InsectACIDE: Debugger-Based Holistic Asynchronous CFI for Embedded System

Yujie Wang*, Cailani Lemieux Mack†, Xi Tan‡, Ning Zhang*, Ziming Zhao‡, Sanjoy Baruah*, Bryan C. Ward†
* Washington University in St. Louis, † Vanderbilt University, ‡ University at Buffalo

***Abstract*—Real-time and embedded systems are predominantly written in C, a language that is notoriously not memory safe. This has led to widespread memory-corruption vulnerabilities in real-time embedded cyber-physical systems (CPS). This is concerning, as such devices are becoming increasingly networked with the Internet of Things (IoT) and other communication technologies (e.g., 5G), rendering them vulnerable to remote attacks. Attackers have demonstrated how memory-corruption vulnerabilities can be used to hijack program control flow to implement arbitrary attacker-controlled logic. One class of defenses that has been developed to prevent such attacks is called *control-flow integrity* (*CFI*), which applies checks at control-flow transitions to ensure the target is valid. Unfortunately, attackers have shown how to divert control flow to seemingly valid targets in an invalid and malicious sequence.**

This paper presents InsectACIDE, the first holistic CFI for embedded and real-time systems that does not require binary instrumentation and that is context sensitive, i.e., it checks that the sequence of control-flow transitions taken is valid, not just individual transitions, thereby detecting such attacks. InsectACIDE is implemented on an embedded Cortex-M processor using the TrustZone trusted execution environment, and holistic context-sensitive CFI is enforced for both applications and the kernel. InsectACIDE uses hardware debugging features on the Cortex-M processor and therefore does not require any kernel or application binary modification. Experimental results show that InsectACIDE incurs significantly less runtime overhead compared to the state-of-the-art holistic CFI solution. Real-time schedulability analysis is presented, along with a schedulability evaluation, to demonstrate the tradeoff between stronger protection and real-time schedulability.

I. Introduction

Embedded and real-time systems are predominantly developed in C, which is notorious for being riddled with memory-corruption vulnerabilities, such as buffer overflows [1]. Microsoft and Google have both independently reported that memory-corruption vulnerabilities account for approximately 70% of the vulnerabilities in their software that is based on memory-unsafe languages like C/C++ [2], [3]. Unless or until such time as all code can be re-written in a memory-safe language like Rust[1], we are forced to reconcile the security of such unsafe code by developing defensive techniques to prevent exploitation based on such vulnerabilities. This is especially true as embedded and real-time systems are becoming increasingly internet-connected through technologies like 5G and the (Industrial) Internet of Things or (I)IoT, making such vulnerabilities easier to exploit by a remote attacker.

Memory-corruption vulnerabilities may take many forms, such as a buffer overflow, underflow, or use-after-free vulnerability, among many others. Work on memory-corruption security therefore considers a strong threat model in which an attacker is assumed to be able to "write what where," or write arbitrary data to an arbitrary point in a writeable data region (e.g., stack or heap). Over the years, attackers have become increasingly sophisticated in their use of such vulnerabilities to execute malicious logic, even if they cannot modify the code itself due to being marked as not writeable [1]. These attacks include code-reuse attacks, such as return-oriented programming (ROP) [4], in which an attacker corrupts addresses in a data space, such as a return address, to divert control flow to attacker-chosen code. Indeed, attackers have demonstrated sophisticated techniques that reuse small snippets of existing code in clever ways. These classes of attacks are broadly called *control-flow hijacking* attacks.

Control-flow integrity (CFI) has emerged as one of the most effective techniques to protect systems against control-flow hijacking [5]–[8]. In CFI, critical control-flow transitions, such as invoking function pointers or function returns, are instrumented with checks to determine whether the target address is valid based upon *a priori* static code analysis. However, attackers have devised increasingly sophisticated techniques that 'bend' the control flow to seemingly valid targets in an invalid and malicious sequence [7], [9], [10]. To mitigate this, the runtime context is used in context-sensitive CFI to further bound the attacker by reducing the equivalence classes of the branching destination. In other words, instead of only checking individual control-flow transitions, instead, sequences of control-flow transitions are checked to be valid or not. Yet, the added security benefit from context-sensitive CFI potentially also comes with significant additional overhead due to the added complexity of checking sequences of control-flow transitions instead of individual ones.

In this paper, we propose to leverage hardware debugging features commonly available in Arm systems to decouple the process of context recording and context verification, reducing the computational overhead through eliminating the worst-case execution-time (WCET) expansion from in-line context-sensitive control-flow verification. Our proposed technique is called InsectACIDE[2], as it uses the hardware-debugging features to implement an $\underline{\text{A}}$synchronous $\underline{\text{C}}$ontrol-flow $\underline{\text{I}}$ntegrity

[1]A lofty goal given the vast amount of established code bases and continued development in unsafe languages given developer expertise and/or preference.

[2]InsectACIDE is intentionally misspelled, but stops cyber varmints in their (malicious control-flow) tracks, nonetheless.

Defense for Embedded and real-time systems.

This hardware-based approach also introduces further opportunities to harden embedded systems to control-flow hijacking. The hardware debugger traces all control-flow transitions in the system, including between and among privileged (kernel) and non-privileged (application) modes. InsectACIDE therefore is holistic in that it checks both userspace and kernelspace code for control-flow hijacking. The InsectACIDE hardware-based approach is also novel in that it does not require any binary modification to either the kernel or applications. This is beneficial to real-time and embedded systems for several reasons. First, the whole memory layout is often carefully crafted to maximize cache affinity or otherwise be more amenable to WCET analysis. Second, embedded and real-time systems often have highly customized and board-specific toolchains, which could render it difficult to integrate with security instrumentation.

Recognizing the need for stronger system protection, there have been several works that explore the use of hardware debugging feature for CFI protection [6]–[8], [11]–[16]. However, most prior work focuses on protecting server platforms, except a closely related technique called SHERLOC [8], a system that uses hardware-debugging features on Arm Cortex-M platforms to enable CFI. However, SHERLOC only provides basic CFI without considering the execution context or the real-time implications [8], [17]. InsectACIDE aims to bridge this gap in making the hardware-assisted asynchronous CFI protection stronger by incorporating execution context and while also targeting real-time predictability. From the real-time-system perspective, the previous work of SHERLOC conducts security checking exclusively in the execution of interrupt handling, which is undesirable for real-time applications and analysis. Building on the observation that safety problems of cyber-physical systems manifest in the physical world, InsectACIDE adopts a new design paradigm where the timing of security-policy enforcement is governed by the timing of the physical world interactions (actuation more specifically), decoupling from the cyber state event (debug interrupt) as adopted by SHERLOC.

On the other hand, with all its advantages, it is important to note that this new design paradigm comes with a cost. Even though the new paradigm of decoupling the policy enforcement from the system behavior measurement generally applies to various system implementations, using hardware for control-flow-event recording imposes an implicit constraint, restricting it to platforms that support hardware-enabled program behavior monitoring. Furthermore, while the decoupling facilitates innovative ways to schedule policy enforcement, the reliance on triggers based on the physical world's impact limits its application to real-time CPS, and it also imposes an additional overhead on the computational load of the system. This paper makes the following contributions.

- We propose InsectACIDE[3], a novel context-sensitive, holistic asynchronous CFI for embedded and real-time systems that does not require binary instrumentation.
- We tackle several technical challenges to design InsectACIDE, integrating asynchronous checking into the scheduling, including minimizing the attack surface induced by asynchronicity, and minimizing system overhead.
- We implement a prototype on the Armv8-M architecture and evaluate its performance on FreeRTOS.
- We present a response-time analysis and schedulability evaluation to demonstrate the tradeoffs between hardware- and software-based context-sensitive CFI.

[3]The artifact and source code are available on the website: https://insectacide.github.io/.

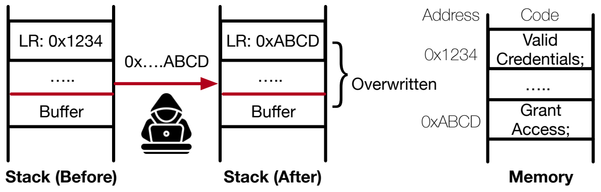

Fig. 1: Control-flow hijacking example.

II. Background

In this section, we review how memory corruption can be used by an attacker to hijack control flow, which we aim to prevent in our defense. We then review relevant background on the hardware features we leverage in our implementation.

A. Control-flow Hijacking Attacks

Control-flow hijacking attacks are a class of cyber attacks in which an attacker exploits a memory-corruption vulnerability to redirect control flow to execute code of their choosing. Given the extensive use of software and communication networks in cyber physical systems, the potential for malicious actors to tamper with the control flow of software components presents a tangible threat. The reliability and correct operation of these systems, especially safety-critical systems, are of the utmost importance to prevent the possibility of financial loss, injury, or even loss of life.

Figure 1 shows a simple example of a control-flow hijacking attack on Arm. In this example, there is a buffer-overflow vulnerability that is exploited by an attacker to write data past the intended buffer end and overwrite part of the stack including the Link Register (LR). Here, the original linked address in executable memory is `0x1234`, which points to a function to validate some credentials. However, the attacker is able to overwrite that address with `0xABCD`, which points to a function to grant access to some sensitive data. Thus, when the thread returns from the currently executing function, the access-granting function is run instead, and the integrity of control flow is violated.

There are two main classes of control-flow hijacking attacks [18]. The first is *code-injection attacks*, in which the attacker injects their own malicious code into the address space, for example via a buffer overflow, and then redirects control flow to that code. Thus, the hacker is able to execute arbitrary code. Many modern systems prevent this type of attack by enforcing permissions on code and data that writeable memory is never executed (DEP/W$\oplus$X) [1].

The success of defenses such as DEP/W$\oplus$X have forced attackers to evolve and create attacks in which they cannot inject malicious payloads, but instead must *reuse* existing code. Such attacks are called *code-reuse attacks*, and our previous example is an example of such an attack. We note, however, that our previous example was rather simplistic, and in practice attackers may not be able to divert control flow to a single function to achieve their malicious intents. Attackers have demonstrated how to corrupt multiple stack frames to "string together" a sequence of *gadgets* or snippets of existing code in order to execute arbitrary logic in a specific type of attack called a *return-oriented programming attack* [4].

B. Holistic Control-flow Protection and Context Sensitivity

CFI is a promising solution for protecting systems against control-flow hijacking attacks by enforcing that all control flows follow the control-flow graph (CFG) constructed from static analysis. While many CFI implementations can only protect non-privileged-mode programs [6], [7], [10], [12], [19], some of them can provide holistic protection for the entire non-secure-state program, including both non-privileged and privileged modes [8], [20], [21].

However, CFI checking can be coarse-grained due to the conservativeness of static analysis, such that the set of legitimate transfer targets, known as the equivalence class (EC), can be large, leaving potential space for attackers to maliciously swap targets within the EC undetected. Therefore, context-sensitive CFI has been proposed to leverage context information recorded during runtime, such as callsites and branch conditions, to provide more fine-grained protection [6], [7], [10], [12], [19].

C. Armv8-M Architecture

Memory protection. In Armv8-M, all memory, peripherals, and the processor's control registers share the same address space, i.e., there is no virtual memory. For memory protection, Armv8-M features a Memory Protection Unit (MPU), which is a hardware component that enables developers to specify the start address, length, and access permissions for memory regions. MPU includes MPU_S and MPU_NS to control access permissions for either secure or non-secure state. A permissions violation will result in a `MemManage` fault. TrustZone is an architectural security extension that enforces resource isolation between the untrusted normal world execution and the trusted secure world execution on Arm platforms. a memory region can be configured as secure, non-secure callable (NSC), or non-secure using a combination of the TrustZone hardware Secure Attribution Unit (SAU) and Implementation Defined Attribution Unit (IDAU) to specify the memory attributes.

Debugging features. The Armv8-M architecture features two hardware features we leverage in this work: a tracing unit called the Micro Trace Buffer (MTB) and a Data Watchpoint and Trace unit (DWT). Both the MTB and DWT exception handlers can be configured to be processed only in the secure state. The MTB captures all non-sequential program-counter changes on the microcontroller, including calls, branches, and exceptions. It stores trace records, i.e., source and destination address pairs of the non-sequential PC changes, in the trace buffer, a circular buffer within the SRAM area that can be configured as secure memory. When a predefined watermark is reached, the MTB can trigger a Debug Monitor (`DebugMon`) exception, which can be handled to process the control-flow data. The DWT provides special registers called comparators that can monitor instruction executions and data operations to specified addresses and trigger a `DebugMon` exception when there is a match. Therefore, by configuring these comparators to specific addresses, the DWT can implement program breakpoints.

III. Threat Model

In this work, we focus on detecting control-flow hijacking attacks within the entire non-secure state program. We adhere to the common threat model used in prior CFI work that aim to defend against such powerful attacks [8], [20], [21]. Similar to other security works that leverages trusted execution environment [22]–[24], we assume the presence of vulnerabilities within the non-secure-state program that allow attackers to arbitrarily read and write data memory, and can use code-reuse attack to achieve arbitrary code execution. However, we assume the secure world and hardware are vulnerability free. Furthermore, code injection in normal world is prevented using existing memory-protection technologies, such as the MPU. Different from existing CFI solutions with kernel protection [25], [26], InsectACIDE does not assume the application and kernel are running as a single binary in the privileged mode. Similar to previous CFI works [5], [8], [20], [21], [25]–[27], we do not consider non-control data attacks [9], [28], such as Data-Oriented Programming [29], that corrupt non-control variables, potentially increasing the tasks' WCET. Similarly, hardware attacks [30], side-channel attacks [31], and availability attacks [32]–[34] are out of scope.

IV. InsectACIDE Design

In this section, we will first summarize the design goals then explain how the InsectACIDE achieves those goals.

A. Design Goals

G1 ***Context-sensitive control-flow integrity.*** *In order to mitigate advanced control-flow hijacking attacks, such as control jujutsu [35], context-sensitive control-flow integrity should be enforced.*

Traditional CFI approaches, including those described in Sec. II, check that each individual control-flow transition is valid. However, it is possible to construct a malicious sequence of control-flow transitions made out of individual transitions that are in fact valid [9], [35]. A stronger form of control-flow integrity therefore checks that the *sequence* of control-flow transitions is valid, rather than simply individual ones.

This is known as context-sensitive control-flow integrity [7], [10], [12], [14], and it checks not only whether an individual control-flow transition is valid, but if it is valid within the given context. Context-sensitive control-flow integrity is therefore a stronger form of protection than ordinary CFI.

G2 ***Holistic protection.*** *Control-flow protection should be provided for all non-secure-state code, including both non-privileged (applications) and privileged (kernel) code.*

CFI is usually applied at compile time and therefore only protects an individual application. While there have been several efforts to include CFI in the kernel [5], [8], [20], [21], [25], [26], InsectACIDE aims to provide holistic protection that checks all application code and kernel code while supporting the application and kernel privilege separation.

G3 ***No binary modification.*** *For compatibility, there should be no required binary modification of the protected code.*

Much previous work on CFI has applied compile-time transformations to insert checks in the binary. However, this requires recompiling the application and may invalidate the WCET analysis of the application. Furthermore, many embedded systems use highly customized, often vendor-specific toolchains to build and deploy software. Preserving the binary allows transparent implementation of CFI agnostic to specific toolchains.

G4 ***Real-time adaption.*** *To satisfy real-time requirements, the system should be designed for real-time predictability.*

Using the MTB for tracing events allows for the asynchronous processing of integrity checks. Our goal is to enable integrity checks during idle time, where possible, while also supporting real-time schedulability analysis.

G5 ***Control-guided protection.*** *Control-flow protection must be enforced before physical-control outputs are allowed.*

Our context-sensitive control-flow protection decouples the recording and checking of control-flow transitions. This could in theory create a window of opportunity for an attacker to achieve malicious effects until the time of check. To mitigate this threat, we require that checks are completed before any control outputs for a given task are allowed. We must therefore intercept such events and enforce that all checks are completed so as to prevent any malicious control actuation.

B. InsectACIDE Architecture

An overview of InsectACIDE is shown in Fig. 2. The hardware MTB (recall from Sec. II) is used in order to record control events without any software instrumentation (G3). The MTB also records control events in both user and kernel mode, allowing for holistic control-flow protection (G2). In order to fully implement holistic control-flow protection, we must check all control-flow events, which requires periodically processing the MTB contents, and we design our implementation so as to minimize the analytical worst-case interference for such processing triggers to improve real-time predictability (G4). Finally, we use TrustZone and the debugging features (recall from Sec. II) to intercept control and switch to the secure world at any control-output time (G5), and finish all necessary checks for context-sensitive control-flow protection

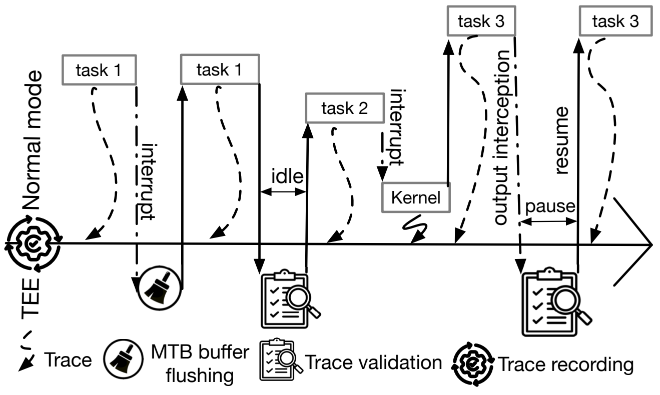

Fig. 2: Overview of InsectACIDE.

(G1). In the rest of this section, we elaborate upon the details of this design, and in Sec. V, we describe salient implementation details necessary to realize this architecture.

Hardware-assisted control-event recording. The MTB records all control-flow events on a core for all privilege levels, including both non-privileged and privileged execution modes. The traced control-flow events are stored in a secure memory buffer in the TEE, making them immune to attacks from either a malicious application or a corrupted kernel. The size of the MTB is configured through a specific register, referred to as `MTB->WATERMARK`, the maximum buffer size varies on different platforms (4KB in our evaluation). Once the buffer is full, an exception is raised that is handled in the TEE. This serves as an InsectACIDE entry point.

Context-sensitive control-flow protection. When an exception is raised, the recorded trace data must be copied from the buffer to device memory before returning to normal mode for later context-sensitive checks. We note that the MTB trace data contains all non-sequential control-flow events, and this may include events from multiple separate processes and context switches into and out of the kernel to handle system calls or other interrupts. In comparison to traditional single-event CFI, in our context-sensitive control-flow protection we check a sequence of control-flow events. As that sequence is often process-specific, an incorrect sequence can lead to false positives. Therefore, when parsing the MTB trace data we must demultiplex the single stream of control-flow events into per-process and kernel-related control-flow events so that they can be checked separately.

A straightforward method, similar to prior work [8], is using the address ranges of each application for demultiplexing (recall, our target platform is an Arm Cortex-M microcontroller, which has only an MPU with a single linear address space, rather than an MMU and virtual memory). However, such demultiplexing can be incorrect because some processes may share code segments due to shared libraries or kernel function calls. To demultiplex control-flow events correctly in order to ensure the soundness of context-sensitive checking, InsectACIDE adds additional procedures during trace process-

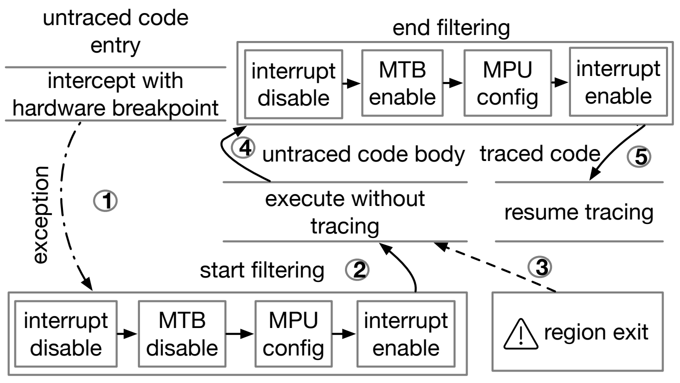

Fig. 3: Binary-preserving selective tracing.

ing by identifying context-switching from events of interrupts and the kernel, and managing trace data with an assigned process identifier, as detailed in Sec. V.

Asynchronicity-induced attack surface. By deferring checks until control output (G5), the attacker could theoretically cause a series of malicious control-flow events that are not checked until after an attacker has accomplished their malicious intent. To ensure that attackers cannot exploit this time window, InsectACIDE enforces that all checks must be completed before critical operations. Critical operations can include accessing sensors and actuators. On Cortex-M microprocessors, accessing peripherals is similar to accessing specific memory addresses. A straightforward approach is to use software instrumentation to insert reference monitors to track access to these addresses and verify the status of trace checking when encounters occur. However, this approach requires binary instrumentation, violating G3. InsectACIDE instead leverages the hardware features of the DWT. Specifically, InsectACIDE can set the comparators of the DWT to specific data or code addresses to trigger an exception. Both exceptions can then be trapped in the TEE, and the status of trace checking can be subsequently verified. Similarly, since the MPU_NS can configure the access rights of memory regions, InsectACIDE configures memory regions for these peripherals to be read-only using the MPU_NS. Consequently, every write operation to them will trigger an MPU fault exception, captured by the secure state via setting a DWT comparator to the exception handler entry. As a result, because MPU_NS, DWT, and MTB configuration can only be made within TrustZone secure mode, attackers cannot exploit the asynchronicity time window to bypass checking.

Binary-preserving scheduling integration. In order to minimize the real-time impacts of InsectACIDE (G4), we perform as much trace processing as possible during otherwise idle time. While conceptually straightforward – process and check any MTB contents at idle so that new tasks start with a fresh MTB – care must be taken to implement this without modifying the scheduler or other applications in the normal mode (G3). In FreeRTOS there must be at least one ready task at any point in time, including the idle task. If there is nothing left to process, the idle task spins in a busy loop waiting for an asynchronous event to trigger a new task. However, this busy-waiting loop rapidly fills the MTB as each invocation of the busy loop adds events to the MTB. This may leave the MTB nearly full when a new task arrives, requiring processing. Therefore, we need to disable the MTB during idle time after flushing its contents. However, we must also ensure that the MTB is re-enabled before any other tasks are allowed to run, otherwise those control events would not be captured or checked. Ideally, one can simply have the scheduler turn off MTB recording before switching to the idle task. However, this requires modification of the existing system either with source code modification or binary rewriting. Neither would meet the binary-preservation design goal (G3). Instead, InsectACIDE leverages the DWT feature to trap execution back to the secure world upon entry into the idle task. By processing the idle task inside the secure world, it is possible to ensure that MTB is turned off and on appropriately within the idle task. It also ensures that upon context switch MTB is turned back on, where the timer interrupt that initiates the scheduler is first processed by the secure world before passing it back to the normal world.

Binary-preserving selective tracing. While the MTB serves as a powerful tool for recording all control-flow events, it does not support selective tracing. In context-sensitive CFI, the useful control-flow events (or CFI-relevant events) are either security-related events that should be checked, including all indirect branches, or context-related events, such as the events of function calls (when choosing callsite as context). The recording of other events that are CFI-irrelevant imposes unnecessary overhead on trace processing. For example, as evaluated in Section VI, when choosing the callsite as the context, 52% of the events are neither security-related nor context-related. These events must be processed and filtered when flushing the MTB, which can introduce an additional runtime overhead of 48%.

To reduce the real-time impact, InsectACIDE uses a secure selective tracing approach that enables or disables the MTB in a binary-preserving manner. Specifically, for code segments that generate only superfluous events, InsectACIDE turns off the MTB during their execution. In practice, such code segments can be identified during WCET analysis, as analyzing the worst-case scenario requires a full understanding of the branching structures. More details on selecting such code segments can be found in Sec. V. However, even with the code segments selected, there are additional challenges due to the limited number of debugging breakpoint registers for detecting entries and exits. This is often exacerbated by the fact that many segments may not have well-defined exits. To tackle this, we propose to use the MPU, particularly the normal world banked register, MPU_NS to setup a sandbox by configuring any code outside of the segment as non-executable (NX), as illustrated in Fig. 3. Upon exiting the sandbox, a memory access fault will be raised by the hardware, trapping the execution of the system to the memory fault handler and

Algorithm 1: Region Selection

```
   Input: $regions,\ P,\ K$
   Onput: $s\_regs$          // selected untraced regions
1  $s\_regs \leftarrow map()$
2  for $r$ in $regions \cup \{entry\}$ do
       /* Select from regions reachable from r   */
3      $p, regs \leftarrow reachable(r, P)$
4      $scores \leftarrow$ [score($p$,$r'$,$regs$) for $r'$ in $regs$]
5      $s\_regs[r] \leftarrow topKregs(scores)$      // top-K regions

6  Function score($p$,$r$,$regs$):
7      $R_s \leftarrow \{\}$
       /* if r not dominated by any other region */
8      if $!domed(r,regs)$ then
           /* get regions dominated by r          */
9          $D_s \leftarrow domRs(r, regs)$
           /* select K regions in sub-program p
              from regions D_s by recursion       */
10         $R_s \leftarrow RegionSelection(D_s, p, K)[r]$
       /* reduced number of worst-case trace     */
11     return $\mathcal{W}(p) - [\mathcal{W}(p \backslash R_s) + \epsilon * |R_s|]$
```

the secure world. This empowers the secure world to re-enable the MTB monitoring even if there are vulnerabilities inside this code segment that might have given the adversary the power to run arbitrary code. As a result, this approach reduces the number of traced events without sacrificing security, thereby reducing the WCET of trace processing. The detailed security analysis is shown in Sec. VII.

V. Implementation

InsectACIDE is built on a single-core Arm Cortex-M33 platform, and the implementation contains four parts, control-flow analysis, runtime configuration, selective tracing, and trace processing.

Control-flow analysis. To validate control-flow transfers, InsectACIDE conducts context-sensitive control-flow analysis. Context can take various forms; in the prototype, function call sites are selected as the context. Static analysis can be applied at either the source-code level or the binary level, with the difference between the two lying in the additional effort of binary reverse-engineering, which can affect analysis precision. Then, the analysis results are encoded as a table for runtime reference.

Runtime configuration. There are several key system security configurations. During the system boot, InsectACIDE routes the interrupts of DWT and MTB into the secure state and configures the MPU_NS to ensure the code integrity of the entire system. Moreover, InsectACIDE sets DWT comparators at the entry of the MPU fault handler to enable the secure state to capture the non-secure state's MPU fault, as well as the idle task to detect system idleness. Such MPU_NS region and DWT watchpoint usage (out of the total 8 MPU_NS regions and 4 DWT comparators in our evaluation platforms) are minimally required for system implementation. In addition, InsectACIDE uses SAU and IDAU to configure the necessary address ranges relevant to the configurations of the above-mentioned hardware as secure memory ranges, accessible only by the secure state. Furthermore, the memory used by the secure world would be configured as secure memory using SAU and IDAU to specify the secure world attribute to prevent malicious tampering from DMA.

Selective tracing. The key idea of selective tracing is to disable MTB on certain code regions in which no CFI-relevant events will be generated. However, such mechanism requires not only limited hardware resources (DWT comparators and MPU regions) to track the entry points, but also the MPU-enforced sandbox to detect exits and re-enabling the MTB. This poses a non-trivial problem of determining "which K code regions" should be monitored. In this problem, limited breakpoints should be utilized to detect code regions that have a higher probability of being executed soon and can yield enough performance gain by turning off MTB even with the overhead of MPU sandboxing configuration.

The first step is to identify the candidate regions where performing selective tracing will be beneficial. The idea is to identify the code regions in the control-flow graph that only contain CFI-irrelevant events, including direct and conditional branches (based on callsite as context). These partitioned code snippets are labeled as candidate regions if selective tracing offers performance benefit. To determine if there is performance benefit, InsectACIDE compares the estimated time to process the maximum number of traces that can be generated by the code snippet to the sandbox overhead. The worst-case traces are obtained from a tool we constructed, $\mathcal{W}$, based on aiT [36], an industry-standard WCET analysis tool, by finding the path that can generate the maximum number of control-transition instructions. Once the benefits of individual candidate regions are identified, the next step is to look into the temporal proximity of the code snippet. The allocation of limited hardware breakpoints to track candidate regions can be formulated as an optimization problem, which InsectACIDE leverages a greedy algorithm based on a computed priority score, as described in Alg. 1. Specifically, the algorithm calculates a score for each reachable region at each potential reallocation point (including region candidates and task entry) to prioritize the region that offers the most benefits in reducing the total trace processing time. Intuitively, regions that always execute earlier (i.e., dominators) than others (i.e., dominance) are prioritized because their breakpoints can be reallocated to later ones after exiting. As such, a dominance region is assigned a 0 priority (Line 7 to 8), since breakpoints are re-assigned at its dominators' exit. The analysis results are encoded in a map at compile time.

Trace processing. Since selective tracing cannot filter out all CFI-irrelevant events, these events are filtered *on the fly* when reading from the MTB. Only the remaining trace data is moved to device memory for further processing.

Control-flow events are recorded indiscriminately by the MTB. To check a sequence of control-flow events, traces recorded by the MTB require additional processing to differentiate control-flow events between different tasks and interrupts. InsectACIDE follows the existing work [8] to differentiate interrupt events from others and handle (possibly nested) interrupts, ensuring that interrupt entry and exit addresses always match. InsectACIDE is built on top of this interrupt handling method to demultiplex events for different processes. Specifically, since most context switching is also triggered by interrupts, with the guarantee of consistency between interrupt entry and exit ensured by the prior approach, InsectACIDE can determine the entry points of process release and resume by comparing the task addresses. After the detection, InsectACIDE retains the process identifier and transfers all subsequent non-interrupt events to the corresponding process buffer until new interrupt events occur, requiring an update of the identifier accordingly. A corner case arises when two tasks may resume at the same address if context-switch interrupts happen to occur at the same code location, making it difficult to distinguish which task is currently executing. However, this issue can be resolved by considering the scheduling policy. For example, in a fixed-priority scheduler that uses a FIFO policy for tasks with equal priorities, the later preempted task always has a higher or equal priority than the one preempted earlier. Therefore, when the corner case occurs, the executed task is the one that was preempted later. As a result, control events, even for shared code, can be distinguished for different tasks. With the demultiplexed traces, InsectACIDE validates backward edges to enforce functions to return to their call sites in a manner similar to [8]. For forward-edge transfers, InsectACIDE uses trace data (context) stored in the corresponding buffer to retrieve offline-computed EC for validation. Once the validation is complete, the checked trace is removed from the buffer.

VI. Evaluation

The evaluation is conducted on the Arm Versatile Express Cortex-M prototyping system (V2M-MPS2+) [37], which is configured as a single-core Cortex-M33 microcontroller using the AN505 FPGA image [38]. The device is equipped with a 4MB code/flash region and 4MB SRAM, including a 4KB allocation for the MTB trace buffer.

A. Performance Evaluation

To measure the performance impact of InsectACIDE, we utilize real-time benchmarks to assess the system overhead.

Experiment setup. InsectACIDE is designed to work on embedded microprocessors running a real-time operating system; therefore, we selected programs in the TACLeBench benchmark [39] and integrated them into FreeRTOS. Since this section aims to understand the general system overhead, each benchmark program is evaluated individually as periodically released tasks. While Sec. VI-C assesses the security benefits of context sensitivity, the overhead of context-sensitive checks in this section is simulated at direct function call sites due to the absence of indirect function calls in the benchmark programs. Additionally, an operation that writes to a fixed memory location is inserted at the end of the program to simulate control outputs. We configure the DWT to set a

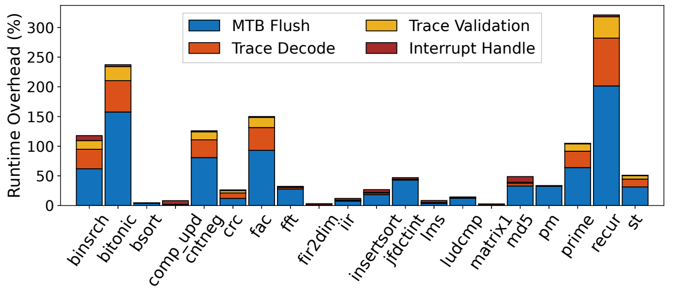

Fig. 4: Runtime-overhead breakdown.

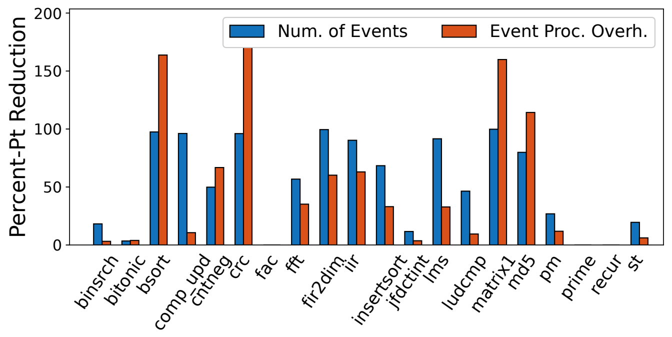

Fig. 5: Event quantity and event processing overhead reduction with selective tracing.

breakpoint to this memory location and allocate one of eight MPU regions for selective tracing.

Runtime overhead. The runtime overhead of InsectACIDE comprises four components, as shown in Figure 4.

Specifically, (i) MTB flushing involves reading from the MTB and pre-filtering events that are not of interest. (ii) Trace decoding is conducted asynchronously to further separate the pre-processed trace data into meaningful checking sequences based on different interrupts and tasks, which are then (iii) validated and removed. (iv) Interrupt handling encompasses the time spent on exceptions/interrupts related to the MTB becoming full, selective tracing, and real-time integration mechanisms. Handling these exceptions and interrupts includes hardware-configuration operations and entering and leaving the TEE.

MTB flushing and selective tracing – The majority of the runtime overhead is attributed to MTB flushing, resulting in an average runtime expansion of 51%. Further investigation reveals that this is mainly due to the speed limitation of the MTB. Reading from the MTB takes approximately twice as long as reading from device memory. The variation in MTB flushing overhead among different benchmarks is primarily linked to the utilization of selective tracing. As depicted in Figure 5, selective tracing, on average, can reduce the number of traced events and event processing overhead by 52% and 48%, respectively, highlighting the effectiveness of this approach. The overhead variation between Figure 4 and Figure 5 is highly correlated. The more effective selective tracing is, the lower the overhead of MTB flushing. Among the benchmarks, the program *recur* exhibits the highest MTB flushing overhead and the least effective selective tracing. This is because the program relies heavily on non-filterable recursive function calls rather than filterable loops. Consequently, filtering traces cannot offset the overhead incurred by dynamically turning the MTB hardware on and off.

Decoding, validation and interrupt handling – Asynchronous trace decoding incurs the second-largest overhead of 14%. This is because performing context-sensitive checking requires the separation of interrupts and different tasks. This separation involves not only identifying control-flow types from raw address data but also extracting context-switching events from traces. Subsequently, asynchronous control-flow checking incurs an overhead of 6% with the decoded traces. In terms of interrupt handling, InsectACIDE experiences a low overhead. This is primarily because the MTB, idle breakpoint, and selective-tracing runtime configuration handling occur infrequently compared to trace processing.

Comparison with alternative approaches - We evaluated InsectACIDE against both hardware-tracing-based and software-instrumentation-based alternative holistic CFI approaches designed for Cortex-M. We chose SHERLOC [8], an approach that also leverages the MTB, as the hardware-based comparison target. To compare with software-based approaches, we implemented callsite-sensitive CFI as well as a stronger variant, path-sensitive CFI. For path-sensitive CFI, our comparison focused only on recording, since checking is scheduled asynchronously, and the overhead of context recording includes the recording of all conditional branches, function calls, and indirect branches (i.e., if-else statements, function calls, and returns).

As shown in Figure 6, the average runtime overhead for InsectACIDE with callsite context is lower than the overhead of SHERLOC because InsectACIDE employs selective tracing while SHERLOC does not. InsectACIDE exhibits a similar average overhead compared to the software-based approach with callsite context. However, when more context, such as path-sensitivity, is employed to provide stronger protection, InsectACIDE outperforms the software-based approaches with only a 149% average overhead, while the software-based approach incurs a 364% average overhead. The difference arises from the requirement of holistic protection for the entire non-secure state program. To achieve this, the trace data must be placed in the TEE to prevent attackers from corrupting it. These operations involve context-switching between the normal and secure mode, thereby introducing significant overhead. In contrast, InsectACIDE automatically uses hardware to trace within the TEE, resulting in fewer context-switches. In summary, InsectACIDE offers runtime-efficiency benefits while implementing a stronger form of context-sensitive security checks.

Memory consumption. Considering that some trace data is temporarily stored in device memory, although they are removed later after being checked, an extreme case can occur when the temporarily stored trace data exceeds the device memory limit. In such a case, InsectACIDE has to pause to check. However, our measurement of the memory consumption of InsectACIDE in the worst-case scenario – where there is no trace checking and the trace data are not removed – shows that the benchmark tasks can generate 35KB of trace data on average, while our board has 4MB of memory. After checking the control-flow data, the associated memory can be freed, and therefore, this memory limitation should not be an issue in practice.

Component Predictability. Given that InsectACIDE does not add any inline instructions, the introduced overhead is related to trace processing and interrupt handling, with different components exhibiting varying levels of predictability. Since the defense is deployed in real-time systems, it is essential that the predictability is also well understood.

Among all the components, trace filtering and the handling of `DebugMon` and idle-task preemption interrupts exhibit remarkably predictable execution times, where the difference between worst-case execution time and best-case execution time is very small. The reason for the trace filtering component is that its operations mainly involve referencing instruction opcodes and tables. For the latter component, it is due to relatively fixed workloads, which include performing context switching and modifying corresponding hardware registers. However, the components of trace decoding, control event validation, and selective tracing configuration can exhibit much larger variations in execution time, where the worst-case execution time can be up to 870% of the best-case execution. This is due to the variability of trace events in different program executions, and the need to reallocate the limited number of hardware breakpoints, which involves referring to a pre-defined reallocation policy.

B. Schedulability Analysis

In order to evaluate the real-time impacts of InsectACIDE in comparison to other CFI approaches, we conducted a schedulability evaluation. This evaluation includes overheads and parameter values inspired by our previous empirical results. The goal of this evaluation is to demonstrate how these overheads affect real-time schedulability, especially in comparison to other CFI approaches.

Modeling. Before we detail the specific parameters of our schedulability evaluation, we first must describe our task model and how we include the effects of the InsectACIDE overheads. We assume a standard sporadic task model in which there are n tasks, $\{\tau_1, \ldots, \tau_n\}$ that release an infinite sequence of jobs with a minimum inter-arrival time of T_i and a WCET of C_i. We assume implicit deadlines, so the deadline of each job is T_i time units after its release. We assume a fixed-priority scheduler as this is what FreeRTOS supports, and we assume tasks are assigned rate-monotonic priorities.

We then must incorporate the temporal effects of InsectACIDE, including flushing the buffer and applying the

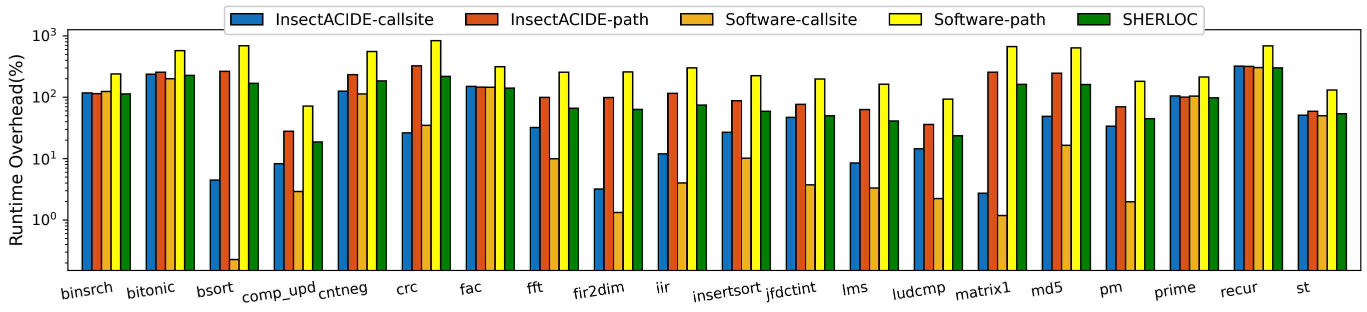

Fig. 6: Comparison with alternative approaches.

checks, in our response-time analysis. Recall the standard response-time analysis for fixed-priority tasks:

$$R_i = C_i + \sum_{\tau_j \in HP(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{1}$$

We now demonstrate how to incorporate the overheads of InsectACIDE. We first review the relevant overhead sources. Recall from Sec. IV that a `DebugMon` exception must be handled to flush the buffer when the MTB is filled. This requires iterating through the entire buffer. This must be done every time that the MTB is filled, regardless of which task is running, as otherwise control-flow events will not be recorded for checking. We make the conservative assumption that every task has a control output, or another output that must be checked before the deadline of the task. Therefore, there is a corresponding check that must also be completed before the deadline, so we model this as a per-event overhead, which we define as δ.

We have also measured the rate at which control-flow events occur, and thus can determine how often the MTB is filled. We augment the sporadic task model to include the maximum number of control-flow events, E_i, that can occur during a job of τ_i. In our schedulability study we assume that there is a control-event frequency F_i, and the number of control events in task τ_i is determined as $E_i = F_i C_i$. We assume the MTB is of size S.

Now we derive how to incorporate these modeled overheads into our response-time analysis. By definition, each job τ_j that runs in the scheduling window of τ_i introduces $E_j = C_j F_j$ control events, each of which introduces δ additional demand. We also observe that we cannot guarantee that the buffer is empty at the critical instant, and must therefore assume that the buffer is full and will trigger an interrupt immediately after the critical instant, forcing a check of the full buffer size, assumed to be S. Thus, our resulting response-time analysis including overheads is adapted from Eq. 1 as

$$R_i = S\delta + C_i + E_i\delta + \sum_{\tau_j \in HP(\tau_i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil (C_j + E_j\delta) \right). \tag{2}$$

Note that this assumes that the control outputs occur at the end of τ_i, and thus all control-flow events in the task must be checked before the control output. Therefore, at the end of the task, a MPU/DWT fault will trigger InsectACIDE to check any buffered control-flow events to be processed, even if the buffer is not full. (We note that the previous work of SHERLOC [8] behaves similarly to InsectACIDE except that it does not apply any filtering and does not enforce checks that are as strong as ours; hence it, too, can be modeled as Eq. 2 by using different values for δ and F.)

In practice, not all tasks may have control outputs, but there may be dependencies between non-control-output tasks as the output of one task may be the input of the next. In such cases it may be possible to defer checks of non-control tasks until after their deadline, potentially further improving schedulability analysis. We leave analyzing this type of behavior to future work. We have also only considered a single core in our schedulability evaluation as our target platform only had one core. We are not aware of any similar microcontroller that both has multiple cores and support for the MTB and DWT. However, having a separate core to process checks in parallel could offer further benefits to asynchronous CFI.

Experimental design. To evaluate the effects of these overhead sources on real-time schedulability, we conducted a schedulability study based on randomly generated task systems. We used SchedCAT [40] to analyze the response times of tasks using rate-monotonic scheduling on a single-core system, which is representative of our hardware platform. Our experimental design applied commonly used task-system distributions defined by Brandenburg [41] and also encoded in SchedCAT. We include a representative subset of these graphs in Fig. 7.

While we tested all of the standard distributions described in SchedCAT and [41], here we review only the distributions used in the graphs in Fig. 7. We used short, medium, and long periods uniformly distributed in $[3, 33]$ ms, $[10, 100]$ ms, and $[50, 250]$ ms, respectively. We also considered utilizations that were light, medium, and heavy, uniformly distributed in [0.001, 0.1], [0.1, 0.4], and [0.5, 0.9], respectively. For each combination of periods and utilizations, we generated task systems with total utilization in $[0.1, 0.2, \ldots, 1]$. We randomly generated F_i from a uniform distribution with a maximum of 100 cycles/event and minimum of 5 cycles/event taken from what we observed from our implementation. We set the hardware clock speed to 20 MHz.

We compared the results from Eq. 1, which represents a system without InsectACIDE to the results from Eq. 2, which represents InsectACIDE. Within InsectACIDE, we compared the schedulability of tasks with filtering to the schedulability of tasks without filtering. We also considered the schedulability of using software tracing to the other hardware-based options. Finally, we compared InsectACIDE to SHERLOC [8], which we see as the most similar related work.

As previously discussed, task sets may generate control-flow events at different rates. Thus, we define a parameter F that is set per-task from a uniform distribution in $[5, 100]$ based on the range we observed in our experimental findings. We use this F for the no-filtering tests and the SHERLOC tests.

For the filtering tests, we define an "improvement factor" between F_{nofilter} and F_{filter}. For each task, we pick an improvement factor from a uniform distribution in [1, 596] based on the range we observed in our experimental findings. We then divide each task's F_{nofilter} by its improvement factor to arrive at its F_{filter}.

We also compare against a software-only instrumentation (with callsite as context), as described in Sec. VI-A. To approximate the schedulability of this approach we simply inflate the execution time by the overhead factor of 0.57 that we observed in our experimental measurements, since there is no previous schedulability analysis.

For the rest of the non-normal approaches, we noted through our experiments that δ differs between the different cases. Notably, the cost for flushing the MTB differs for the filtering approach to account for the added overhead of the selective tracing mechanism, and the cost of checking each entry differs for SHERLOC. Thus, we define $\delta = 2.238\,\mu s$ for the no-filter case, $\delta = 3.543\,\mu s$ for the filtering case, and $\delta = 2.032\,\mu s$ for SHERLOC.

Results. Fig. 7 shows three representative graphs from our study, from which we make several observations.

For task systems with long periods and heavy utilizations, the utilization loss to overheads in InsectACIDE is comparatively smaller. In our response-time analysis (Eq. 2) there is carry-in work at the critical instant that exists from events that are buffered but not yet processed. This has a comparatively smaller effect when periods are longer, as shown in Fig. 7(a). Conversely, when periods are much shorter, this overhead is quite costly, as can be seen in Fig. 7(c). We note that this behavior is observed for both InsectACIDE and the prior state-of-the-art, SHERLOC.

Next we consider the effects of selective filtering. In the implementation of selective filtering there are greater overheads, and more of the recorded control-flow events must be checked (vs. discarded if they are not security relevant – e.g., a for loop instead of a function return). Therefore, the per-event processing time is greater, while the number of events is reduced. Interestingly, in many of the schedulability graphs these effects largely counteracted one another and filtering usually had only a modest benefit, at best.

We also observe that InsectACIDE and SHERLOC, the hardware-based approaches, do not dominate the software-based approach, or vice versa. However, we note that both InsectACIDE and SHERLOC have advantages in that they do not require any binary instrumentation, whereas software-based approaches do.

Finally, we note that InsectACIDE applies context-sensitive CFI whereas SHERLOC does not. Therefore, InsectACIDE is a stronger defense and in many cases the analytical utilization loss due to additional trace processing cost is quite modest, with the possibility of even improved performance using our selective-filtering technique.

TABLE I: Quantitative security evaluation

Task	**SHERLOC**		**InsectACIDE**		**Difference**	
	Avg	**Lg**	**Avg**	**Lg**	**Avg**	**Lg**
failsafe check	2.8	4	1.2	2	↓57%	↓50%
collision prevent	2	2	1	1	↓50%	↓50%
flight mode manage	8.6	12	6.2	12	↓28%	0
geofence breach avoid	1.8	5	1.3	3	↓27%	↓40%
vtol attitude control	2.3	3	2.2	3	↓4%	0

Avg/Lg: average/largest equivalence class size.

C. Security Evaluation

To demonstrate the security benefits of InsectACIDE, both qualitative and quantitative evaluations are conducted to compare their ability to detect control-flow hijacking attacks. Five real-time tasks from PX4 [42] (as listed in Table I) are used in this case study.

Qualitative demonstration of added protection from the use of application context. To demonstrate the attack, a buffer-overflow vulnerability is manually injected by replacing a safe length-checked `strncpy` function in the `Mavlink` module with an unsafe `strcpy` function without length checking, as shown in Listing 1. The manual injection of vulnerability is a common security evaluation technique for defensive systems, and was used in [5], [43]. This enables attackers to perform out-of-bound writes.

We implemented an attack to exploit the over-approximation in the points-to analysis, which is conducted on an indirect function call `_current_task.task->activate(last_setpoint)` (Line 7). The indirect function call uses a virtual function pointer of the object in the C++ class `FlightTask` to call its member function `activate`, which is responsible for configuring flight parameters of the current flight mode. Due to the C++ class inheritance feature, there are twelve possible targets at the indirect call, as shown in the SHERLOC-EC box. One of the flight modes is `Failsafe`, a critical functionality to guarantee safety in unexpected situations, and its invocation pattern is shown from Line 4 to 9. In this case, the attacker's goal is to divert the control-flow of the `Failsafe` process to another one of twelve targets, such as `FlightTaskOrbit::activate`,

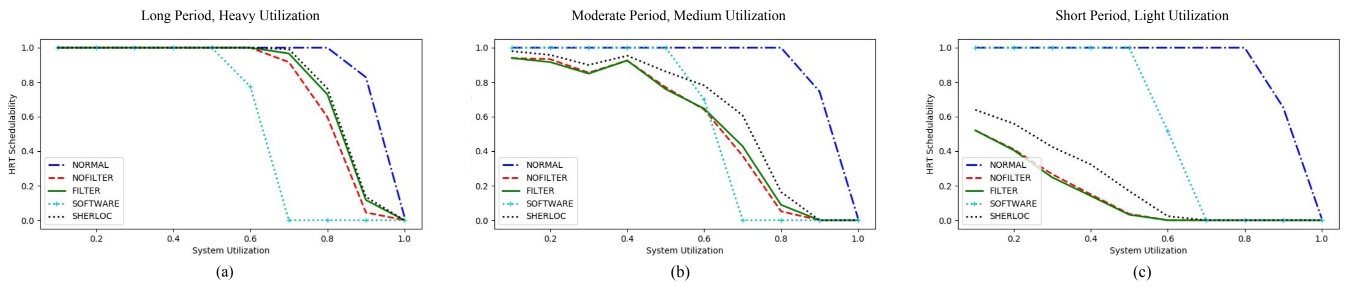

Fig. 7: Results from schedulability study – see Sec. VI-B.

```
01 bool should_publish_current(mavlink_statustext_t &msg_statustext){ //malicious input
-02   strncpy(_log_msg.text+offset,msg_statustext.text, max_to_add);
+03   strcpy(_log_msg.text+offset,msg_statustext.text);
   }
04 FlightTaskError switchTask(FlightTaskIndex new_task_index){
05   switch (new_task_index) { case FlightTaskIndex::Failsafe:
06       _current_task.task = flightTaskFailsafe; }
07       _current_task.task->activate(last_setpoint); /* indirect call to protect */
     }
08 void FlightModeManager::tryApplyCommandIfAny() {
09   switchTask(FlightTaskIndex::Failsafe);/* callsite1, context to leverage */
   }
10 void FlightModeManager::start_flight_task() {
11   error = switchTask(FlightTaskIndex::Orbit);/* callsite2 context */ }
SHERLOC-EC  FlightTaskDescend::activate  FlightTaskOrbit::activate  ***::activate
  FlightTaskAuto::activate  InsectACIDE-EC  FlightTaskFailsafe::activate
```

Listing 1: Example of attack detection.

to subvert the safety of the system. This attack cannot be detected by SHERLOC, but InsectACIDE can utilize the context at Line 9 and check the control-flow transition events recorded at both Line 7 and 9 together to narrow its EC, ultimately detecting the attack.

Quantitative analysis. We use EC size as a security metric to quantify the added security protection. While this is by no means a complete indication of security, EC size effectively quantifies the benefits of leveraging the context, since the fundamental vulnerability behind naive CFI is the ambiguity of pointer values within the EC. Intuitively, the smaller the size of the EC, the less room there is for an attacker to adversarially manipulate by corrupting the pointer value with other values in the same EC. The largest and average EC size of indirect branches for each task are measured. To facilitate a comparison between SHERLOC and InsectACIDE, the difference in the measured metric, calculated as $|\frac{\text{InsectACIDE}-\text{SHERLOC}}{\text{SHERLOC}}|$, is shown in Table I. InsectACIDE can effectively reduce the EC size by up to 57% using context.

However, for the task *vtol_attitude_control*, where InsectACIDE performs least effectively, we found that this is due to the ineffectiveness of using callsite as context, because the target information of most code pointers cannot be propagated along the function callsite. This also shows that context-sensitive CFI depends heavily on the selection of context type.

VII. Security Analysis

Security of system implementation. An attacker may attempt to corrupt metadata to change malicious trace data into benign ones in order to bypass detection. However, InsectACIDE configures the MTB to be accessible only in TrustZone. Any attempts to access the trace data from normal mode or normal world DMA controller will trigger a fault.

The InsectACIDE implementation disables the MTB in three instances. If an attacker is able to forcibly disable the MTB they could bypass security checks, so we assess each of these cases individually. First, when the MTB becomes full, a DebugMon exception is raised and the buffer is flushed in the secure mode. During this flushing, the MTB is disabled. However, before disabling the MTB, InsectACIDE also disables interrupts, leaving no opportunity for the attacker to forcibly preempt the flushing and hijack control flow without detection. The second case is when trace data is processed during idle time. Although interrupts are enabled in this case, the interrupts are detectable in secure world, and MTB is re-enabled before context switches back to normal world. The third case is when performing selective tracing. In this case, the MPU is configured to trap any code outside of the selected untraced code, ensuring that the MTB is re-enabled at the exit of selective-trace segment.

Security of control-flow protection. An attacker may leverage the asynchrony between trace recording and trace checking to launch an attack, such as skipping the idle task to delay trace checking, to cause critical consequences before being detected. However, InsectACIDE ensures that all CF events are checked prior to interaction with the physical world. Moreover, attackers may attempt to leverage the selective tracing mechanism, in which the MTB is disabled, to initiate malicious control transitions. However, the code segments that are filtered are carefully selected to not contain any indirect transfers. As a result, attacks cannot hijack control solely within the selectively untraced code.

VIII. Related Work

Control-flow protection on embedded system. The comparison with related work is shown in Table II. To achieve *binary-preserving* properties, existing works utilize either off-the-shelf hardware [8], [16], [46], [47] or customized hard-

TABLE II: Control-flow Protection on Embedded System

System	Arch.	No Inst.	Non Priv.	Priv.	Ret. Addr.	Cxt	RT Ada.
RECFISH [27]	R		✓		✓		✓
μRAI [5]	M		✓	✓	✓✗		
Silhouette [26]	M			✓	✓		
Kage [25]	M			✓	✓		✓
CaRE [21]	M		✓	✓	✓		
TZmCFI [20]	M		✓	✓	✓		
SHERLOC [8]	M	✓	✓	✓	✓		
ECFI [44]	A		✓		✓		✓
FastCFI [45]	Cus.	✓	✓		✓	✓	✓
Kadar et al. [46]	A	✓	✓✗		✓✗	✓✗	✓
InsectACIDE	M	✓	✓	✓	✓	✓	✓

Arch: architecture, Cus: customized hardware, Inst: instrumentation requirement, Priv: privileged mode protection, Ret Addr: return address protection, Ctx: context-sensitive forward-edge checking, RT Ada: real-time adaption, ✓✗: partially support.

ware [45], [48]. This is because hardware support is necessary for monitoring a program whose binary is unmodified. InsectACIDE can be deployed on commodity hardware. Existing approaches providing the *holistic protection* [5], [8], [20], [21] often rely on modification of the system binary. However, InsectACIDE transparently ensures the context-sensitive CFI. Another line of work focuses on the timeliness of program execution [47], [48] rather than control-flow integrity, or only checks partial control flow events [46] to accommodate real-time requirements. Therefore, they do not provide holistic protection either. Additionally, most of the works, including SHERLOC [8], do not perform *context-sensitive* checks for fine-grained control-flow protection, or offer *real-time adaptation* to optimize performance or provide analysis for real-time guarantees, with the exception of [25], [27], [44]–[46]. Furthermore, SHERLOC does not address unique challenges stemming from real-time operating systems, such as the idle task in FreeRTOS. In comparison to these works, InsectACIDE is the first to support all the above-mentioned properties using existing hardware on Arm Cortex-M.

Hardware-assisted tracing. Existing works leverage various tracing hardware for security protection. On server platforms such as x86, BTS (Branch Trace Store), LBR (Last Branch Record) tracing, and Intel Processor Trace (PT) are used for context/path-sensitive control-flow integrity [6], [7], [11]–[16]. On the Arm Cortex-A and M platforms, both CoreSight Embedded Trace Macrocell (ETM) and MTB have been utilized to monitor control flows [8], [46]. InsectACIDE also leverages the hardware to provide traces to provide security protection.

IX. Discussion and Limitations

Finer-grained security scheduling policy. The existing implementation of InsectACIDE conducts security checking during idle time, and the scheduling of trace processing workloads for different tasks is considered a unified security task, requiring that all trace validation must be completed before any physical outputs can occur. One might argue that not all tasks' traces have to be validated; instead, only trace data of tasks that generate output needs validation. If this were the case, we could schedule the trace processing workloads of different tasks separately to gain real-time benefits. However, an attack on one task can potentially affect the entire system, as all tasks and kernels share a single address space. One method to mitigate this is to use intra-space isolation. It is important to note that a hijacked process may never schedule the idle task, in this case, such system anomalies can be detected by observing unexpected number of traces for processing.

System requirements. InsectACIDE makes use several hardware features, including the debugging (e.g., MTB), the trusted execution environment (TrustZone), memory access control (MPU), and therefore may not be directly applicable to low power platforms that do not provide these hardware features. Even though it is possible to leverage software realization (such as software sandboxing) to realize the hardware features, there will likely be additional performance overhead.

From the performance perspective, the added security protection from context sensitivity can incur non-trivial performance overhead (both computational overhead and memory overhead). This might also change the system size, weight and power requirements. On the bright side, it might be possible to leverage the idle core in modern multi-core processors to conduct security enforcement without the need to change system hardware.

X. Conclusion

In this paper we have presented InsectACIDE, the first binary-preserving, asynchronous, context-sensitive, and holistic CFI for embedded and real-time systems. InsectACIDE provides control-flow protection for both userspace and kernel processing using Arm TrustZone, and uses hardware debugging features to separate the recording of control-flow events from the checking for correct control flow. We have implemented InsectACIDE on an Arm Cortex-M processor and have presented empirical evaluations of the overheads of the approach. Our experimental results show that InsectACIDE incurs less runtime overhead compared to the state-of-the-art holistic CFI solution, and our real-time schedulability analysis and evaluations demonstrate the tradeoff between improved protection with InsectACIDE, and schedulability.

Acknowledgment

We thank the reviewers for their valuable feedback. This work was partially supported by the NSF (CNS-1916926, CNS-2038995, CNS-2154930, CNS-2229427, CNS-2141256, NSF 2237238, CNS-2238635, CPS-2229290), and ARO (W911NF2010141), Washington University in St. Louis, and Vanderbilt University.

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE Symposium on Security and Privacy*, 2013.

[2] G. Thomas, "A proactive approach to more secure code," 2019.

[3] C. Project, "Memory safety," 2020.

[4] H. Shacham, "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*, 2007.

[5] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing embedded systems with return address integrity," in *Network and Distributed Systems Security Symposium*, 2020.

[6] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *26th USENIX Security Symposium*, 2017.

[7] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[8] X. Tan and Z. Zhao, "SHERLOC: Secure and holistic control-flow violation detection on embedded systems," in *ACM Conference on Computer and Communications Security*, 2023.

[9] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *USENIX Security*, 2015.

[10] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *28th USENIX Security Symposium*, 2019.

[11] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012.

[12] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[13] P. Yuan, Q. Zeng, and X. Ding, "Hardware-assisted fine-grained code-reuse attack detection," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2015.

[14] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: Guarding control flows using Intel processor trace," *ACM SIGPLAN Notices*, 2017.

[15] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent backward-edge control flow violation detection using Intel processor trace," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017.

[16] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient CFI enforcement with Intel processor trace," in *IEEE International Symposium on High performance computer architecture*, 2017.

[17] S. Baruah, P. Ekberg, M. Hosseinzadeh, A. Li, B. Ward, and N. Zhang, "Who's afraid of butterflies? a close examination of the butterfly attack," in *2023 IEEE Real-Time Systems Symposium (RTSS)*, pp. 53–63, IEEE, 2023.

[18] T. Mishra, T. Chantem, and R. Gerdes, "Survey of control-flow integrity techniques for embedded and real-time embedded systems," *ACM Transactions on Embedded Computing Systems*, 2021.

[19] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[20] T. Kawada, S. Honda, Y. Matsubara, and H. Takada, "TZmCFI: RTOS-aware control-flow integrity using trustzone for Armv8-M," *International Journal of Parallel Programming*, 2021.

[21] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2017.

[22] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang, "RT-TEE: Real-time system availability for cyber-physical systems using Arm TrustZone," in *IEEE Symposium on Security and Privacy*, 2022.

[23] J. Wang, Y. Wang, and N. Zhang, "Secure and timely GPU execution in cyber-physical systems," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[24] J. Wang, Y. Wang, A. Li, Y. Xiao, R. Zhang, W. Lou, Y. T. Hou, and N. Zhang, "ARI: Attestation of real-time mission execution integrity," in *32nd USENIX Security Symposium*, 2023.

[25] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell, "Holistic control-flow protection on real-time embedded systems with Kage," *USENIX Security Symposium*, 2022.

[26] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *USENIX Security*, 2020.

[27] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-flow integrity for real-time embedded systems," in *31st Euromicro Conference on Real-Time Systems*, 2019.

[28] Y. Wang, A. Li, J. Wang, S. Baruah, and N. Zhang, "Opportunistic data flow integrity for real-time cyber-physical systems using worst case execution time reservation," in *USENIX Security Symposium*, 2024.

[29] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[30] M. A. Elmohr, H. Liao, and C. H. Gebotys, "EM fault injection on ARM and RISC-V," in *IEEE International Symposium on Quality Electronic Design*, 2020.

[31] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Truspy: Cache side-channel information leakage from the secure world on arm devices," *Cryptology ePrint Archive*, 2016.

[32] A. Li, M. Sudvarg, H. Liu, Z. Yu, C. Gill, and N. Zhang, "Polyrhythm: Adaptive tuning of a multi-channel attack template for timing interference," in *IEEE Real-Time Systems Symposium*, 2022.

[33] A. Li, J. Wang, S. Baruah, B. Sinopoli, and N. Zhang, "An empirical study of performance interference: Timing violation patterns and impacts," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2024.

[34] H. Liu, Y. Wu, Z. Yu, Y. Vorobeychik, and N. Zhang, "Slowlidar: Increasing the latency of lidar-based detection using adversarial examples," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023.

[35] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[36] C. Ferdinand and R. Heckmann, "aiT: Worst-case execution time prediction by static program analysis," in *Building the Information Society*, Springer, 2004.

[37] "Arm MPS2+ FPGA prototyping board." https://www.arm.com/products/development-tools/development-boards/mps2-plus, 2017.

[38] "AN505: Cortex™-M33 with IoT kit FPGA for MPS2+ Version 2.0." https://developer.arm.com/downloads/view/AN505, 2017.

[39] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

[40] B. Brandenburg, "SchedCAT." In B. Brandenburg and M. Gül, "Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations," Proceedings of the 37th IEEE Real-Time Systems Symposium, 2016.

[41] B. B. Brandenburg, *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[42] L. Meier, D. Honegger, and M. Pollefeys, "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *IEEE International Conference on Robotics and Automation*, 2015.

[43] Z. Sun, B. Feng, L. Lu, and S. Jha, "OAT: Attesting operation integrity of embedded devices," in *IEEE Symposium on Security and Privacy*, 2020.

[44] A. Abbasi, T. Holz, E. Zambon, and S. Etalle, "ECFI: Asynchronous control flow integrity for programmable logic controllers," in *Proceedings of Annual Computer Security Applications Conference*, 2017.

[45] L. Feng, J. Huang, J. Hu, and A. Reddy, "FastCFI: Real-time control-flow integrity using FPGA without code instrumentation," *ACM Transactions on Design Automation of Electronic Systems*, 2021.

[46] M. Kadar, G. Fohler, D. Kuzhiyelil, and P. Gorski, "Safety-aware integration of hardware-assisted program tracing in mixed-criticality systems for security monitoring," in *IEEE 27th Real-Time and Embedded Technology and Applications Symposium*, 2021.

[47] W. Chen, I. Izhibirdeev, D. Hoornaert, S. Roozkhosh, P. Carpanedo, S. Sharma, and R. Mancuso, "Low-overhead online assessment of timely progress as a system commodity," in *35th Euromicro Conference on Real-Time Systems*, 2023.

[48] D. Lo, M. Ismail, T. Chen, and G. E. Suh, "Slack-aware opportunistic monitoring for real-time systems," in *IEEE 19th Real-Time and Embedded Technology and Applications Symposium*, 2014.