ARM are very modular, allowing hardware vendors a high degree of flexibility to add or remove features to adjust manufacturing costs and provide a wide portfolio of devices at every price point. There is, thus, a need to design feasible CFI mechanisms that operate completely in software (or with minimal hardware requirements), to allow for portable designs. However, the overhead of such designs remains to be seen.

Advanced CFI and Beyond: As discussed in Section 3.2, there is a need to consider context sensitivity in real-time embedded systems to thwart attacks that can bypass even fine-grained CFI. We are not aware of the existence of such techniques. Finally, there is a gap in research for embedded and real-time embedded systems regarding state-of-the-art **data-oriented programming (DOP)** [50] attacks. These do not redirect control flow but attack program data, such as the counter variable used for a loop. Such attacks cannot be mitigated using any of the CFI designs discussed in this article since they do not cause deviations in the control-flow path. Note that techniques such as timing deviation detection discussed in Section 5.2 may be able to detect such attacks, but the assumption here is that the attacker is knowledgeable and does not violate the MAW during an attack. Data-oriented attacks are powerful and have been shown to be capable of influencing program output as well as disclose private information.

7 CONCLUSION

We have examined multiple CFI schemes in the article, from the core mechanisms that help enforce CFI to the necessary workarounds required to support them in resource-constrained embedded environments. We have also looked at the modifications necessary to support real-time schedulers and how real-time characteristics can be effectively utilized for CFI. While CFI has been adopted by higher-end systems, designs for resource-constrained embedded systems are still mostly academic and not yet widely deployed due to unmanageable performance overhead in some cases. As we have seen, CFI will undoubtedly have overhead due to hardware constraints, but techniques such as laziness that trades off detection speed with overhead could provide an interesting avenue for future work.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] *Clang 12 Documentation.* 2020. Retrieved October 24, 2020, from https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[2] Standard Performance Evaluation Corporation. 2020. https://www.spec.org/benchmarks.html.

[3] Fardin Abdi Taghi Abad, Joel Van Der Woude, Yi Lu, Stanley Bak, Marco Caccamo, Lui Sha, Renato Mancuso, and Sibin Mohan. 2013. On-chip control flow integrity check for real time embedded systems. In *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA'13).* IEEE, 26–31.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.
[5] Ali Abbasi, Thorsten Holz, Emmanuele Zambon, and Sandro Etalle. 2017. ECFI: Asynchronous control flow integrity for programmable logic controllers. In *Proceedings of the 33rd Annual Computer Security Applications Conference*.
[6] N. S. Almakhdhub, Abraham A. Clements, S. Bagchi, and M. Payer. 2020. μRAI: Securing Embedded Systems with Return Address Integrity. In *2020 Network and Distributed Systems Security (NDSS) Symposium*.
[7] Fatima M. Anwar, Luis Garcia, Xi Han, and Mani Srivastava. 2019. Securing time in untrusted operating systems with TimeSeal. In *2019 IEEE Real-Time Systems Symposium (RTSS'19)*.
[8] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A survey. *Computer Networks* 54, 15 (2010), 2787–2805.
[9] Z. B. Aweke and T. Austin. 2018. uSFI: Ultra-lightweight software fault isolation for IoT-class devices. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE'18)*. 1015–1020. https://doi.org/10.23919/DATE.2018.8342161
[10] Richard Barry et al. 2008. FreeRTOS. https://www.embeddedrelated.com/Documents/190225_FreeRTOS_Embedded_World.pdf.
[11] Nicolas Bellec, Simon Rokicki, and Isabelle Puaut. 2020. Attack detection through monitoring of timing deviations in embedded real-time systems. In *32nd Euromicro Conference on Real-time Systems (ECRTS'20)*. 1–22.
[12] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. 2011. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)*. Association for Computing Machinery, New York, NY, 353–362. https://doi.org/10.1145/2076732.2076783
[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] 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* 50, 1, Article 16 (April 2017), 33 pages. https://doi.org/10.1145/3054924
[15] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP'19)*. IEEE, 985–999.
[16] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security'15)*. USENIX Association, Washington, D.C., 161–176. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini.
[17] Nicholas Carlini and David Wagner. 2014. {ROP} is still dangerous: Breaking modern defenses. In *23rd {USENIX} Security Symposium ({USENIX} Security'14)*. 385–399.
[18] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*. 559–572.
[19] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. 2011. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*.
[20] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H. Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attack. In *2014 Network and Distributed Systems Security (NDSS) Symposium*, Vol. 26. 1–14.
[21] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. 2016. HCFI: Hardware-enforced control-flow integrity. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*. 38–49.
[22] Abraham A. Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. 2017. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP'17)*. IEEE, 289–303.
[23] EEMBC The Embedded Microprocessor Benchmark Consortium et al. 2015. CoreMark-Pro. https://www.eembc.org/coremark-pro.
[24] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, Vol. 98. San Antonio, TX, 63–78.
[25] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*. 763–780.
[26] Leo R. Dalesio, Jeffrey O. Hill, Martin Kraimer, Stephen Lewis, Douglas Murray, Stephan Hunt, William Watson, Matthias Clausen, and John Dalesio. 1994. The experimental physics and industrial control system architecture: Past, present, and future. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 352, 1–2 (1994), 179–184.

[27] Thurston H. Y. Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. 555–566.

[28] 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.

[29] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. 2010. *Return-oriented Programming without Returns on ARM*. Technical Report. Technical Report HGI-TR-2010-002, Ruhr-University Bochum.

[30] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security'14)*. USENIX Association, San Diego, CA, 401–416. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi.

[31] Ruan de Clercq and Ingrid Verbauwhede. 2017. A survey of hardware-based control flow integrity (CFI). *arXiv preprint arXiv:1706.07257* (2017).

[32] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J. Walls, and John Criswell. 2021. Holistic control-flow protection on real-time embedded systems with Kage.

[33] R. Earnshaw. 2005. ARM Procedure Call Standard for the ARM Architecture.

[34] M. A. El-Sarraf, A. El-Sayed Abdo, and Mahmoud A. Abdulwahab. 2013. Usability of epoxy/ilmenite composite material as an attenuator for radiation and a restoration mortar for cracks. *Annals of Nuclear Energy* 60 (2013), 362–367.

[35] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 901–913.

[36] Nicolas Falliere, Liam O. Murchu, and Eric Chien. 2011. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response* 5, 6 (2011), 29.

[37] Mahsa Foruhandeh, Yanmao Man, Ryan Gerdes, Ming Li, and Thidapat Chantem. 2019. SIMPLE: Single-frame based physical layer identification for intrusion detection and prevention on in-vehicle networks. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC'19)*. Association for Computing Machinery, New York, NY, 229–244. https://doi.org/10.1145/3359789.3359834

[38] Aurélien Francillon and Claude Castelluccia. 2008. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*. 15–26.

[39] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. Jitguard: Hardening just-in-time compilers with sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2405–2419.

[40] Jiri Gaisler et al. 2001. The LEON processor user's manual. *Gaisler Research* 2 (2001).

[41] Shay Gal-On and Markus Levy. 2012. Exploring coremark a benchmark maximizing simplicity and efficacy. In *The Embedded Microprocessor Benchmark Consortium*.

[42] Thanassis Giannetsos, Tassos Dimitriou, Ioannis Krontiris, and Neeli R. Prasad. 2010. Arbitrary code injection through self-propagating worms in Von Neumann architecture devices. *Computer Journal* 53, 10 (2010), 1576–1593.

[43] Zonghua Gu, Chao Wang, Ming Zhang, and Zhaohui Wu. 2014. WCET-aware partial control-flow checking for resource-constrained real-time embedded systems. *IEEE Transactions on Industrial Electronics* 61, 10 (2014), 5652–5661. https://doi.org/10.1109/TIE.2014.2301752

[44] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization (WWC-4'01) (Cat. No. 01EX538)*. IEEE, 3–14.

[45] Xiaochen Hao, Mingsong Lv, Jiesheng Zheng, Zhengkui Zhang, and Wang Yi. 2019. Integrating cyber-attack defense techniques into real-time cyber-physical systems. In *2019 IEEE 37th International Conference on Computer Design (ICCD'19)*. IEEE, 237–245.

[46] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B. Bobba. 2017. Contego: An adaptive framework for integrating security tasks in real-time systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS'17)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[47] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B. Bobba. 2018. A design-space exploration for allocating security tasks in multicore real-time systems. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE'18)*. IEEE, 225–230.

[48] 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.

[49] ARM Holdings. [n.d.]. ARMv7-M Architecture Reference Manual, December 2014. https://developer.arm.com/documentation/ddi0403/latest/.
[50] 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 *2016 IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 969–986.
[51] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. 2006. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*. 2–12.
[52] Zhijun Huang, Tao Zheng, Yunxiu Shi, and Ang Li. 2012. A dynamic detection method against ROP and JOP. In *2012 International Conference on Systems and Informatics (ICSAI'12)*. IEEE, 1072–1077.
[53] Intel. [n.d.]. A Technical Look at Intel's Control-flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html.
[54] Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. 2020. TZmCFI: RTOS-Aware Control-Flow Integrity Using TrustZone for Armv8-M. *International Journal of Parallel Programming* 49 (2020), 1–21.
[55] Junsung Kim, Karthik Lakshmanan, and Ragunathan Rajkumar. 2012. Rhythmic tasks: A new task model with continually varying periods for cyber-physical systems. In *2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*. IEEE, 55–64.
[56] Sreenath Krishnadas. 2016. Concept and Implementation of AUTOSAR compliant Automotive Ethernet stack on Infineon Aurix Tricore board. (2016). https://nbn-resolving.org/urn:nbn:de:bsz:ch1-qucosa-212626.
[57] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. 2019. uXOM: Efficient eXecute-only memory on {ARM} Cortex-M. In *28th {USENIX} Security Symposium ({USENIX} Security'19)*. 231–247.
[58] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. 2004. Enlisting hardware architecture to thwart malicious code injection. In *Security in Pervasive Computing*. Springer, 237–252.
[59] Jinfeng Li, Liwei Chen, Qizhen Xu, Linan Tian, Gang Shi, Kai Chen, and Dan Meng. 2020. Zipper stack: Shadow stacks without shadow. In *European Symposium on Research in Computer Security*. Springer, 338–358.
[60] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. 2020. PACStack: An Authenticated Call Stack. arXiv:1905.10242 [cs.CR]
[61] Arm Ltd. 2020. ARM Cortex-R. https://developer.arm.com/ip-products/processors/cortex-r.
[62] Microsoft Ltd. [n.d.]. Control Flow Guard - Win32 Apps. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.
[63] Minzhao Lyu, Dainel Sherratt, Arunan Sivanathan, Hassan Habibi Gharakheili, Adam Radford, and Vijay Sivaraman. 2017. Quantifying the reflective DDoS attack capability of household IoT devices. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'17)*. Association for Computing Machinery, New York, NY, 46–51. https://doi.org/10.1145/3098243.3098264
[64] Stephen McCamant and Greg Morrisett. 2005. Efficient, verifiable binary sandboxing for a CISC architecture. (2005). https://dspace.mit.edu/handle/1721.1/30542.
[65] Mitra Nasri, Thidapat Chantem, Gedare Bloom, and Ryan M. Gerdes. 2019. On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'19)*. IEEE, 103–116.
[66] Ahmad M. K. Nasser and Di Ma. 2020. SecMonQ: An HSM based security monitoring approach for protecting AUTOSAR safety-critical systems. *Vehicular Communications* 21 (2020), 100201.
[67] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N. Asokan. 2017. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 259–284.
[68] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. 2002. Control-flow checking by software signatures. *IEEE Transactions on Reliability* 51, 1 (2002), 111–122.
[69] James Pallister, Simon Hollis, and Jeremy Bennett. 2013. BEEBS: Open benchmarks for energy measurements on embedded platforms. *arXiv* (2013), arXiv–1308.
[70] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.
[71] Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. 2017. Compiler-assisted loop hardening against fault attacks. *ACM Transactions on Architecture and Code Optimization* 14, 4, Article 36 (Dec. 2017), 25 pages. https://doi.org/10.1145/3141234
[72] Donald Ray and Jay Ligatti. 2012. Defining code-injection attacks. *ACM Sigplan Notices* 47, 1 (2012), 179–190.

[73] Abhishek Rhisheekesan, Reiley Jeyapaul, and Aviral Shrivastava. 2019. Control flow checking or not? (For soft errors). *ACM Transactions on Embedded Computing Systems* 18, 1 (2 2019). https://doi.org/10.1145/3301311

[74] Gerardo Richarte et al. 2002. Four different tricks to bypass stackshield and stackguard protection. (2002). https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.3034&rep=rep1&type=pd.

[75] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)* 15, 1 (2012), 1–34.

[76] Monireh Safari and Reihaneh Khorsand. 2018. PL-DVFS: Combining power-aware list-based scheduling algorithm with DVFS technique for real-time tasks in cloud computing. *Journal of Supercomputing* 74, 10 (2018), 5578–5600.

[77] Sonal Saha and Binoy Ravindran. 2012. An experimental evaluation of real-time DVFS scheduling algorithms. In *Proceedings of the 5th Annual International Systems and Storage Conference*. 1–12.

[78] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. 2013. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems* 49, 4 (2013), 404–435.

[79] Dr Sarwar Sayeed, Hector Marco-Gisbert, Ismael Ripoll, and Miriam Birch. 2019. Control-Flow Integrity: Attacks and Protections. *Applied Sciences* 9, 20 (2019), 4229.

[80] Simon Schuster, Peter Ulbrich, Isabella Stilkerich, Christian Dietrich, and Wolfgang Schröder-Preikschat. 2017. Demystifying soft-error mitigation by control-flow checking - A new perspective on its effectiveness. *ACM Transactions on Embedded Computing Systems* 16, 5s (2017), 1–19. https://doi.org/10.1145/3126503

[81] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. 552–561.

[82] Z. Shao, Q. Zhuge, Y. He, and E. H. Sha. 2003. Defending embedded systems against buffer overflow via hardware/software. In *19th Annual Computer Security Applications Conference, 2003. Proceedings*. 352–361. https://doi.org/10.1109/CSAC.2003.1254340

[83] Kang G. Shin and Yi-Chieh Chang. 1995. A reservation-based algorithm for scheduling both periodic and aperiodic real-time tasks. *IEEE Transactions on Computing* 44, 12 (1995), 1405–1419.

[84] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. 2001. *Vulcan: Binary Transformation in a Distributed Environment*. Technical Report MSR-TR-2001-50. 12 pages. https://www.microsoft.com/en-us/research/publication/vulcan-binary-transformation-in-a-distributed-environment/.

[85] Bowen Tang, Huan Ying, Wei Wang, and Huabin Tang. 2017. Eternal War in Software Security: A Survey of Control Flow Protection. In *2016 7th International Conference on Education, Management, Computer and Medicine (EMCM'16)*. Atlantis Press.

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

[87] Xavier Vera. 2020. Inside tiger lake: Intel's next generation mobile client CPU. In *Hot Chips Symposium*. 1–26.

[88] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. 2015. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing* 13, 4 (2015), 437–450.

[89] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. 203–216.

[90] Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi, and Bryan C. Ward. 2019. Control-flow integrity for real-time embedded systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS'19)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[91] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.

[92] Nathanael R. Weidler, Dane Brown, Samuel A. Mitchell, Joel Anderson, Jonathan R. Williams, Austin Costley, Chase Kunz, Christopher Wilkinson, Remy Wehbe, and Ryan Gerdes. 2019. Return-oriented programming on a resource constrained device. *Sustainable Computing: Informatics and Systems* 22 (2019), 244–256.

[93] Penglin Yang, Limin Tao, and Haitao Wang. 2018. RTTV: A dynamic CFI measurement tool based on TPM. *IET Information Security* 12, 5 (2018), 438–444.

[94] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. 2016. Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'16)*. IEEE, 1–12.

[95] Ming Zhang, Zonghua Gu, Hong Li, and Nenggan Zheng. 2018. WCET-aware control flow checking with super-nodes for resource-constrained embedded systems. *IEEE Access* 6 (2018), 42394–42406. https://doi.org/10.1109/ACCESS.2018.2852805

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

[97] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. 2020. Silhouette: Efficient protected shadow stacks for embedded systems. In *29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 1219–1236. https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-jie.

[98] Nikola Zlatanov. 2016. ARM Architecture and RISC Applications. (2016). https://www.researchgate.net/profile/Nikola-Zlatanov/publication/295921119_ARM_Architecture_and_RISC_Applications/links/56d0ece908ae059e375d4df5/ARM-Architecture-and-RISC-Applications.pdf.

Received 17 March 2021; revised 28 January 2022; accepted 5 April 2022