A Procrastinating Control-Flow Integrity Framework for Periodic Real-Time Systems

Tanmaya Mishra
tanmayam@vt.edu
Virginia Tech
Arlington, Virginia, USA

Jinwen Wang
jinwen.wang@wustl.edu
Washington University in St Louis
St Louis, Missouri, USA

Thidapat Chantem
tchantem@vt.edu
Virginia Tech
Arlington, Virginia, USA

Ryan Gerdes
rgerdes@vt.edu
Virginia Tech
Arlington, Virginia, USA

Ning Zhang
zhang.ning@wustl.edu
Washington University in St Louis
St Louis, Missouri, USA

ABSTRACT

Connected embedded systems and cyber-physical systems exhibit larger attack surface than isolated ones. Control-flow integrity (CFI) is a set of techniques to prevent attackers from redirecting program control-flow and performing arbitrary computation, by detecting and checking control-flow transfers. Currently CFI for real-time systems either operate in-line with code execution, often depending on hardware mechanisms for improved performance and/or security guarantees, or focus solely on budget management when performing CFI out-of-order. In this work, we exploit the predictable release pattern of periodic real-time systems to create a novel CFI framework. This framework (1) consists of a novel real-time task model, which explicitly considers CFI related execution along with the regular portion of the tasks, and (2) presents a novel hardware-assisted trusted scheduler to enable a unique combination of out-of-order and in-line control flow enforcement on forward edge and backwards edge, respectively, to minimize performance overhead while ensuring real-time deadlines. Our framework provides the flexibility to model arbitrary forward-edge CFI as security tasks, so that we may strategically schedule them, and provide schedulability and correctness analysis to explicitly ensure that CFI verification is always performed on time without affecting the timeliness of the real-time tasks. Simulations show that our new task model outperforms existing work in terms of resource usage, thus allowing for more complex and sophisticated CFI to be implemented. We implement our approach on real hardware and microbenchmarks confirm that our approach has comparable in-line overhead as existing work.

CCS CONCEPTS

• **Computer systems organization → Real-time operating systems**; *Embedded software*; • **Security and privacy → Trusted computing**.

This work is licensed under a Creative Commons Attribution International 4.0 License.

RTNS 2023, June 07–08, 2023, Dortmund, Germany
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9983-8/23/06.
https://doi.org/10.1145/3575757.3575762

KEYWORDS

control-flow integrity, real-time systems

ACM Reference Format:
Tanmaya Mishra, Jinwen Wang, Thidapat Chantem, Ryan Gerdes, and Ning Zhang. 2023. A Procrastinating Control-Flow Integrity Framework for Periodic Real-Time Systems. In *The 31st International Conference on Real-Time Networks and Systems (RTNS 2023), June 07–08, 2023, Dortmund, Germany.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3575757.3575762

1 INTRODUCTION

Real-time systems are now more connected than ever before. Data sharing and inter-operation have allowed these systems to better understand their environment and make advanced decisions, improving our quality of life. However, interconnecting such traditionally isolated systems can provide entry-points for attackers to exploit, infiltrate, and modify system operation. For example, the modern car is composed of hundreds of computers, also called electronic control units (ECU), that constantly communicate to maintain smooth and safe vehicle operation. Traditionally, these systems do not communicate with external systems. However, modern conveniences such as Bluetooth connectivity for vehicle occupants, or vehicle-to-everything (V2X), have increasingly exposed these systems to external influence. Prior work [18] shows that such exposure increases the possibility of exploitation by attackers.

At the same time, many such real-time systems are resource-constrained in processing power and/or memory. For example, vehicle ECUs used for advanced driver assistance systems (ADAS) utilize microcontrollers that operate at hundreds of MHz of clock frequency [27] whereas a modern server or desktop computer's CPU clock runs in the order of a few GHz. Further, ADAS mechanisms such as automatic cruise control, traction control, pedestrian-detection etc., have real-time requirements [15] to ensure occupant safety. Resource limitations and real-time deadline requirements can also be found in other applications such as industrial control systems, robotics, etc. Therefore, both of these attributes must be considered when designing for secure systems.

Once an attacker infiltrates a system, they can perform a variety of system level runtime attacks. We focus on a set of state-of-the-art attacks called control-flow hijacking attacks. Here an attacker either injects code, or leverages memory corruption vulnerabilities to overwrite control data, such as return addresses, to manipulate the execution of the program. Return-oriented programming (ROP) [37]

is a popular attack that systematically abuses return addresses to *reuse* existing program code. ROP, and its variants are, therefore, also called *code-reuse* attacks. To defend against such attacks, a set of defenses called control-flow integrity (CFI) have been proposed. CFI detect, and prevent, any deviation from the intended control-flow paths of the program. Note that in this work we refer to CFI as "CFI technique" or "CFI mechanism", interchangeably.

Due to the limited resources in embedded systems, modern CFI techniques work around hardware limitations [43], utilize architecture extensions [34], require custom hardware [29], or even reduce detection precision (coarse-grained CFI) and lazily perform checks at checkpoints [12]. Even fewer CFI techniques exist that explicitly consider the timing guarantees of real-time systems. Prior work, such as RECFISH [39] provides a large scale schedulability study of traditional CFI concepts in embedded systems. Here CFI is introduced *in-line* of code-execution, inflating the worst-case execution time (WCET) of tasks. On the other hand, Kadar et al. [24] introduce CFI budget management techniques to ensure timing guarantees. Here, hardware tracing mechanisms gather control-flow information during program execution, and then verify them at a later time (*out-of-order* to program execution) when the trace buffers are full. Execution budgets are enforced by temporarily suspending CFI checks to prevent the possibility of deadline misses.

Since these mechanisms trade off CFI enforcement accuracy for better real-time performance, they inevitably introduce "blind-spots" that could be exploited. Specifically, by not performing CFI in-line, a smart attacker could modify system output just-in-time before it is sent out. An out-of-order defense mechanism without explicit timing bounds on the completion of verification/enforcement may not be able to detect such an attack in time. In this work, our key observation is that in periodic real-time systems with predictable job release patterns, we can a) perform some CFI checks out-of-order of system code execution, b) correctly determine deadlines for these checks so that the aforementioned blind-spots are mitigated, and c) relax these deadlines, when possible, to improve resource utilization. To determine and relax these deadlines, we were motivated by the fact that many real-time industrial-control and robotics systems gather sensor data (system input) at high frequencies, but send actuator control commands (system output) at lower frequencies. For example, anti-lock braking systems (ABS) sample wheel speed sensors every few milliseconds, but send brake commands at an order of magnitude lower frequency [20]. Combined with the increasing adoption of timing models such as Logical Execution Time [25] model to mitigate data staleness when sharing data between tasks, it is possible to determine when system output is generated, and if there is an opportunity to relax deadlines of CFI related operations. Therefore, in this work we make the following contributions:

(1) Decompose CFI operations into in-line and out- of-order components while still maintaining the effectiveness of CFI operations, improving their efficiency, and allowing more complex CFI such as context-sensitive CFI to be implemented
(2) Propose a new, integrated real-time/security task model, which explicitly considers CFI related execution along with the regular portion of the tasks by modeling forward-edge CFI (such as verifying function calls) as separate security tasks with their own deadlines. This allows for CFI checks to be made in time to detect and neutralize CFI related attacks without affecting the temporal correctness of the real-time applications.
(3) Provide a strategy to relax deadlines of security tasks to reduce resource utilization. We provide a schedulability and correctness analyses to ensure such deadline relaxations do not undermine the fact that the control flow integrity policy is strictly enforced before system output (actuation) is generated.
(4) Implement in real hardware a novel trusted scheduler based on ARMv8-M TrustZone to support our integrated CFI framework.
   (a) Unlike prior work [21, 24, 39], our framework is the first to guarantee scheduler integrity by protecting the scheduling infrastructure that is crucial in real-time scheduling. The trusted infrastructure consists of scheduler and job queues.
   (b) Record forward-edge control-flow information, while simultaneously performing, in-line, backward-edge CFI to further improve performance. This is done via a novel trapping mechanism that also ensures the integrity of CFI records. Our system also supports context-sensitive CFI on commodity hardware.

Note that since CFI techniques continue to be refined by security researchers, the focus of our paper is on the novel, integrated real-time/security CFI framework to ensure timely completion of out-of-order CFI policy enforcement and lightweight in-line CFI checks, rather than new CFI schemes. As a result, the discussion centers around the newly proposed framework to allow implementation of existing CFI techniques to be performed correctly out-of-order to program execution on commodity ARM Cortex-M hardware. To the best of our knowledge, we are the first to address the real-time perspectives of out-of-order CFI enforcement while assuming a strong adversary capable of compromising the normal world OS.

2 RELATED WORK

CFI is a state-of-the-art topic in system security with over two decades [16, 26, 32] of work, due to the increasing popularity and sophistication of control-flow hijacking attacks such as return-oriented programming (ROP [37]) and its variants [13, 37]. CFI mechanisms are built to protect either the *backward-edge*, such as the return target of a function, or the *forward-edge* such as the target of an indirect function call, or a combination of backward and forward-edges. While backward-edge CFI is simpler to implement since it requires verifying past control-flow transfers, forward-edge CFI is more difficult since it requires more information to ascertain possible future control-flow paths. Forward-edge CFI, therefore, requires a record of program control-flow information such as a control-flow graph (CFG), which is obtained a-priori to deployment, and is heavily dependent on the quality of the available CFG. A wide variety of forward-edge CFI is available, such as fine-grained CFI [16] that provides a correlation between a jump source and all possible jump targets, and coarse-grained CFI [42] that utilizes a less defined CFG and, therefore, allows reduction in the number of checks performed during control-flow transfers.

CFI has been increasingly adopted by many higher-end commercial hardware and software systems such as in Microsoft's Windows operating system as part of its Control-Flow Guard defenses [30], the Clang compiler [2] as well as Intel's Control-Flow Enforcement [23] technology in their recent line of processors. However,

CFI built specifically for resource-constrained embedded systems (a comprehensive study is available here [32]) are far fewer in number. In the case of real-time embedded systems, even fewer CFI defenses exist in prior work such as Walls et al.'s RECFISH [39] which provides an extensive schedulability study of the implementation of common CFI techniques, particularly shadow stacks [17], and labeling to enforce forward-edge CFI [16] on realistic real-time embedded hardware. Shadow stacks protect backward-edges such as function return locations, while labeling is used to correlate a jump source location with possible jump target locations. However, RECFISH assumes a weaker threat model as it requires a privilege isolation between tasks and the real-time operating system (RTOS). It also does not utilize real-time properties of the system to improve the efficiency of the CFI mechanism. Bellec et al. [10], on the other hand, propose a detection mechanism that exploits the temporal predictability of real-time systems to detect greater-than-normal execution time but requires specialized hardware. More recently, the work by Serra et al. [36] provides a CFI framework that depends on pointer authentication features recently introduced in the ARMv8.3-A instruction set architecture (ISA), while providing the ability to perform timing analysis. However, under the strong threat model we assumed where the adversary can compromised the kernel, instrumentation that the existing approaches rely on for security policy enforcement are no longer trustworthy, limiting the security protection.

Two prior works, Kadar et al. [24] and Hasan et al. [21] are similar to our work. Specifically, they abstract CFI, or security mechanisms in general, as security tasks. However, no currently available defense mechanism for real-time systems, including these two works, considers the temporal relation between *when an attack occurs* (attacker modifies program flow) and *when the attack takes effect (modified program flow affects system output). For example, by not defining such a relation, an attacker could have already affected system output before the verification mechanism detects an attack.*

Note that [24] also utilizes debug architectures such as ARM CoreSight [40] to perform program trace capture and requires additional computing resources (extra processor core). However, such technologies are components of optional debug architectures which may not be viably deployed at scale due to their additional silicon cost. Finally, our framework's ability to defer the execution of CFI mechanisms to later in a controlled manner to improve resource usage is novel and orthogonal to any particular hardware implementation technique for CFI.

3 PRELIMINARIES

3.1 Hardware model

We require that the target system consists of a single ARM TrustZone equipped processor. ARM TrustZone for Cortex-M [41] is a widely available component of the Platform Security Architecture (PSA) extensions on the ARMv8-M state-of-the-art microcontroller architecture. TrustZone creates two processor domains, *non-secure* and *secure*. The former is used to run legacy software, such as an RTOS and task code, whereas trusted supervisory code executes in the latter. ARMv8-M has a flat address space consisting of RAM, flash, and system peripherals. The division of address space between the two domains is enforced via a vendor hard-wired implementation-defined attribution unit (IDAU) and a system attribution unit (SAU). The SAU supports multiple *regions* depending on the implementation. Each region is a set of registers where secure domain code can programmatically upgrade a non-secure domain address space to secure domain. Switching between the domains has low runtime overhead [31]. Non-secure domain code cannot read/write contents of secure domain memory and can only call specially marked *non-secure callable* (NSC) secure domain code. Other cross-domain memory accesses from the non-secure domain cause a `HardFault`. We utilize the SAU and the `HardFault` in our mechanism (Section 8). It should be noted that the hard-fault on ARM processors is traditionally utilized to handle critical system exceptions. As part of the ARMv8-M *mainline* architecture (implemented in the Cortex-M33 architecture), an illegal cross-domain memory access (TrustZone violation) launches a dedicated `SecureFault` exception handler. However, we aim to target even the ARMv8-M *baseline* architecture (implemented in the lower-powered Cortex-M23 variant) that bundles a TrustZone violation into a `HardFault`, and calls the `HardFault` exception handler, to reduce manufacturing costs. We discuss differentiating between a legitimate system fault/exception and an attack in Section 9. In this work, we refer to either of the fault exceptions as a hard-fault exception for simplicity.

3.2 Software Model

The software consists of an RTOS executing periodic real-time tasks with known worst-case execution time (WCET). While the RTOS and application task code normally would execute within the same domain, we split them apart. Specificaly, we execute the RTOS scheduler in the secure domain and task code in the non-secure domain to ensure the integrity of the scheduler and its output (Section 8.2). Hardware drivers for peripherals are also kept in the non-secure domain since they are traditionally integrated into task code in embedded systems. The application program's tasks are referred to as *application tasks*. Application tasks are further divided into *internal* and *output* tasks (Section 5.1). The sensor task and actuator task in Figure 1 are examples of internal and output tasks, respectively. Separately, we introduce the concept of *security tasks* (Section 5.2). We utilize the rate difference between different application tasks to opportunistically relax deadlines of the security tasks (Section 6).

We consider that tasks release data at their deadlines, similar to the popular Logical Execution Time model [25] which is widely being considered for automotive and industrial systems since it implicitly addresses the problem of data staleness. While our core idea does not require this assumption, it simplifies our approach and reduces the complexity of our deadline relaxation model (Section 6). It is assumed that data synchronization between tasks happens through shared buffers or similar mechanisms.

3.3 Threat model

We assume the use of a system that supports TrustZone. An attacker can compromise any tasks and attain the highest privilege within the non-secure domain. This is representative of real-world threats. For example, task and RTOS code execute at the same privilege level in FreeRTOS [8] by default to eliminate the runtime overhead of privilege level switching. Prior related work, such as RECFISH [39]

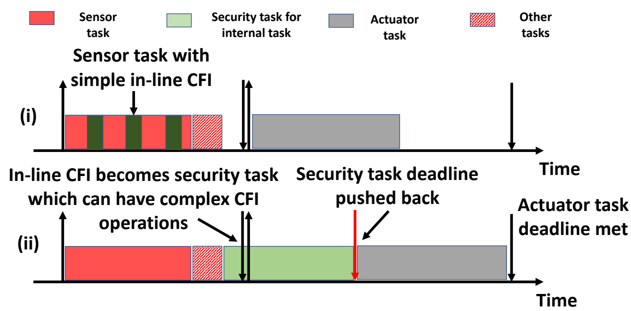

Figure 1: (i) Task system where sensor task has simple in-line CFI, (ii) Proposed procrastinating model where in-line CFI is bundled into a security task (Section 5.2) which can implement complex CFI operations (accomodate greater WCET) but with relaxed deadlines to lower resource utilization.

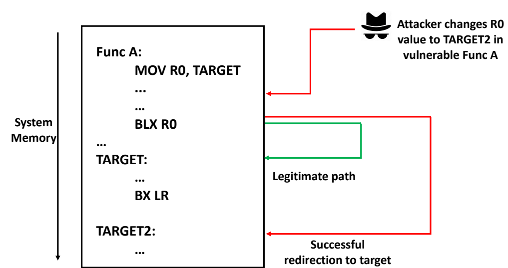

Figure 2: Code redirection on a generic microcontroller. Attacker modifies contents of `R0` to change branch target.

requires privilege isolation between tasks and the RTOS since they utilize memory protection units (MPU). We assume a *stronger* threat model that considers a compromised non-secure domain.

Similar to other CFI research, we focus on defending against code-reuse attacks, wherein an attacker will target indirect branches (branches on a general purpose register or link register) to implement a forward-edge or backward-edge attack, respectively. Data-only attacks are out of the scope of this work since they cannot be detected by CFI [22]. Figure 2 shows a sample attack: An attacker forces code execution to redirect to an unintended target address by exploiting a vulnerability in `Func A`. By chaining together multiple such branches to different blocks of instructions, an attacker can perform arbitrary computations.

4 OVERVIEW OF PROCRASTINATING CFI

We leverage the periodic nature of real-time tasks to reduce the overhead associated with CFI operations, which, in turn, allow us to support more complex/sophisticated CFI than in previous work. A motivating example can be seen in Figure 1.(i) which shows a typical control-system where a *sensor task* gathers sensor data, and reports the data after processing to an *actuator task* that sends out commands to control an actuator. The sensor task has in-line CFI, like most prior work. On the other hand, we propose an approach shown in Figure 1.(ii) where the in-line CFI can be consolidated into a separate security task. By leveraging the fact that the data produced by an input task is not used until the corresponding output task starts executing, the deadline for the security task can be relaxed, potentially allowing for more complex and/or time consuming CFI such as context-sensitive CFI [38] to be implemented (denoted by the lighter green color in the figure) without affecting the original deadline of the input nor output tasks. Our approach consists of the following components:

Procrastinating CFI Task Model (Section 5) - Introduces security tasks, differentiates them from application tasks, and provides some important properties of the tasks themselves.

Security Task Deadline Relaxation (Section 6) - Derives relations to relax security task deadlines such as in applications like ABS [20] where actuation takes place at a lower frequency than sensing.

Ensuring Correctness and Schedulability (Section 7) - Addresses an implicit relationship between output and security tasks and shows that our model is readily compatible with any resource sharing-aware scheduling algorithm.

Mechanism (Section 8) - Presents an overview of our on-device mechanism (Figure 3) which logs forward-edge transfers and performs backward-edge CFI. Details on our trusted scheduling infrastructure can be found in this section as well.

While our task model itself advances the state-of-the-art by introducing the notion of explicit deadlines to complete CFI to ensure attacker detection before system output is generated, our mechanism further improves the state-of-the-art by extending our model to allow for context-sensitive CFI on commodity hardware.

5 PROCASTINATING CFI TASK MODEL

Prior work, especially lazy schemes [7, 12] which check for an attack at specific points of time during program execution, cannot always detect an attacker before it affects system output. For example, an attacker could send commands to an actuator controlled by the system before it is detected by the CFI mechanism. We exploit the predictable pattern of periodic real-time systems and design a novel task model that will play a key role in our CFI mechanism. The design of this task model is based on the strict requirement that all pending CFI checks must be completed before the system output is generated i.e., in the example stated above, the CFI check must complete before any command is sent to the actuator. The model also assumes that there exists a mechanism (Section 8) to capture information of control-flow transfer events such as the source and target addresses that can be retrieved at a later time for verification.

Notations: We consider that the system consists of n periodic real-time *application tasks*, which are synchronously released at time 0 (Section 5.1) with known WCET and implicit deadlines. The tasks are scheduled using EDF+SRP (although the tasks are independent, some explicit dependencies must be considered for correctness (Section 7)). For each application task, we introduce a new periodic real-time *security task* (Section 5.2) that models forward-edge CFI that will be performed on the task's control-flow logs. In Section 6, we show that these security tasks may not have implicit deadlines. An instance of any of these tasks is called a job. τ_i is the ith application task and $\tau_{s,i}$ is the corresponding security task. D_i, P_i, C_i are the relative deadline, period and WCET of τ_i, respectively. Similarly $C_{s,i}, D_{s,i}, P_{s,i}$ are the WCET, period and relative deadline of $\tau_{s,i}$, respectively. The jobs $j_{i,k}$ and $j_{s,i,k}$

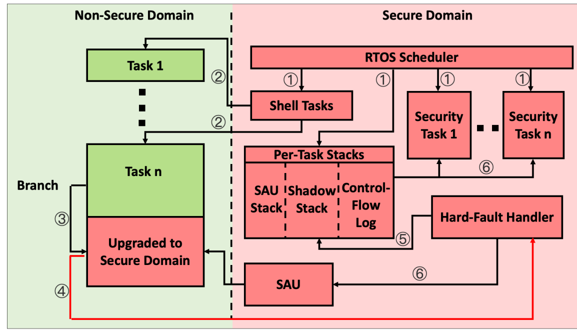

Figure 3: Overview of procrastinating CFI mechanism to capture forward-edge control-flow logs (reading source and destination addresses of branch) and backward-edge verification. Branching into upgraded memory launches fault handler that performs both operations - Section 8.

are the kth instances of τ_i and $\tau_{s,i}$, respectively, and their absolute deadlines are $d_{i,k}$ and $d_{s,i,k}$, respectively. The task utilizations are represented by U_i for τ_i and $U_{s,i}$ for $\tau_{s,i}$. Please see Section 5.2 for more details.

5.1 Application task model

Our application model is based on a real-time control system, such as an industrial control or robotics system. Every task releases its computed data, and consumes input data, only at its deadline and arrival times, respectively, similar to the Logical Execution Time (LET) model [25]. Each application task is either an a) *internal task* that collects data from sensors and/or performs computations, or an b) *output task* that controls actuators or sends messages to external systems. Therefore, the sensor and actuator tasks in Figure 1 are examples of an internal and output task, respectively. More importantly, motivated by applications such as ABS discussed earlier, *we assume that the input tasks have higher frequencies than output tasks.* Our model also assumes that the output tasks are given exclusive write access to peripherals (we provide a sample low-overhead mechanism in Section 8.2). Since data is released only at deadlines (and in our case, periods), data availability is predictable, regardless of the scheduling algorithm [25]. This removes the need to explicitly consider task dependency. Internal tasks provide data to output tasks either directly or through other internal tasks. As stated in Section 3.2, it is assumed that tasks share data via shared buffers or similar mechanisms. We assume that multiple internal jobs can queue data in the shared buffer to be consumed by the output job when it arrives, such as being copied to memory that is accessible only to the output job. Once the data is consumed by the output job[1], the buffer is purged for newer data.

Data dependency from an adversarial perspective. While logical relations between internal tasks and output tasks can be deduced by generating a directed acyclic graph (DAG) of the flow of data from internal to output tasks, the same relation cannot be assumed to hold true when seen from an adversarial context. For example, consider a system that contains a set of 5 tasks $\tau_a, \tau_b \ldots \tau_e$ with $\tau_a, \ldots \tau_c$ as internal and the remaining as output, such that the data relation between the tasks can be two chains in the DAG graph with $\tau_a \rightarrow \tau_b \rightarrow \tau_d$ and $\tau_c \rightarrow \tau_e$. We cannot assume that an attacker controlling internal task τ_a will only affect the input of τ_b which then filters to τ_d. That is, it could try to directly access the shared buffers of both τ_d *and* τ_e unless the system explicitly prevents such modifications. While TrustZone (which we use in our mechanism in Section 8.2) could be used for isolating these buffers, due to the limited number of discrete memory regions (up to 8 for our test hardware) that could be managed/upgraded by the SAU, we cannot overtly depend on TrustZone to manage every memory region for complex applications. Scheduling techniques must be utilized to reduce the pressure on a TrustZone-based memory upgrade mechanism. In summary, we cannot assume any relation between the internal and output tasks from a security perspective other than that system output is generated by output tasks at their deadline, and output tasks consume all their input data when they arrive.

[1]Here an attacker could abuse the temporal assumptions of the framework which we consider in Section 7.1.

5.2 Security task model

Under the assumption of a strong attacker, all application tasks are considered equally susceptible to an attack. Possible attack vectors include malicious external input supplied to the task(s) either directly (e.g., sensor input) or through shared data from another task that may trigger a bug and allow an attacker to take control of the task. Therefore, it is necessary to check the control path of all tasks before a system output is generated.

A security task encapsulates all the forward-edge CFI required for indirect branches (see our threat model in Section 3.3). that need to be carried out for a corresponding application task (internal or output). Note that CFI only requires control-flow logs (such as those generated by our mechanism in Section 8) and does not access the data shared between application tasks. Therefore, security tasks require that the corresponding application tasks execute and generate these logs before they can perform any verification. We discuss the implementation of the security tasks in Section 8.4. Note that CFI for backward edge control-flow transfer is not considered since they are relatively inexpensive to verify in-line. Valid return locations are simply the instruction after a forward-edge transfer (such as a function call/branch). Therefore, unlike forward-edge CFI, backward-edge CFI do not require an expensive search of a valid path control-flow graph. Further, capturing backward-edge transfer endpoints would be very expensive due to their relatively larger number than that of forward-edges for typical programs (see comparison in Table 1).

The WCET of a security task $C_{s,i}$, under a worst case number of control-flow transfer events during regular execution $\mathbb{N}$, is:

$$C_{s,i} = \mathit{cfi}(\mathbb{N}), \quad (1)$$

where $\mathit{cfi}()$ provides the WCET of the CFI implementation given an input set of control transfer events. Further, since a security task tracks the execution of the corresponding application task, it will also inherit its period, i.e., $P_{s,i} = P_i$.

We will now state some important properties of security jobs based on the observations of the behavior of the application jobs. Here we are considering that the system is originally schedulable,

i.e., for a set of n tasks, the total utilization of the tasks satisfies $\sum_{i=1}^{n} U_i \leq U_A^{sched}$, where U_A^{sched} is the schedulable utilization of a scheduling algorithm A.

Property 1. *The deadline of a security job $\tau_{s,i,k}$, released at the same time as its corresponding application job $\tau_{i,k}$ is:*

$$d_{s,i,k} \geq d_{i,k}. \tag{2}$$

For an output job, the corresponding security job deadline must equal output job deadline for an attack to be detected in a timely manner since the output is generated at this time. On the other hand, the deadline for an internal job's security job may be greater due to the rate difference between an input and output tasks discussed earlier (Section 5.1). Note that since security tasks require control-flow logs before they can perform verification, it is expected that the scheduler tie-breaks to the advantage of the application task if it shares the same priority with the corresponding security task. Then, if an internal job is controlled by an attacker, the latest time to detect the attack is at the deadline of the ouput task, i.e.,

Property 2. *An attacker controlled k^{th} job of an internal task τ_i targeting an upcoming output job $\tau_{j,l}$, will be detected by the security job $\tau_{s,i,k}$ before the system generates its output if:*

$$d_{s,i,k} \leq d_{j,l}. \tag{3}$$

To recap, the deadline of a security task corresponding to an output task must be no greater than the deadline of the output task while the deadline for those corresponding to internal tasks can potentially be relaxed. We discuss deadline relaxation in the next section. Also, we address the implicit data dependency between security and output jobs that can be exploited by the attacker (Section 5.1) in the subsequent section. In a nutshell, the data dependency problem can be solved by modeling output and security tasks as having shared resources and scheduling algorithms such as EDF+SRP can be used.

6 SECURITY TASK DEADLINE RELAXATION

It is potentially worthwhile to relax deadlines of the newly introduced security jobs to improve resource usage. This is possible for security tasks of internal tasks in the case where *output tasks execute at lower frequencies than internal tasks.* This is a reasonable assumption in control systems since output tasks must control actuators, which have a physical limit on how often they can respond to commands and/or due to system requirements. For example, ABS in vehicles have a much higher wheel speed sensor input rate than brake application rate [28].

Consider two tasks τ_1 and τ_2, where τ_1 is an internal task, such as a sensor task, and τ_2 is an output task, such as an actuator controller. Let's say that jobs $\tau_{1,m}$ and $\tau_{2,l}$ synchronize at time t which is when the lth job of τ_2 arrives and $t \geq d_{1,m}$. While the synchronization of data needs to happen at time t for correct system operation, the same need not be true for security tasks. For example, if the following is true:

$$t < d_{2,l} - (C_2 + C_{s,2}), \tag{4}$$

then there is a positive amount of time between the synchronization point and the last time unit at which the internal job's security job must complete to allow enough processor bandwidth to the output job and its security job. Since $d_{2,l} - (C_2 + C_{s,2})$ is evidently before

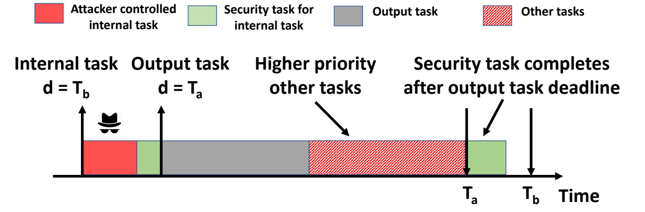

Figure 4: Race condition: Attacker is able to execute and affect input of output task, and the output task's deadline occurs before the security task is able to complete execution.

$d_{2,l}$, the deadline of the security job of the internal job can be set to $d_{2,l} - (C_2 + C_{s,2})$ while ensuring our initial guarantee that CFI checks always complete in-time before the system generates output. Figure 1.(ii) provides an overview of what we aim to achieve. Therefore, based *solely* on Property 2:

Property 3. *Consider output tasks $\tau_j \in \mathbf{O}$, where $\mathbf{O}$ is the set of all output tasks in the system. For the kth job of a security task corresponding to an internal job $\tau_{i,k}$ that synchronizes with the upcoming l^{th} job of τ_j (i.e., $d_{i,k} \leq r_{j,l}$ where $a_{j,l}$ is the arrival time of $\tau_{j,l}$), $d_{s,i,k}$, that satisfies Property 2, can be relaxed such that*

$$d_{s,i,k} = min\{d_{j,l} - (C_j + C_{s,j})\}, \tau_j \in \mathbf{O}. \tag{5}$$

Proof. To prove by contradiction, consider $d'_{s,i,k} > d_{s,i,k}$. Also, let $\tau_{s,i,k}$ executes up to $d'_{s,i,k}$. Further, let a pair of output and corresponding security tasks $\tau_{j,l}$ and $\tau_{s,j,l}$ begin execution at $d'_{s,i,k}$ and let j be the `argmin` satisfying Eq. 5. Then $d'_{s,i,k} + C_j + C_{s,j} > d_{j,l}$ which results in an attack not being detected before system output is generated. □

To generalize Eq. 5, we define a *push back value* for each security task. Recall from Section 5.1 that an attacker controlled internal task may synchronize with any output task. Therefore, let's consider a pair of internal and output tasks, τ_i and τ_j, respectively. For the latest job of τ_i that has a deadline earlier than/at the arrival of lth job of τ_j (and therefore can synchronize with τ_j), the push back value $\Psi_{i,j}^l$ for the job of the security task $\tau_{s,i}$ corresponding to the internal job is:

$$\Psi_{i,j}^l = (l \cdot P_j - (C_j + C_{s,j})) \bmod (P_i). \tag{6}$$

However this push back value may not be universally applied to any job of $\tau_{s,i}$. That is, for some instance of $\tau_{s,i}$, applying this push back may violate Property 3. Instead, a minimum $\Psi_{i,j}$, that can be applied for any job of $\tau_{s,i}$, can be derived as:

$$\Psi_{i,j} = min\{\Psi_{i,j}^l\},\ l = 1 \dots l^{max}. \tag{7}$$

Knowing that τ_i and τ_j are periodic, the push back values will repeat after every hyperperiod of tasks τ_i and τ_j. Therefore, l^{max} can be stated as:

$$l^{max} = \frac{\text{lcm}(P_i, P_j)}{D_j}, \tag{8}$$

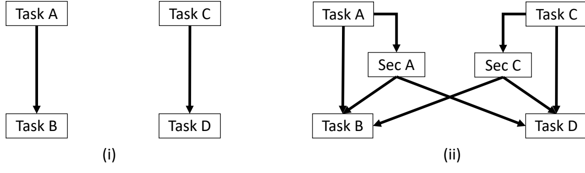

Figure 5: (i) Sample data dependency between internal tasks A and C with output tasks B and D. (ii) Dependencies when security tasks are considered.

where $\text{lcm}(\cdot,\cdot)$ returns the least common multiple of its inputs. We now generalize this for $\tau_{s,i}$ with respect to all $\tau_j \in \mathbf{O}$.

Property 4. *For task sets where output tasks have lower frequency than internal tasks, the security task $\tau_{s,i}$'s deadline can be pushed back by Ψ_i where*

$$\Psi_i = \min\{\Psi_{i,j}\},\ \forall\ \tau_j\ \in\ \mathbf{O}. \tag{9}$$

Finally, the relative deadline of the security task $\tau_{s,i}$ can now be calculated as:

$$D_{s,i} = D_i + \Psi_i. \tag{10}$$

7 ENSURING CORRECTNESS AND SCHEDULABILITY

The deadline relaxation approach described above is only correct if the attacker-controlled task behaves correctly temporally (see Footnote 1). We now discuss a *race condition* that an attacker-controlled internal task may exploit to break the correctness of the deadline relaxation method.

7.1 Race condition between output and security tasks

Consider a situation where a security job is released earlier (alongside an attacker-controlled internal job) but with a deadline *greater* than an upcoming output job (Figure 4). Under our model, this internal job does not synchronize with the output job (Section 5.1) and hence Property 2 needs not hold here. However, an attacker could ignore this behavior and overwrite the data to be synchronized with the output job. Since the security job has a later deadline than the output job, it may be prevented from completing verification until after the deadline of the output job. This could break the correctness of our approach in that we will then not be able to detect an attacker before the system output is generated. This scenario is summarized in Property 5.

Property 5. *If the attacker-controlled internal job $\tau_{i,k}$ and the target output job $\tau_{j,l}$ obey the following relations:*

$$a_{i,k} < a_{j,l} \text{ and } d_{i,k} > d_{j,l}, \tag{11}$$

where $a_{i,k}$ and $a_{j,l}$ denote the arrival times of $\tau_{i,k}$ and $\tau_{j,l}$, respectively, there is no guarantee that the attacker job will be detected before the output of the system is made when scheduled using EDF.

We now elaborate on the root cause of this race condition, which exists only because security tasks are added. Consider the data-dependency DAG in Figure 5(i), which represents a system with 4 tasks such that tasks A and C are internal and task B and D are output tasks. Such data dependency can be removed in our model (Section 5.1), which makes similar assumptions as in the LET paradigm, since the output job consumes its input from shared buffers at arrival. However, the addition of security tasks result in a DAG similar to the one shown in Figure 5(ii). This is to account for the verification that the security tasks must complete by the time the system output is generated.

Since an internal job and corresponding security job are released at the same time, the scheduler must allow any security job that has been released to execute to completion. To enforce this requirement and eliminate any potential race condition, we model each security task as sharing a logical resource with a corresponding output task. Therefore, each tuple $(\tau_i, \tau_{s,j})$, $\tau_i \in \mathbf{O}$ where $\mathbf{O}$ is the set of output tasks, shares a logical resource, $\mathbf{R}_{i,j}$ with a security task $\tau_{s,j}$ corresponding to an internal task.

7.2 Scheduling and correctness analysis

By explicitly defining a shared resource between security and output tasks, the race condition can now be prevented by utilizing any shared resource-aware scheduling algorithm such as EDF+SRP [9]. The security job must "capture" the shared resource when it arrives. If an output job arrives such that it satisfies the race condition (Property 5), it is blocked until the pending security jobs "release" the shared resource. The security jobs "release" the shared resource after they have verified all logged control-flow transfer events. The scheduler waits for these "release events" before scheduling the output job. This solves the race condition. Note that we utilize such a mechanism over considering ordered dependencies [19] since a security job may or may not be considered a "predecessor" for an output job depending on the respective arrival/deadlines (see (11)).

Other cases that can occur but are implicitly handled by any shared resource-aware scheduling algorithm are:

(1) The internal and security job have higher priority than the output jobs. In this case, the security job will always complete before the output job executes.
(2) The internal and security job arrive later and have a later deadline/lower priority. Under our model, the output job consumes its input from any shared buffers at arrival. It will also execute till completion (including preemption by higher priority jobs) before the attacker-controlled internal job gets to execute.

By explicitly considering the dependency between security tasks and output tasks, we also improve security by reducing the attack surface, since only the security tasks and scheduler communicate via the shared resource and both of these entities execute within the secure domain of TrustZone (Section 8.2), which is inaccessible to the attacker under our threat model (Section 3.3).

Summarizing our discussion: for a set of application tasks, we have corresponding security tasks. Each security task can be represented by a tuple ($C_{s,i}$, $D_{s,i}$ and $P_{s,i}$), where period $P_{s,i}$ is the same as the corresponding application task's period and $D_{s,i}$ is derived using Property 4. Therefore, the total system utilization increases from the original $U = \sum \frac{C_i}{P_i}$, to U_{total} given by:

$$U_{total} = \sum \frac{C_i^{'} + C_{s,i}}{P_i}, \tag{12}$$

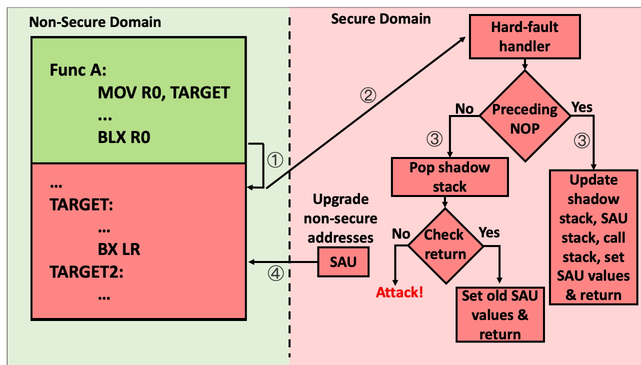

Figure 6: SAU based function-call enforcement.

where C_i' includes C_i and the in-line overhead of our mechanism. See Section 8.1 for implementation details and Section 10 for experimental results.

8 PROCRASTINATING CFI MECHANISM

The task model presented in the Section 5 permits deadlines of security tasks to be opportunistically relaxed so that their execution may be delayed. However, there are two implicit requirements: a) logging of control-flow events, and b) the security tasks must be given an opportunity to execute regardless of whether an attack occurs. For a) we present a novel *function-block* level CFI enforcement that is able to log forward control-flow events and enforce backward-edge CFI on existing off-the-shelf hardware. A function block includes a function and its prologue and epilogue (compiler-generated additional code). For b) we move the scheduler into the secure domain to ensure the integrity of secure task scheduling.

We utilize TrustZone for ARMv8-M detailed in Section 3.1. Note that "TrustZone" can refer to two different architecture extensions that allow creating a trusted execution environment on ARMv8-A and ARMv8-M processors. TrustZone for ARMv8-M mimics the simplicity of an MPU commonly found in microcontrollers. Unlike in ARMv8-A [33], prior work [31] shows that switching between secure and non-secure domains takes just a few CPU cycles, is highly predictable, and is triggered by a single *secure gateway* (`SG`) instruction. Our mechanism takes advantage of this low overhead and requires minimal modifications to the application code.

8.1 SAU based function-block enforcement and shadow stack

We now present our approach for bare-metal systems. Since we target resource-constrained systems, we limit our approach to function blocks, reducing the storage needs of call logs and in-line instrumentation overhead. As stated in Section 3.3, we assume that the attacker targets indirect branches for control-flow hijacking attacks. Compilers tend to convert function calls to direct branches (corroborated by our case study in Section 10.1) since they execute faster, therefore, it would be prudent to check indirect function calls only after they take place. Since we are only aware of attacks where multiple functions are required for an attack [35], indirect calls to locations within the same function can be ignored.

Detecting function blocks during runtime can either be done via custom compilers (employed by some prior work) that label functions, or padding the function blocks with NOPs or magic (non-instruction) values such that they can be detected by code during execution. We align and pad function blocks with one of ARM's NOP equivalents via a command to the linker program.

Figure 6 provides an overview of our approach. The SAU upgrades all but the currently executing function from the non-secure to secure domain. Since (1) a function call involves a branch to this upgraded memory, (2) it is cross-domain access that triggers a hard fault (Section 3.1). (3) The HardFault handler checks whether the target of the cross-domain access is the start of a function block. If so a) the current SAU values and return address are stored on the SAU and shadow stacks, respectively and b) the SAU is updated to shift the non-secure window over this function. The SAU values are calculated at runtime by scanning for padding to find the start of a function block. When a function returns, the return address is verified against the expected address from the shadow stack. The previous SAU configurations are then popped from stack into the SAU. Any verification failure is detected as an attack. (4) The memory access right is then enforced by new SAU configurations.

8.2 RTOS modifications

We now apply the scheme in Section 8.1 to an RTOS to complete our CFI mechanism (Figure 6):

RTOS components in secure domain- The RTOS scheduler and timer are moved into the secure domain. Many RTOS, such as FreeRTOS, already maintain an additional per-task stack in the secure domain to save secure function call execution state across context switches. As shown in Figure 3, we migrate the scheduler into the secure domain, and flip the utility of the task stacks by creating shell tasks that utilize the secure domain stack as the "main" stack. These shell tasks use the `BLXNS` instruction to the branch start of the task's application code in the non-secure domain. The scheduler, as before, handles both the non-secure and secure stacks with minimal modifications. Note that on ARMv8-M, there is no performance difference between the two domains and switching between them takes only a few clock cycles [31]. Therefore, these modifications do not introduce any significant performance penalty.

Implement stack support - SAU values, the control-flow log, and return address shadow stacks are created per task. The hard-fault handler is notified of the currently executing task by the RTOS and updates the correct stacks during runtime.

Implement security tasks in secure domain - These tasks directly accesses the call log stacks.

Other modifications - As stated in Section 5.1, we assume that only output tasks can *write* to system output (for example, the UART output buffer). This can be achieved by rearranging all code that accesses output peripherals into a memory region that is guarded by a single SAU region. The scheduler can disable/enable this SAU region when context switching to/from an output task, by flipping a single bit, ensuring a hard-fault occurs if accessed by other tasks.

8.3 Design Alternatives

A drawback of utilizing a fault handler for CFI is that the hard-fault handler (or `SecureFault` handler - see Section 3.1) has the highest

priority in the system, and could cause priority inversion, e.g., a ready-to-run high priority application task must wait while the hard-fault handler, triggered by a function call in a lower-priority task, executes. While the fault occurs only at function calls/returns, the number of function calls often depends on the coding style of the developer. Excessive function calls can impose non-trivial overhead in our current design to track the function-level control flow. Fortunately, this can be kept in check by leveraging compiler optimizations. Reducing the number of function calls also mitigates the degree of path explosion in control flows, at the expense of coarser-grained control. This provides an opportunity to trade-off performance and security along the dimensions of code size, number of functions, granularity of CFI, as well as run-time overhead. Our experimentation data (Section 10) shows that the total overhead of the fault handler is small (our implementation requires a total of $\approx 50 - 350$ cycles on our test hardware. However, checking for an indirect call takes just ≈ 60 cycles (≈ 5 us). It is possible to further reduce the overhead by only analyzing the transitions that involve safety-critical functions. While we intend to address this overhead in the future, using the fault-handler approach allows greater flexibility to implement complex runtime CFI not possible via binary instrumentation. For systems where this overhead is not acceptable, an instrumentation-based approach can be used. Note that either approach can be used with our task model.

8.4 Verification of Control Flow in Security Task

As discussed in Section 2, different CFI mechanisms exist in literature that trade off security and performance: e.g., verify the past N control-flow events together such as in PathArmor [38] where it performs a depth-first search to find a matching path within the CFG of the application. Security tasks in our framework can straightforwardly implement these techniques and our scheduling strategy balances overhead due to security and schedulability.

9 SECURITY ANALYSIS

We analyze our mechanism in the context of our threat model (Section 3.3). Moving the scheduler (and timer) into the secure domain ensures that it cannot be modified by the non-secure domain. This ensure that the scheduler can be trusted to correctly execute and report the currently running task to the hard-fault handler. Similarly, the attacker cannot access the shadow, SAU, or the call log stacks saved in secure domain. The call logs stack can be a circular to allow security tasks to quickly detect an ongoing attack.

Since the hard-fault exception has a higher priority than other peripheral interrupts, the integrity of any updates to the SAU, shadow, or call log stacks is guaranteed. Similar guarantees can be afforded to the timer interrupt by allotting it the next lower priority.

As discussed in Section 3.1, the Cortex-M23 architecture, unlike Cortex-M33, utilizes the `HardFault` for both TrustZone violations, and generic hard faults. Therefore there could be difficulty in differentiating between a system fault and an attacker-influenced fault. We address this issue by determining whether the Link Register (LR) is updated during the function call. This is because function calls are implemented via the `BL` or `BLX` instruction which update the LR. If the LR is not updated, it could be due to a system fault. Faults that modify the link register alongside a `BL` or `BLX` instruction, and force a branch to secure domain, are outside the scope of our work and would require special forensic analysis. Other instructions such as `BX LR`, `POP PC` or `MOV PC, <reg>` are sometimes used for returns but never for legitimate function calls. If the handler detects their use for a forward-edge flow transfers, they are treated as malicious.

We require all application tasks to have corresponding security tasks (Section 5.2). The system designer may relax this requirement.

Table 1: Control-flow transfers in popular CPS.

Software	Forward (% total)	Backwards (% total)	Total
ArduCopter	2139 (0.72%)	7160 (2.41%)	296581
PX4	6211 (0.08%)	44003 (0.61%)	7178749
TurtleBot	647 (0.25%)	1130 (0.44%)	255680
Rover	4 (0.41%)	8 (0.83%)	964

10 EVALUATION

We now present microbenchmarks of our implementation on hardware to compare against prior work, and then evaluate approach via simulations.

10.1 Control flows in Cyber Physical System (CPS) Software

Code used in CPS predominantly follows a predictable control flow sequence. To gain a realistic understanding of the expected overhead of control flow integrity in software programs for CPS, we developed a custom back-end pass in the LLVM compiler suite [28] to count the percentage of indirect control flow transfers in the control loop of widely-deployed real-time CPS, including ArduCopter[3], Rover in ArduPilot[5], PX4[4], and TurtleBot[6]. As shown in Table 1, in real-time CPS, forward and backward control-flow transfers are equal or less than just 1% and 6%, respectively, in all programs.

10.2 Experimental setup

We utilize a Nuvoton NuMaker PFM-M2351 (ARMv8-M baseline architecture) development board [1] for our experiments. The onboard flash memory and RAM are 512 KBytes and 96 KBytes, respectively. The processor operates at 12 MHz. We modify FreeRTOS according to Section 8.2 to support our approach. Modifications include altering the base addresses used by the scheduler to swap the non-secure and secure domain stacks (≈ 20 instructions each for saving and loading the SAU stacks during context-switching), adding 15 lines of C code to initialize and manage the SAU and shadow stacks, and ≈ 20 instructions added during RTOS bootup. The majority of our modifications are in the hard-fault handler (≈ 180 lines of C and in-line assembly).

10.3 Overhead

As stated above, we aim to gauge the in-line overhead of our approach. The results are as follows:

RTOS - We modify the FreeRTOS scheduler to store and load the SAU values on a task context switch. We noted overheads of 42 and 45 CPU cycles for these operations, respectively. To lock/unlock output functions as stated in Section 8.2 for an output task, $\approx 5 - 10$ CPU cycles more would be required to flip the relevant SAU region enable bits. The scheduler operation takes a total of 357 cycles on

Table 2: Deadline relaxation of security tasks.

Task set utilization without security tasks	Average total utilization	Percentage of security task with deadline relaxations	Average percentage of deadline relaxation (95% CI)
0.1	0.14	25.52	3.09 ± 1.47
0.2	0.25	20.85	5.37 ± 2.74
0.3	0.36	22.22	4.84 ± 2.52
0.4	0.49	20.29	4.65 ± 2.44
0.5	0.60	21.54	4.08 ± 2.39
0.6	0.7	23.38	3.71 ± 2.0
0.7	0.82	22.54	2.97 ± 1.73
0.8	0.93	20.29	2.89 ± 1.74

our hardware, including the SAU value load/store operations. Therefore, our modifications add a 28% overhead to scheduler operation. While this may seem high, our scheduler overhead is lower than that of the closest prior work [39] (32%), while having a broader threat model (Section 3.3) and ability to support more advanced CFI techniques (such as context-sensitivity which requires knowledge of multiple control-flow transfers instead of just one) in the security task by opportunistically relaxing its deadline. In addition, we believe that this overhead is mainly due to the slow on-board memory of our hardware (see below).

Hard-fault handler - 257-326 cycles are required for validating a function call, storing the SAU and shadow stack values, and setting up the new SAU values. 60 cycles are required to determine whether a call occurred due to an indirect branch and log the start and end points. We noted a 53 cycle overhead for function returns.

10.3.1 Comparison with RECFISH. Our approach resembles, temporally, the in-line overhead of RECFISH [39]. However, it exceeds the capability of RECFISH, supporting the possibility of out-of-order CFI checks in security tasks. We report a lower scheduler overhead (28% vs RECFISH's 32%), but a higher function call overhead (386 cycles vs RECFISH's 317). These could be attributed to differences in target architecture. However, both approaches have the similar points of interest, i.e., they operate during a context switch and during an indirect jump. In fact, our approach incurs an overhead during normal operation *only* when an indirect branch calls a function. Therefore, we believe the schedulability study for RECFISH also applies to our approach and acts as an upper bound. We also believe that with comparable system memory and flash, our approach will have a much lower total overhead.

10.3.2 A note on memory overhead. Our approach requires stacks in the secure and non-secure domain. This is *not* an overhead we introduce. FreeRTOS' default design includes these stacks for supporting TrustZone operation, such as to save context of a secure function call during regular execution. We simply re-purpose these stacks when we move the scheduler into the secure domain.

For each function call, 32 bits are required to store the target address – consumed by the security task – and the return address in the shadow stack. If compiler support exists, a shorter unique identifier can be used per function (similar to most forward-edge CFI [39]). Our case study in Section 10.1 shows that indirect calls are very infrequent; the call log stack can therefore be very shallow. Correctly determining the stack size to aid attack detection in case of an overflow during runtime is outside the scope of this work.

Optionally, the SAU start and end addresses are 32 bits each and are stored and popped during function calls and returns, respectively. In severely-resource constrained systems, these can be eschewed at the cost of recalculation during runtime at function returns. We assume that there is enough memory for these.

Table 3: Percentage of tasksets (/10,000) that are only schedulable with deadline relaxations under EDF+SRP. Original task set utilization does not include security tasks. WCET Ratio is ratio of WCET of security tasks to application tasks.

WCET Ratio \ Original System Util	0.85	0.9	0.95
0.1	14.35%	61.67%	54.09%
0.2	58.44%	45.69%	27.40%
0.3	37.41%	17.85%	6.77%
0.4	14.15%	4.54%	0.95%
0.5	3.38%	0.64%	0%

10.4 Simulation study

We now provide a simulation study to assess the real-time aspects of our approach. We a) show that security task deadlines can be relaxed significantly, and b) improve resource usage, which is especially important in overload situations.

10.4.1 Deadline Relaxation. We evaluate the effectiveness of our approach in relaxing security tasks deadlines. The results are presented in Table 2. The task set utilizations, before the addition of security tasks, are recorded in the first column. Each task set consists of 10 application tasks (2 are output tasks), generated using the UUnifast [11] algorithm implemented under the SchedCAT simulation suite [14]. Output tasks have the longest periods of the task set (Section 3.2. We then add security tasks to create task sets of a total of 20 tasks. The WCET of security tasks is calculated as 10% of corresponding application task. We believe this is realistic for forward-edge CFI checks since indirect branches calling functions are usually sparse (RECFISH considers 1 indirect branch for 10^3-10^7 CPU cycles) as corroborated by our case study presented in Section 10.1. An average of 20-25% of security task deadlines can be relaxed by up to ≈ 8% (minimum of 0.1%) of their deadline showing a clear avenue to defer CFI.

10.4.2 Improved Schedulability. Since security task deadlines can be relaxed, it is possible to increase security task load (modeling complex CFI operations) and still maintain schedulability. To simulate this, we generate task sets with utilizations of 0.85, 0.9 and 0.95. We then add security tasks, where the WCET ratio of security tasks to application tasks ranges from 0.1 to 0.5, to see system behavior under borderline overload conditions. We generate 10,000 unique task sets, each consisting of 8 internal tasks and 2 output tasks, per experiment, and summarize the results in Table 3. Accounting for rounding errors in the utilization of each task set, each data point is the number of task sets that cannot be scheduled by EDF+SRP unless the security tasks deadlines are relaxed. When the original task set utilization is 0.85 and the WCET ratio increases from 0.1 to 0.2, we see an increase in number of tasks sets that are only schedulable when deadline relaxations are considered. We then observe that the effectiveness of our approach diminishes at higher utilizations. Nonetheless, it is clear that our approach can support more complex CFI and/or other security mechanisms within the security tasks without compromising the schedulability of the system.

11 CONCLUSION

In this work, we provided an integrated real-time/security framework to support a hybrid in-line and out-of-order CFI for real-time applications. Our novel task model allows for the CFI portion of a real-time task to be decoupled and independently scheduled while ensuring timeliness in both attack detection and in execution. Our framework is supported by a novel trusted scheduling infrastructure that ensures the integrity of the real-time scheduler, records control-flow decisions that can be utilized by any forward-edge CFI mechanism, and implements a shadow stack to defend against backward-edge attacks. Experiments showed that our mechanism has in-line overhead similar to the closest related technique even when more complex/sophisticated CFI are used, and simulation results confirm our deadline relaxation technique is able to reduce resource usage and thus able to schedule tasksets that would have been deemed unschedulable otherwise. In the future, we plan to combine our scheduling framework with modern hardware architectures, built specifically for CFI, such as that presented in [36].

ACKNOWLEDGMENTS

We thank the reviewers for their feedback. This work is supported in part by US National Science Foundation under grants CNS-2038995, 2038726 and 1941524, and by the Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber R&D, innovation and workforce development in Virginia. For more information about CCI, visit cyberinitiative.org.

REFERENCES

[1] 2019. *NuMicro M2351 series – a TrustZone empowered micro-controller series focusing on iot security.*

[2] 2020. *Clang 12 documentation.* Retrieved 2020-10-24 from https://clang.llvm.org/docs/ControlFlowIntegrity.html

[3] 2021. *ArduCopter.* Retrieved 2021-10-19 from https://ardupilot.org/copter/

[4] 2021. *PX4.* Retrieved 2021-10-19 from https://px4.io/

[5] 2021. *Rover.* Retrieved 2021-10-19 from https://ardupilot.org/index.php/slider/ArdupilotRovers/RubidiumRover

[6] 2021. *TurtleBot.* Retrieved 2021-10-19 from https://www.turtlebot.com/

[7] N. Almakhdhub, Abraham Clements, S. Bagchi, and M. Payer. 2020. μRAI: Securing Embedded Systems with Return Address Integrity. In *NDSS*.

[8] Amazon. 2020. Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. https://www.freertos.org/

[9] Sanjoy K Baruah. 2006. Resource sharing in EDF-scheduled systems: A closer look. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, 379–387.

[10] Nicolas Bellec, Simon Rokicki, and Isabelle Puaut. 2020. Attack detection through monitoring of timing deviations in embedded real-time systems. In *ECRTS 2020-32nd Euromicro Conference on Real-Time Systems*. 1–22.

[11] Enrico Bini and Giorgio C Buttazzo. 2004. Biasing effects in schedulability measures. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, ECRTS.* IEEE.

[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.*

[13] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. 30–40.

[14] B Brandenburg. 2013. SchedCAT: The schedulability test collection and toolkit.

[15] Robert Buecs, Pramod Lakshman, Jan Weinstock, Florian Walbroel, R. Leupers, and G. Ascheid. 2018. Fully Virtual Rapid ADAS Prototyping via a Joined Multi-domain Co-simulation Ecosystem. In *VEHITS*.

[16] 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 Survey* (2017).

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

[18] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. 2011. Comprehensive experimental analyses of automotive attack surfaces.. In *USENIX Security Symposium.*

[19] Houssine Chetto, Maryline Silly, and T Bouchentouf. 1990. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems* 2, 3 (1990), 181–194.

[20] Terry D Day and Sydney G Roberts. 2002. A simulation model for vehicle braking systems fitted with ABS. *SAE Transactions* (2002), 821–839.

[21] 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 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[22] 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 *IEEE Symposium on Security and Privacy (SP).*

[23] Intel. 2020. Control-flow Enforcement Technology Specification. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[24] Marine Kadar, Gerhard Fohler, Don Kuzhiyelil, and Philipp Gorski. 2021. Safety-Aware Integration of Hardware-Assisted Program Tracing in Mixed-Criticality Systems for Security Monitoring. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 292–305.

[25] Christoph M Kirsch and Ana Sokolova. 2012. The logical execution time paradigm. In *Advances in Real-Time Systems.*

[26] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2020. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 39–53.

[27] Sreenath Krishnadas. 2016. *Concept and Implementation of AUTOSAR compliant Automotive Ethernet stack on Infineon Aurix Tricore board.* Master's thesis. Technische Universität Chemnitz.

[28] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[29] 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.

[30] Microsoft. 2022. Control Flow Guard - Win32 apps. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

[31] Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. 2020. TEECheck: Securing Intra-Vehicular Communication Using Trusted Execution. In *Proceedings of the 28th International Conference on Real-Time Networks and Systems.*

[32] Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. 2022. Survey of Control-Flow Integrity Techniques for Real-Time Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)* (2022).

[33] Anway Mukherjee, Tanmaya Mishra, Thidapat Chantem, Nathan Fisher, and Ryan Gerdes. 2019. Optimized trusted execution for hard real-time applications on COTS processors. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*. 50–60.

[34] 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.*

[35] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE Symposium on Security and Privacy.*

[36] Gabriele Serra, Pietro Fara, Giorgiomaria Cicero, Francesco Restuccia, and Alessandro Biondi. 2022. PAC-PL: Enabling Control-Flow Integrity with Pointer Authentication in FPGA SoC Platforms. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 241–253.

[37] 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.*

[38] 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.*

[39] 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 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[40] Joseph Yiu. 2009. *The definitive guide to the ARM Cortex-M3.* Newnes.

[41] Joseph Yiu. 2015. ARMv8-M architecture technical overview. *ARM WHITE PAPER* (2015).

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

[43] 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).*