2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS) | 979-8-3503-2176-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/RTAS58335.2023.00018

ISC-FLAT: On the Conflict Between Control Flow Attestation and Real-Time Operations

Antonio Joia Neto
Rochester Institute of Technology
aj4775@rit.edu

Ivan De Oliveira Nunes
Rochester Institute of Technology
ivanoliv@mail.rit.edu

***Abstract*—The wide adoption of IoT gadgets and Cyber-Physical Systems (CPS) makes embedded devices increasingly important. While some of these devices perform mission-critical tasks, they are usually implemented using Micro-Controller Units (MCUs) that lack security mechanisms on par with those available to general-purpose computers, making them more susceptible to remote exploits that could corrupt their software integrity. Motivated by this problem, prior work has proposed techniques to remotely assess the trustworthiness of embedded MCU software. Among them, Control Flow Attestation (CFA) enables remote detection of runtime abuses that illegally modify the program's control flow during execution (e.g., control flow hijacking and code reuse attacks).**

Despite these advances, current CFA methods share a fundamental limitation: they preclude *interrupts* during the execution of the software operation being attested. Simply put, existing CFA techniques are insecure unless interrupts are disabled on the MCU. On the other hand, we argue that the lack of interruptability can obscure CFA usefulness, as most embedded applications depend on interrupts to process asynchronous events in real-time.

To address this limitation, we propose Interrupt-Safe Control Flow Attestation (ISC-FLAT): a CFA technique that is compatible with existing MCUs (i.e., does not require hardware changes) and enables interrupt handling without compromising the authenticity of CFA reports. Similar to other CFA techniques that do not require customized hardware modifications, ISC-FLAT leverages a Trusted Execution Environment (TEE) (in particular, our prototype is built on ARM TrustZone-M) to securely generate unforgeable CFA reports without precluding applications from processing interrupts. We implement a fully functional ISC-FLAT prototype on the ARM Cortex-M33 MCU and demonstrate that it incurs minimal runtime overhead when compared to existing TEE-based CFA methods that do not support interrupts.

I. Introduction

From IoT gadgets to vehicular safety-critical sensors, society has grown accustomed to the pervasiveness of embedded devices. Naturally, this increased reliance is accompanied by a growing risk of embedded software compromise. Unfortunately, prevention of software compromises in embedded devices is especially challenging because they are typically implemented using (one or several) Micro-Controller Units (MCUs). Due to strict cost and energy budgets, MCUs lack security mechanisms commonly found in higher-end general-purpose CPUs, such as strong separation of privilege levels and memory management units to support virtual memory and isolation. Furthermore, embedded devices are often deployed in multitudes, sharing the same (potentially vulnerable) software. Unsurprisingly, the insecurity of embedded software has already resulted in several attacks, including massive Distributed Denial of Service (DDoS) [15], [29] and large-scale exploits with economical and life-threatening consequences [56], [59], [41], [34].

Since preventive approaches are often too costly or unfeasible in resource-constrained embedded devices, security services that enable remote detection of software compromise have attracted attention in recent years [39]. Remote Attestation ($\mathcal{RA}$) [61], [51], [42], [32], [46], [14], [38], [17], [33], [18], [22], [23], [37], [21], [35], [52] is one such service that enables a Verifier ($\mathcal{V}$rf) to assess the trustworthiness of the software executing on a remote low-end embedded device – called a Prover ($\mathcal{P}$rv). In its simplest form (a.k.a. "static $\mathcal{RA}$" or "$\mathcal{RA}$ of binaries"), $\mathcal{RA}$ offers means to detect illegal modifications to the binary installed and running on $\mathcal{P}$rv. However, by itself, it provides no information about the order in which instructions within the binary execute at runtime. In particular, control flow hijacks [10] and code reuse attacks [63], [65] (e.g., via return-oriented programming [60], [24]) can change the order in which instructions execute without modifying the binary. As the binary remains the same, such attacks cannot be detected by static $\mathcal{RA}$.

To address this limitation, Control Flow Attestation (CFA) [11], [31], [30], [74], [67], [28], [69], [75] augments static $\mathcal{RA}$ to provide $\mathcal{V}$rf with an unforgeable "control flow proof", containing the order in which the instructions of the attested binary have executed. As such, CFA enables the detection of control flow hijacks and code reuse attacks, even when these attacks do not modify the installed binary.

CFA defines a Control Flow Graph (CFG). Nodes in the CFG are sequences of non-branching instructions. Edges represent control flow transfers (e.g., jumps, calls, returns, etc.). CFA techniques work by tracking the path taken in the program's CFG during execution. This model assumes that all instructions within the same CFG node are guaranteed to execute sequentially. However, when interrupts are enabled, this assumption is falsifiable because interrupts can cause control flow transfers asynchronously within any given node in the CFG. As a result, existing CFA techniques either assume [11], [67] or enforce [28], [48] disablement of all interrupts in the MCU. On the other hand, most real-time applications are interrupt-based [70]. This conflict can make current CFA methods impractical in many settings.

979-8-3503-2176-0/23/$31.00 ©2023 IEEE
DOI 10.1109/RTAS58335.2023.00018

Contributions. In this work, we aim to reconcile CFA with the real-time needs of MCU application domains. To that end, we propose ISC-FLAT: an Interrupt-Safe Control Flow Attestation method. ISC-FLAT leverages ARM TrustZone-M to isolate side-effects of external interrupts from the execution being attested. Since TrustZone-M is available in several ARM CPUs, ISC-FLAT is readily deployable on "off-the-shelf" MCUs without requiring additional customized hardware. To the best of our knowledge, ISC-FLAT is the first TEE-based CFA approach to securely support interrupts while maintaining CFA integrity. In sum, the anticipated contributions of this paper are three-fold:

- We formulate and characterize the conflict between real-time applications and existing TEE-based CFA methods. To motivate the need for interrupt-safe CFA, we demonstrate interrupt-based attacks on current CFA designs.
- We propose ISC-FLAT, a TEE-based CFA design that supports interrupts without compromising the integrity of underlying CFA proofs. At its core, ISC-FLAT implements a TrustZone-based secure interrupt dispatcher that leverages TrustZone's Nested Vectored Interrupt Controller (NVIC) to interpose itself between any interrupt trigger and its respective service routine. The dispatcher saves the necessary context of the interrupted task within TrustZone's Secure World and verifies that this context is resumed appropriately when the service routine ends. This design ensures CFA integrity for the interrupted task without making service routines part of ISC-FLAT Trusted Computing Base (TCB).
- We implement and evaluate ISC-FLAT on an "off-the-shelf" ARM Cortex-M33 MCU, equipped with TrustZone-M. Our experimental results demonstrate ISC-FLAT *quasi*-negligible overhead when compared to an existing CFA technique that does not support secure interrupts. To foster future research in this topic, we make ISC-FLAT implementation publicly available at [3].

II. Background & Related Work

A. ARM TrustZone-M

ARM TrustZone is a Trusted Execution Environment (TEE) included in modern ARM CPUs. It partitions the System on Chip (SoC) hardware and software into Secure and Non-secure regions (called "Worlds"). Resources belonging to the Secure World are isolated from the Normal (Non-Secure) World, resulting in a secure environment for executing security-critical functions and storing sensitive data. TrustZone's hardware controllers prevent the Normal World from accessing physical memory regions assigned to the Secure World.

TrustZone-M defines the security state of memory segments (i.e., whether a segment in the address space belongs to the Secure or Normal World) by using a combination of the Secure Attribution Unit (SAU), and the Implementation Defined Attribution Unit (IDAU) to enforce spatial isolation. IDAU is a fixed memory map defined by the manufacturer, while SAU is programmable by the Secure World.

A number of prior efforts aim to leverage TrustZone-M to enhance embedded system security from various perspectives, including low latency secure interrupts [50], real-time system availability guarantees [71], Address-Space Layout Randomization (ALSR) without requiring memory management units [44], and support for virtualization [55]. For a comprehensive overview of TrustZone's architecture, we refer the reader to [54].

B. Interrupts & TrustZone-M NVIC

TrustZone-M capable MCUs process all interrupts using two separated Interrupt Vector Tables (IVTs) for the Secure and Normal Worlds. They are managed by an integrated controller called Nested Vector Interrupt Controller (NVIC). Each interrupt can be assigned as Secure or Non-secure by setting a register named Interrupt Target Non-secure (NVIC_ITNS), which is only programmable in the Secure World. In addition, the IVTs can share the same priority level, or secure interrupts can have priority over non-secure ones. The interrupt pipeline follows the standard execution flow if an interrupt is triggered while the CPU is in the same security state as the interrupt. If a Secure interrupt is triggered while the CPU is in the non-secure state, the CPU ignores the Non-Secure IVT and redirects execution to the Interrupt Service Routine (ISR) address stored in the Secure IVT, while automatically pushing the registers of the interrupted task to the non-secure stack. Recent related work [50], [49] takes advantage of the NVIC controllers. Specifically, the NVIC_ITNS register is used to enforce specific interrupt states as a requirement for the additional security features.

C. Static Remote Attestation ($\mathcal{R}$A)

Static $\mathcal{R}$A (a.k.a. "$\mathcal{R}$A of binaries") allows a verifier ($\mathcal{V}$rf) to determine the integrity of an application's binary running on an untrusted remote platform ($\mathcal{P}$rv), i.e., $\mathcal{V}$rf is able to detect illegal modifications to the binary. $\mathcal{R}$A is also a building block for other services, including Control Flow Attestation (CFA), Data Flow Attestation (DFA), and Proof of Execution (PoX) [47]. For instance, since TEE-based CFA requires binary instrumentation (see details in Section II-D), $\mathcal{R}$A is necessary as a part of CFA to guarantee that the instrumentation has not been maliciously removed.

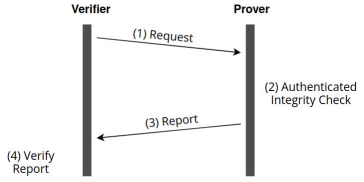


Fig. 1. $\mathcal{R}$A interaction

As depicted in Figure 1, $\mathcal{R}$A is a challenge-response protocol wherein $\mathcal{V}$rf sends an attestation request to $\mathcal{P}$rv containing a cryptographic challenge (Chl) (e.g., a random nonce) to guarantee that the $\mathcal{P}$rv will generate a unique, timely response to this request. $\mathcal{P}$rv receives Chl and computes an authenticated integrity check (e.g., a MAC or a signature) over its own program memory and Chl, and sends the resulting

report back to $\mathcal{V}$rf. Finally, $\mathcal{V}$rf checks the received report against the expected value (i.e., the expected binary for $\mathcal{P}$rv).

Static $\mathcal{R}$A architectures are typically classified as software-based, hardware-based, or hybrid. Software-based $\mathcal{R}$A [62], [26], [66] does not require hardware support. Instead, it leverages precise timing, along with strong assumptions about direct communication and adversarial silence, to detect malware on $\mathcal{P}$rv. To relax these assumptions, hardware-based attestation relies on a hardware root of trust for measurement and reporting to produce unforgeable attestation reports securely. This category includes TPM-based approaches [73], Intel SGX [72], [40], and ARM TrustZone [12]. Hardware-based designs provide a high level of security. However, they are restricted to devices that support these custom features. Between hardware and software-based, hybrid $\mathcal{R}$A [51], [38], [32], [17], [46], [27] requires minimal hardware trust anchors to provide strong security guarantees on resource-constrained devices. Besides just attesting the integrity of a single device's binary, collective $\mathcal{R}$A methods focus on groups of devices [20], [45], [13], [16]. Carpent et al. [21], [23] introduce attestation for self-relocating malware.

D. Control Flow Attestation (CFA)

CFA augments $\mathcal{R}$A by including a proof of the sequence in which instructions were executed in the attested binary. Consequently, CFA can detect runtime attacks that hijack the program's control flow but do not modify $\mathcal{P}$rv's code (e.g., the well-known return-oriented programming and code-reuse attacks [68]). CFA is achieved by tracking and logging the control flow path taken during the execution of the attested binary. While CFA does not actively protect the system against control flow attacks, it provides $\mathcal{V}$rf with information to detect any control flow attack during the execution. As such, $\mathcal{V}$rf can accurately decide if a result produced by this execution (e.g., a sensed value) is trustworthy.

In CFA, $\mathcal{V}$rf requests from $\mathcal{P}$rv an authenticated proof that: **(i)** $\mathcal{P}$rv indeed executed the expected binary (denoted $\mathcal{A}$pp) in a timely manner, i.e., after the most recent request from $\mathcal{V}$rf; and **(ii)** there were no control flow attacks during $\mathcal{A}$pp execution. Optionally, the proof may also include any execution results (e.g., a sensed value) produced by $\mathcal{A}$pp execution on $\mathcal{P}$rv, allowing $\mathcal{V}$rf to assess the trustworthiness of this result based on the CFA verification. We refer to this "proof" as the CFA report. **We define a CFA report as *unreliable* if the control flow path taken during $\mathcal{A}$pp execution differs from the one contained in the CFA report.** Therefore, an unreliable report could make an attack execution oblivious to $\mathcal{V}$rf.

TEE-based CFA methods [11], [67], [43] use automated binary instrumentation to build an authenticated control flow log ($\mathcal{CF}_{\mathsf{Log}}$) containing all control flow transfers, i.e., the destination address of all the branching instructions (e.g, `jumps`, `calls`, `returns`) taken during execution. $\mathcal{V}$rf can then check $\mathcal{CF}_{\mathsf{Log}}$ to decide if the control flow transfers were valid. Every node in the attested program's CFG is instrumented with additional trampoline instructions that trap the execution into the Secure World. Within the Secure World, the branch destination is appended to $\mathcal{CF}_{\mathsf{Log}}$. $\mathcal{CF}_{\mathsf{Log}}$ itself is stored in the Secure World protected memory. This approach assumes that only branching instructions can modify the control flow. Therefore, sequences of regular (non-branching) instructions are treated as "blocks". A block followed by a branching instruction defines a node of the CFG (a block can be empty in the case of two sequential branching instructions). When interrupts are disabled, this approach works well because instructions within a block are unable to modify the program's control flow. However, when interrupts are enabled, any number of control flow transfers may occur within the execution of a single block. Even worse: these transfers would not be saved to $\mathcal{CF}_{\mathsf{Log}}$ and therefore not noticed by $\mathcal{V}$rf. In Section III we elaborate on this issue and further motivate its importance.

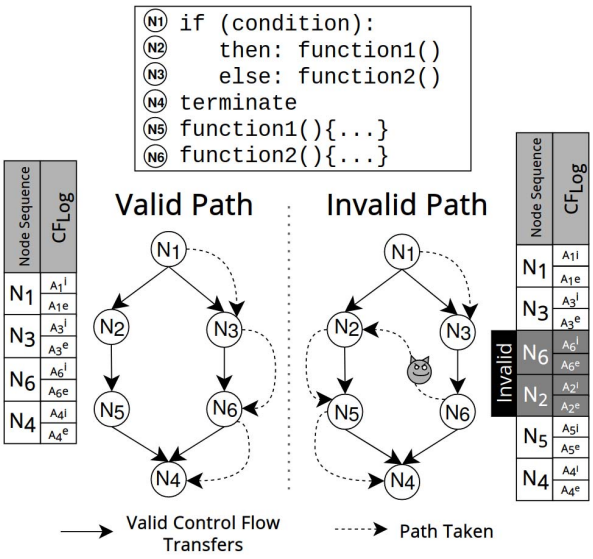


Fig. 2. The figure illustrates an example of a valid (left) and an invalid (right) control flow execution with their respective generated $\mathcal{CF}_{\mathsf{Log}}$.

Figure 2 exemplifies CFA. N_1 - N_6 are nodes in the CFG of the attested application. Solid arrows represent legal control flow transfers, and dashed arrows represent control flow paths taken during a particular program execution. When a node is executed entirely, the additional (instrumented) instructions within the node produce two new entries in $\mathcal{CF}_{\mathsf{Log}}$ (denoted A_x^i and A_x^e, for node N_x). A_x^i represents the memory address of the first instruction of the node N_x, and A_x^e represents the destination of N_x's branching instruction (i.e., N_x last instruction). The example in the left part of the figure shows a valid execution that traverses existing edges in the program's CFG. The path is composed by the node sequence $\{N_1, N_3, N_6, N_4\}$ and generates the $\mathcal{CF}_{\mathsf{Log}}$ entries $\{A_1^i, A_1^e, A_3^i, A_3^e, A_6^i, A_6^e, A_4^i, A_4^e\}$. By inspecting $\mathcal{CF}_{\mathsf{Log}}$, $\mathcal{V}$rf can check the entries sequence to validate the control flow path and the fact that each node executes in its entirety (due to in order appearance of pairs $\{A_x^i, A_x^e\}$, i.e., valid entry and exit points of each node). The right part of the figure exemplifies an invalid execution path, containing an illegal transition from N_6 to N_2. For

instance, this could be caused by an attack (e.g., a buffer overflow) that overwrites `function2()`'s return address to point to `function1()`. In this second case, the control flow path is $\{N_1, N_3, N_6, N_2, N_5, N_4\}$, generating the $\mathcal{CF}_{\mathsf{Log}}$ entries $\{A_1^i, A_1^e, A_3^i, A_3^e, A_6^i, A_6^e, A_2^i, A_2^e, A_5^i, A_5^e, A_4^i, A_4^e\}$. By inspecting $\mathcal{CF}_{\mathsf{Log}}$, $\mathcal{V}\mathsf{rf}$ can identify the control flow abuse due to the invalid sequence of entries $\{A_6^i, A_6^e, A_2^i, A_2^e\}$ present in $\mathcal{CF}_{\mathsf{Log}}$.

C-FLAT [11] is the earliest work to introduce CFA and propose an instrumentation-based method using TrustZone as a TEE. OAT [67] optimizes CFA to reduce the number of measured control flow transfers and enforce data integrity. ReCFA [75] proposes to condense the control flow events to generate a compressed CFA report. LO-FAT [31], LiteHAX [30], and Atrium[74] propose to attest the control flow without instrumenting the executable binary, by adding customized hardware modules, such as a branch filter, a loop monitor, a hash controller, and a hash lookup. Tiny-CFA [28] proposes a hybrid design for low-end MCUs that uses instrumentation and only requires the formally verified hardware from support from APEX proofs of execution [47].

All of these approaches assume that attested applications execute atomically, meaning that they cannot be interrupted. To deal with interrupts, OAT [67] suggests instrumenting all ISRs along with the attested application. While this simplifies the evaluation of the $\mathcal{CF}_{\mathsf{Log}}$, it does not provide information on whether the interrupts have resumed correctly. Moreover, it is not suitable for MCUs running multiple applications. To the best of our knowledge, our work presents the first secure CFA architecture that can be applied to interrupt-prone and real-time systems and applications.

III. Problem Statement: Interrupts & CFA

This section discusses why current TEE-based CFA methods generate **unreliable reports** when interrupts are enabled. It also demonstrates associated attacks in practice. As discussed in Section I, this is an important problem because disabling all interrupts on the embedded device is often impractical.

A. Interrupt-based Attack Examples

We now present interrupt-based CFA attacks and discuss the fundamental limitations explored by these attacks. To demonstrate their practicality, we have also implemented an open-source example of the most general attack case (i.e., example 3 below). For more details on this implementation, see our repository at [2].

Figure 3 illustrates three execution possibilities for $\mathcal{P}\mathsf{rv}$ when interrupts are enabled. I_x represents the implementation of an untrusted ISR unrelated to the attested application $\mathcal{A}\mathsf{pp}$. Note that $\mathcal{V}\mathsf{rf}$ is only concerned with and knowledgeable about $\mathcal{A}\mathsf{pp}$'s binary and is oblivious to low-level system ISRs, such as I_x. Consequently, I_x is not instrumented, and its control flow transfers are not appended to $\mathcal{A}\mathsf{pp}$'s $\mathcal{CF}_{\mathsf{Log}}$ (because $\mathcal{V}\mathsf{rf}$ cannot interpret them). N_y and N_z represent nodes (lists of sequential instructions) within $\mathcal{A}\mathsf{pp}$. Whenever they execute, new entries are added to $\mathcal{CF}_{\mathsf{Log}}$. $\mathcal{A}\mathsf{pp}$ is instrumented to generate two new $\mathcal{CF}_{\mathsf{Log}}$ entries per node: one before executing the node's first instruction and one before the branching instruction (i.e., the last instruction) of the node. This instrumentation is in line with the TEE-based CFA methods discussed in Section II-D. Without interrupts, once a node's execution starts, instructions within the node run sequentially, and this approach generate a reliable $\mathcal{CF}_{\mathsf{Log}}$. However, this can not be guaranteed when interrupts are enabled, as discussed below.

Example 1 in Figure 3 illustrates a benign interrupt control flow transfer, where I_x does not tamper with $\mathcal{A}\mathsf{pp}$ control flow path. Execution starts from the first instruction in N_y, which is a trampoline to the Secure World to add new entry A_y^i to $\mathcal{CF}_{\mathsf{Log}}$. Before reaching the instruction in address 0x400, the interrupt is triggered, and execution is redirected to I_x. After I_x execution, N_y resumes correctly from address 0x400 and proceeds sequentially, finally reaching the second trampoline instruction that adds entry A_y^e to $\mathcal{CF}_{\mathsf{Log}}$. After executing the Node N_y, $\mathcal{CF}_{\mathsf{Log}}$ will have two new log entries $\{A_y^i, A_y^e\}$. Without interrupts or with a benign I_x that resumes N_y correctly, these two new $\mathcal{CF}_{\mathsf{Log}}$ entries indicate that all instructions within N_y executed.

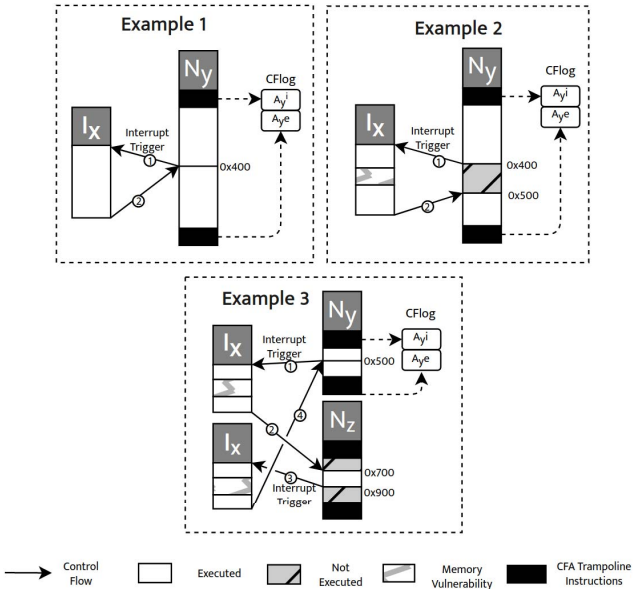


Fig. 3. Interrupt-based attacks on CFA.

We now consider the case where I_x is vulnerable/malicious, as illustrated in Example 2 of Figure 3. In this case, Adversary ($\mathcal{A}\mathsf{dv}$) could corrupt the return address of I_x (e.g., by exploiting a buffer overflow within I_x code), modifying it from 0x400 to 0x500. Once I_x returns, the CPU resumes the execution in address 0x500, skipping all the instructions between these addresses and consequently modifying the control flow and the behavior of $\mathcal{A}\mathsf{pp}$. However, in this case, the generated $\mathcal{CF}_{\mathsf{Log}}$ is the same as in Example 1 since the trampoline instructions of N_y are still reached in the same order.

Example 3 of Figure 3 illustrates an even worse case, where $\mathcal{A}\mathsf{dv}$ leverages control over interrupt configurations. For instance, timer-based interrupts can be configured to interrupt

$\mathcal{P}$rv precisely after a selected number of instructions (see our implementation in [3]). In Example 3, $\mathcal{A}$dv leverages this capability to change the return address of a first instance of I_x to the address 0x700 located inside a different $\mathcal{A}$pp Node: N_z. In addition, the attacker configures the interrupt I_x to be triggered again after a specific number of clock cycles: enough to execute the desired number of instructions but less than enough to reach the trampoline instruction that appends information to $\mathcal{CF}_{\mathsf{Log}}$. For instance, the second interrupt occurs after executing the instructions from 0x700 to 0x900 within N_z. At that point, I_x execution is triggered again. This time, the attacker changes I_x return address to 0x400 (the original return address of the first interrupt). At the end of Node N_y, the new $\mathcal{CF}_{\mathsf{Log}}$ entries generated by the described process would be exactly the same as Example 1. Therefore, $\mathcal{V}$rf would consider this report legal. In reality, however, this attack strategy allows $\mathcal{A}$dv to execute any number of sub-sequences of instructions (gadgets) within $\mathcal{A}$pp nodes, implying arbitrary code reuse and a complete modification of $\mathcal{A}$pp's intended behavior. Given an appropriate gadget set in $\mathcal{A}$pp (which occurs in most programs), this can result in undetected Turing-complete malicious behavior [60].

B. What makes existing CFA vulnerable to interrupt attacks?

Based on the previous discussion, we elucidate the fundamental limitations that enable interrupt-based attacks. This discussion serves as a guideline to the design of ISC-FLAT, presented next, in Section V.

Instrumentation without Atomicity. TEE-based CFA methods assume that instructions within a node are executed atomically. Therefore, the binary is only instrumented in each node's first and last instruction position. Interrupts falsify this premise, enabling control flow transfers at any point within the node. An naïve solution to this problem would be to instrument every instruction within the node. However, such an approach would incur extremely high overhead, requiring context switches between Secure and Normal Worlds for every executed instruction in $\mathcal{A}$pp.

Untraceable ISRs. ISRs are external to $\mathcal{A}$pp; thus, their behavior is not reflected in $\mathcal{CF}_{\mathsf{Log}}$. Since they are untraceable, security mechanisms should be in place to ensure non-interference of ISRs of the $\mathcal{A}$pp's execution.

No Stack Isolation. Unfortunately, ISRs run as privileged code in the Normal World (see [53] for related discussion). Therefore, they have access to all of the Normal World's stack (including control flow associated data and registers pushed to the stack during function calls). Conversely, they can modify interrupt configurations and also misconfigure Memory Protection Units (MPU) when applicable. These issues imply that any protection against malicious/compromised ISRs must be enforced by $\mathcal{P}$rv's Secure World.

IV. ISC-FLAT: Overview

ISC-FLAT architecture comprises three modules to support secure interruptable CFA: (i) Interrupt Safety Module (ISM), (ii) CFA Measurement Engine, and (iii) Remote Verification Engine.

- **Interrupt Safety Module (ISM):** This module aims to augment TEE-based CFA with the capability to generate reliable reports while enabling interrupts. To this end, ISM securely initializes attestation and creates a dispatcher inside the Secure World that interposes itself between interrupt triggers and the respective ISR execution. The dispatcher configures protections to the interrupted $\mathcal{A}$pp before the ISR can execute.
- **CFA Measurement Engine:** The measurement engine tracks $\mathcal{A}$pp's control flow and manages $\mathcal{CF}_{\mathsf{Log}}$. This is achieved through instrumentation of $\mathcal{A}$pp's binary. Before $\mathcal{A}$pp's deployment on $\mathcal{P}$rv, an automated instrumentation script adds instructions to $\mathcal{A}$pp's assembly code for each node of $\mathcal{A}$pp's CFG (as described in Section II-D). This instrumentation invokes a Secure World component that appends new entries to $\mathcal{CF}_{\mathsf{Log}}$ according to the control flow path taken during $\mathcal{A}$pp's execution.
- **Remote Verification Engine:** This module is executed by $\mathcal{V}$rf to analyze the CFA report and detect violations either due to software control flow attacks or malicious interrupts during $\mathcal{A}$pp's execution.

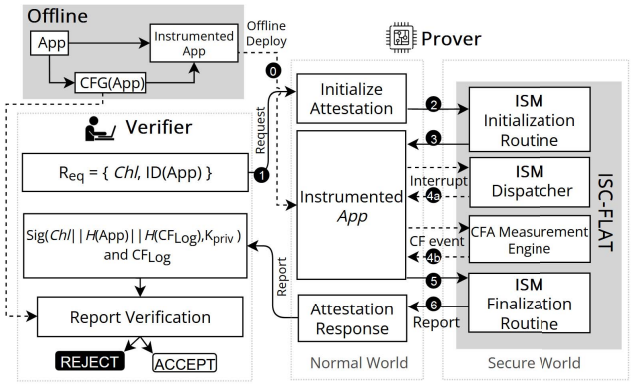


Fig. 4. Overview of ISC-FLAT protocol.

Figure 4 illustrates ISC-FLAT's protocol. Before being deployed on $\mathcal{P}$rv, each CFG node of $\mathcal{A}$pp binary is instrumented with additional instructions to track the execution control flow (step ⓿). These additional instructions activate the CFA engine by adding entries to $\mathcal{CF}_{\mathsf{Log}}$ during $\mathcal{A}$pp execution.

As indicated in step ❶, ISC-FLAT's protocol starts with $\mathcal{V}$rf locally generating a request $\mathcal{R}$eq: { $\mathcal{C}$hl, $ID(\mathcal{A}\mathsf{pp})$ } composed of: a challenge/nonce ($\mathcal{C}$hl) and an identifier ($ID(\mathcal{A}\mathsf{pp})$), where $\mathcal{A}$pp is the application that should be executed and attested. $\mathcal{C}$hl is a unique random number to provide liveness, preventing replayed CFA reports. To initiate an attested execution of $\mathcal{A}$pp, $\mathcal{V}$rf sends $\mathcal{R}$eq to $\mathcal{P}$rv.

Upon receiving $\mathcal{R}$eq, $\mathcal{P}$rv must call the ISM Initialization Routine, implemented in the Secure World, to create an attestation instance associated with $\mathcal{C}$hl (step ❷). Failure to do so implies the inability to produce a valid CFA report (since the ISM finalization routine – detailed below – only signs CFA reports associated with initialized attestation instances). ISM Initialization Routine also computes a hash of all the memory

regions in $\mathcal{P}$rv that contain $\mathcal{A}$pp's executable, resulting in $H(\mathcal{A}\mathsf{pp})$. Next, it makes the memory regions storing this binary immutable (see Section V-D for details). Similar to $\mathcal{C}$hl, $H(\mathcal{A}\mathsf{pp})$ is also assigned to this attestation instance. $\mathcal{C}$hl and $H(\mathcal{A}\mathsf{pp})$ are defined during initialization (before $\mathcal{A}$pp execution) and included in the signed CFA report (see below). Therefore, any attempt to initialize an incorrect binary $\mathcal{A}\mathsf{pp}_{\mathcal{A}\mathsf{dv}}$ is detectable by $\mathcal{V}$rf during verification, due to the mismatch $H(\mathcal{A}\mathsf{pp}) \neq H(\mathcal{A}\mathsf{pp}_{\mathcal{A}\mathsf{dv}})$. Finally, the ISM Initialization Routine uses NVIC to configure all interrupts to be handled by the Secure World's ISM dispatcher. The dispatcher, in turn, is responsible for redirecting interrupts to their real ISRs after enabling ISC-FLAT's interrupt protections.

Following ISM Initialization Routine, $\mathcal{A}$pp execution starts (step ❸). During execution, all control flow transfers are saved to $\mathcal{C}\mathsf{F}_{\mathsf{Log}}$ in the Secure World (step ❹a) by the CFA engine. Similar to $\mathcal{C}$hl and $H(\mathcal{A}\mathsf{pp})$, $\mathcal{C}\mathsf{F}_{\mathsf{Log}}$ is associated to this particular attestation instance. During $\mathcal{A}$pp execution, due to the initial NVIC configuration, all interrupts are trapped to ISM before their handling by their respective (untrusted) ISRs (step ❹b). For all interrupts, ISM Dispatcher will save $\mathcal{A}$pp's context and "lock" $\mathcal{A}$pp's state (including $\mathcal{A}$pp's stack). After this process, ISM redirects the interrupt to its actual (untrusted) ISR, located in the Normal World. Once finalized, ISR must return to the ISM Dispatcher module to re-enable $\mathcal{A}$pp's execution (i.e., unlock $\mathcal{A}$pp's state and stack). In this final stage, the ISM Dispatcher assures that $\mathcal{A}$pp's control flow and context are resumed appropriately. This protects $\mathcal{A}$pp's execution against (otherwise untraceable) interrupt-based CFA attacks.

Once $\mathcal{A}$pp execution is over, the ISM Finalization Routine is called (step ❺) to produce a signed CFA report ($\mathcal{R}(\mathcal{A}\mathsf{pp})$) containing the produced $\mathcal{C}\mathsf{F}_{\mathsf{Log}}$ along with $\mathcal{C}$hl and $H(\mathcal{A}\mathsf{pp})$. Optionally, the report may include any output (out) produced by $\mathcal{A}$pp's execution. If all steps succeed, $\mathcal{R}(\mathcal{A}\mathsf{pp})$ is sent to $\mathcal{V}$rf (step ❻). Finally, $\mathcal{V}$rf uses the Remote Verification Engine to decide on $\mathcal{R}(\mathcal{A}\mathsf{pp})$'s trustworthiness (The remote verification process is detailed in Section V-E).

V. ISC-FLAT in Detail

This section further details each component in ISC-FLAT. We start by specifying the system and adversary models. Then we describe each of ISC-FLAT's components, namely CFA Measurement Engine, Interrupt Security Module (ISM), and Remote Verification Engine.

A. System Model

We consider that $\mathcal{P}$rv is a single-core, bare-metal MCU, equipped with a TEE, such as ARM TrustZone-M. $\mathcal{P}$rv hosts multiple untrusted applications, including untrusted privileged software in the form of ISRs or a simple real-time operating system (RTOS). All untrusted software modules (including the application to be attested – $\mathcal{A}$pp) execute in the Normal World to keep security critical functionality (including the trusted CFA implementation) isolated within the Secure World. In line with the CFA related work (see Section II-D), we assume the following $\mathcal{P}$rv features, which are implemented by the ARM TrustZone-M (v8) architecture:

- $\mathcal{P}$rv can securely store a secret key (sk) within the Secure World (making it inaccessible to the Normal World). The (ideally minimal) code inside the TEE's Secure World is trusted.
- $\mathcal{P}$rv features separate IVTs for Normal and Secure Worlds. Any interrupts can be delegated to either one of the IVTs. The Secure-IVT has priority over the Normal-IVT (when an interrupt exists in both IVTs).
- $\mathcal{P}$rv features a Non-Secure Memory Protection Unit (NS-MPU). NS-MPU is a hardware monitor that controls memory access in the Normal World. Nonetheless, the Secure World can hijack control of the NS-MPU by restricting Normal World's access to NS-MPU configuration registers[1]. As discussed in Section V-D, ISC-FLAT leverages this capability to ephemerally make $\mathcal{A}$pp's code immutable yet executable in the Normal World.

We assume that $\mathcal{V}$rf has knowledge of $\mathcal{A}$pp's binary and CFG but is unaware of other (potentially malicious/vulnerable) application/system-level software executing on $\mathcal{P}$rv. $\mathcal{V}$rf also has knowledge of the public key (pk) corresponding to the sk (stored within $\mathcal{P}$rv's Secure World).

B. Adversary Model

We consider a strong adversary ($\mathcal{A}$dv) that has full control of $\mathcal{P}$rv's Normal World (including the privileged mode in the Normal World). $\mathcal{A}$dv can modify Normal World code (e.g., through code injection attacks) and trigger control flow hijacks and code re-use attacks. Similarly, $\mathcal{A}$dv can control interrupt configurations to call ISRs at any time [19] and modify/corrupt ISR implementations. On the other hand, $\mathcal{A}$dv is unable to tamper with Secure World-resident software and data. Conversely, it cannot disable or bypass TEE hardware-enforced access control rules and guarantees.

Compared to prior work, this threat model considers a stronger $\mathcal{A}$dv that leverages interruptions and malicious Normal World code during CFA of $\mathcal{A}$pp. As such, it presents a more realistic case where CFA must securely co-exist with the real-time requirements and multiple tasks in embedded devices. We consider invasive physical attacks that modify hardware out-of-scope, as they require an orthogonal set of anti-tampering methods [57].

C. CFA Measurement Engine

The measurement engine generates $\mathcal{C}\mathsf{F}_{\mathsf{Log}}$ and works in parallel with ISM (described in Section V-D) to support secure interruptable CFA. It leverages binary instrumentation to construct $\mathcal{C}\mathsf{F}_{\mathsf{Log}}$ during $\mathcal{A}$pp execution.

Instrumenting Control Flow Transfers. Static analysis [64] is used to generate $\mathcal{A}$pp's CFG, denoted $CFG(\mathcal{A}\mathsf{pp})$ (recall the CFG definition from Section II-D). The CFG is used to both (i) instrument $\mathcal{A}$pp's binary before deployment

[1]In Armv8-M is possible to restrict the Normal World from accessing the NS-MPU by marking the NS-MPU configuration memory region as belonging to the Secure World, using TrustZone-M SAU (recall SAU from Section II-B).

(see below); and (ii) by $\mathcal{V}$rf to verify the CFA reports (see Section V-E). Before deployment, each node of $CFG(\mathcal{A}\text{pp})$ in $\mathcal{A}$pp's binary is instrumented to save control flow events to $\mathcal{CF}_{\text{Log}}$. The additional instructions are trampolines that redirect the execution to a function implemented within the Secure World that appends the control flow information to $\mathcal{CF}_{\text{Log}}$. As $\mathcal{CF}_{\text{Log}}$ is stored within the Secure World, it cannot be directly modified by Normal World software. Specifically, the instrumentation is added before each node's first and last instructions (recall that the last instruction is the branching instruction of the node). Therefore, during execution, $\mathcal{CF}_{\text{Log}}$ will be constructed as the sequence of control flow transfers, containing two entries per node.

In each node's entry point, the instrumentation introduces a single *branch and link* (*bl*) instruction that branches to a Non-Secure Callable (NSC) function. The NSC function calls the CFA engine and sends the Link Register (LR) value (containing the returning address) as an argument to be recorded in the $\mathcal{CF}_{\text{Log}}$. The entry generated at this point represents the memory address of the node's first instruction, and its presence in $\mathcal{CF}_{\text{Log}}$ indicates that the node execution started from the first instruction.

Two cases must be considered for the instrumentation at the end of a node, depending on the nature of the node's branching instruction. If the branching instruction is a *conditional branch, direct jump, or call*, the additional instruction is the same as for the entry point, and the value of the log entry represents the memory address of the node's last instruction. If the branching instruction is an *indirect jump/call or return*, the entry to be logged is the (dynamically determined) branch destination. In this case, the instrumentation adds two instructions: first, it performs a *push* of the destiny address to the stack and then a *bl* to an NSC function that will *pop* the destiny address from the stack and send it to the CFA engine. By using this method, dynamically defined illegal branches can be detected. The instrumentation approach outlined here permits the observation of all direct and indirect branches that take place during the execution of $\mathcal{A}$pp. Upon completing the instrumentation process, $\mathcal{A}$pp can be installed on $\mathcal{P}$rv, where it can operate as a Normal World application.

Measuring the Control Flow. Once the ISM Initialization Routine is called, it allocates a secure memory space to store $\mathcal{CF}_{\text{Log}}$ within the Secure World and sets a flag to enable the addition of new log entries. From this point until the end of $\mathcal{A}$pp's execution, every time the CPU reaches the trampoline instructions, it will pass the control to the Secure World, which will append the related memory addresses (entries) in the memory that comprises $\mathcal{CF}_{\text{Log}}$.

D. Interrupt Security Module (ISM)

ISM is the key feature to make CFA reliable when interrupts are enabled. As shown in Figure 5, ISM is implemented in the Secure World to safeguard $\mathcal{A}$pp against all interrupts before they are handled by their untrusted ISRs in the Normal World. In doing so, ISM assures the CFA integrity of the interrupted $\mathcal{A}$pp on the following fronts: (i) protecting the stack portion containing the context and the returning address of the interrupted $\mathcal{A}$pp; (ii) blocking ISRs from executing/modifying code sections in $\mathcal{A}$pp or generating/tampering with $\mathcal{CF}_{\text{Log}}$ entries; and (iii) ensuring that $\mathcal{A}$pp is resumed correctly.

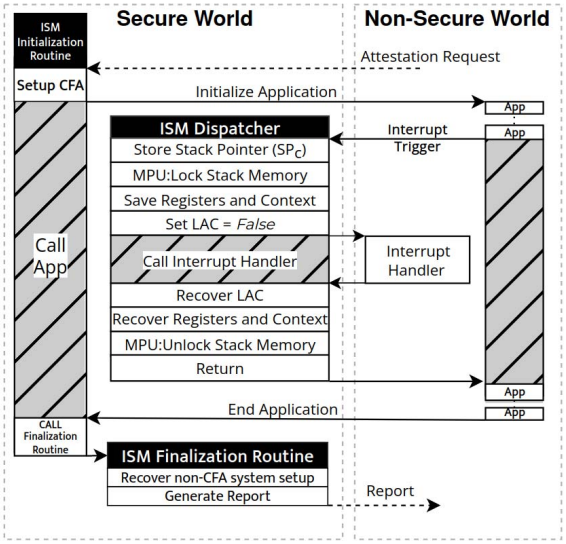


Fig. 5. Illustration of the ISM workflow.

ISM includes three sub-modules. ISM Initialization Routine is responsible for all initial configurations that will guarantee the security of the attestation process and activate the secure interrupt dispatcher. ISM Dispatcher intermediates every ISR activation to restrict access to the memory belonging to $\mathcal{A}$pp as well as protecting $\mathcal{A}$pp control flow integrity from external interference. ISM Finalization Routine is responsible for generating the signed CFA report, restoring pre-attestation system configurations, and finalizing the attestation instance.

1) **ISM Initialization Routine:** When $\mathcal{P}$rv receives the request $\mathcal{R}$eq from $\mathcal{V}$rf, the ISM Initialization Routine function, located in the Secure World, must be called to initialize the attestation. This routine contemplates necessary configurations to initialize and ensure the CFA security by following the subsequent steps:

- **Step 1:** Check if another application is currently being attested. If so, returns an error flag.
- **Step 2:** Configure TrustZone SAU to set the memory region containing NS-MPU rule configurations as a Secure World region. This revokes Normal World permission to configure NS-MPU.
- **Step 3:** Configure the NS-MPU to change the permission of $\mathcal{A}$pp's program memory addresses to read-only. *This will protect $\mathcal{A}$pp's binary against modifications during its execution/attestation*.
- **Step 4:** Hash $\mathcal{A}$pp's program memory, generating $H(\mathcal{A}\text{pp})$. $\mathcal{V}$rf will use this to later verify $\mathcal{A}$pp's binary integrity.
- **Step 5:** Assign all interrupts to be handled by the interrupt *Dispatcher*, in the Secure World. This is achieved by setting NVIC_ITNS. All the interrupts are set as Secure Interrupts by activating all entries in TrustZone's NVIC Secure-IVT to point to the ISM Dispatcher. Therefore, the ISM Dispatcher

obtains the ability to interpose itself between all interrupts and the execution of their (untrusted) ISRs in the Normal World.

• **Step 6:** Save the current value of the Normal World's Stack Pointer SP_0. As this is done immediately before $\mathcal{A}$pp execution starts, the Dispatcher can use SP_0 to determine the location of $\mathcal{A}$pp's stack and "lock" it, i.e., prevent modifications during the execution of eventual ISRs.

• **Step 7:** Set the Log Access Control ($LAC = True$) flag. LAC indicates that the measurement engine is allowed to add new entries to $\mathcal{CF}_{\mathsf{Log}}$.

• **Step 8:** Call $\mathcal{A}$pp function in the Normal World.

2) **ISM Dispatcher:** While $\mathcal{A}$pp is running, the interrupt *Dispatcher* handles all the interrupts before redirecting them to their original ISR implementation. The Dispatcher's goals are to: (i) track all interrupts and control their permissions to add new entries in $\mathcal{CF}_{\mathsf{Log}}$ (ii) guarantee that the $\mathcal{A}$pp's context, including registers, the return address, and data in the stack, is the same after the interrupting ISR completes; (iii) prevent any attempt to bypass the Dispatcher and avoid the required context-recovery from happening. As soon as an interrupt is triggered, Secure World assumes the CPU control and creates an instance of the Dispatcher to track the interrupt. At this point, the Dispatcher saves the current Non-Secure Stack Pointer on a temporary variable (SP_c). Depending on the architecture, it is also necessary to store other critical registers that need protection (i.e., any registers not pushed to the stack before the interrupt). Next, the Dispatcher locks the $\mathcal{A}$pp's stack by setting the NS-MPU to define the region within addresses [SP_0, SP_c] as a read-only (recall that after Initialization Routine, NS-MPU cannot be modified by the Normal World). This will temporarily prevent the Normal World from overwriting the $\mathcal{A}$pp's stack, protecting it from ISR interference. Finally, the Dispatcher stores the previous LAC value and sets $LAC = False$, implying that no entries will be appended to $\mathcal{CF}_{\mathsf{Log}}$ until $LAC = True$ again. This prevents the ISR from adding fake entries to $\mathcal{CF}_{\mathsf{Log}}$.

The Dispatcher then passes control to the Normal World ISR and waits for the ISR. To find the address of the Non-Secure ISR code, the Dispatcher looks-up the NS-IVT position associated with the triggered interrupt. Note that $\mathcal{A}$pp execution cannot be resumed unless the ISR returns to the Dispatcher. Meanwhile, the $\mathcal{A}$pp's context is blocked. As a consequence, in order to produce a valid attestation report, the untrusted ISR must eventually give control back to the Dispatcher, by returning appropriately, otherwise, a valid attestation report can not be produced for $\mathcal{A}$pp execution. When the Dispatcher receives control back, it changes LAC to its previous value and recovers all the $\mathcal{A}$pp's context, including the stack pointer register, to the same values as before the interrupt triggering. Finally, the previous configuration of the NS-MPU is restored, and $\mathcal{A}$pp execution is resumed.

Interrupt Preemption. Note that a new dispatcher instance is created every time an interrupt is triggered. In the case of a preempting interrupt, multiple dispatcher instances – each related to each active interrupt – will exist simultaneously. Whenever the preemption is triggered, the LAC value, registers, and NS-MPU configuration belonging to the preempted task are pushed onto a stack within the Secure World. Then, the new dispatcher instance sets LAC to False and reconfigures the NS-MPU accordingly. When the preempting interruption concludes, the former NS-MPU/LAC/registers values are popped from the protected stack.

3) **ISM Finalization Routine:** Once the execution of $\mathcal{A}$pp ends, the ISM finalization routine, implemented in the Secure World, is automatically called to generate the CFA report and reset configurations made by the Initialization Routine at the beginning of attestation (before $\mathcal{A}$pp attested execution). We note that the finalization routine function is not callable by the Normal World. Thus, signing the CFA report without a prior call to initialize the attestation is impossible. The signed report is bound to the data produced and stored by the latest ISM Initialization Routine call, including $\mathcal{C}$hl and $H(\mathcal{A}\mathsf{pp})$. Specifically, the finalization routine performs the following actions:

Producing CFA Report. The final CFA report is denoted by $\mathcal{R}(\mathcal{A}\mathsf{pp}) = \{\sigma^{\mathcal{A}\mathsf{pp}}, \mathcal{CF}_{\mathsf{Log}}\}$. It contains all information needed to prove to $\mathcal{V}$rf which control flow path was taken by $\mathcal{A}$pp. $\mathcal{CF}_{\mathsf{Log}}$ is the verbatim control flow path. The signature $\sigma^{\mathcal{A}\mathsf{pp}}$ has the format

$$\sigma^{\mathcal{A}\mathsf{pp}} = Sig^{sk}(H(\mathcal{CF}_{\mathsf{Log}})||H(\mathcal{A}\mathsf{pp})||\mathcal{C}\mathsf{hl}),$$

where Sig^{sk} denotes a cryptographic signature computed using Secure World's secret key sk. The signature is computed on $H(\mathcal{CF}_{\mathsf{Log}})$, $H(\mathcal{A}\mathsf{pp})$, and $\mathcal{C}$hl. Therefore, it authenticates $\mathcal{CF}_{\mathsf{Log}}$, $\mathcal{A}$pp's binary and proves the freshness of $\mathcal{R}(\mathcal{A}\mathsf{pp})$ (through $\mathcal{C}$hl). Optionally, $\mathcal{R}(\mathcal{A}\mathsf{pp})$ (and respective signature) may include any results produced by $\mathcal{A}$pp's execution, proving that the results (e.g., sensed quantities) were produced by a trustworthy execution of $\mathcal{A}$pp.

Restoring Non-CFA System Status. The finalization routine sets $LAC = False$ to disable new additions to the $\mathcal{CF}_{\mathsf{Log}}$. In addition, it re-configures Non-Secure Interrupts to be handled in the Normal World by their original ISRs without going through the Dispatcher checks. The original NS-MPU configuration also returns to its original state.

E. Remote Verification Engine (executed by $\mathcal{V}$rf)

Authentication. Upon receiving $\mathcal{R}(\mathcal{A}\mathsf{pp})$ from $\mathcal{P}$rv, $\mathcal{V}$rf verifies the signature using pk and based on the hash of $\mathcal{A}$pp's expected binary, $\mathcal{C}$hl, and according to the copy of $\mathcal{CF}_{\mathsf{Log}}$, that is included in $\mathcal{R}(\mathcal{A}\mathsf{pp})$.

Control Flow Validation. If all authentication checks (above) succeed, $\mathcal{V}$rf is assured that $\mathcal{CF}_{\mathsf{Log}}$ corresponds to $\mathcal{A}$pp's execution on $\mathcal{P}$rv. As a final step, $\mathcal{V}$rf can analyze $\mathcal{CF}_{\mathsf{Log}}$ through a variety of means. The simplest approach is to run $\mathcal{CF}_{\mathsf{Log}}$ through $\mathcal{A}$pp's legal CFG (recall Figure 2). Furthermore, to verify backward edges (e.g., return address integrity) $\mathcal{V}$rf emulates a shadow stack [68] for $\mathcal{A}$pp's reported $\mathcal{CF}_{\mathsf{Log}}$.

VI. Security Analysis

ISC-FLAT security argument is two-part: (1) we show security when interrupts are disabled; and (2) we show that ISM assures that $\mathcal{CF}_{\mathsf{Log}}$-s produced by ISC-FLAT when interrupts

are enabled are equivalent to those produced in case (1), when interrupts are disabled.

A. Security Argument – CFA engine

Recall from Section V-A that $\mathcal{A}$dv takes advantage of vulnerabilities in the Normal World attempt forgery of CFA reports. To attack the CFA engine without interrupts, $\mathcal{A}$dv must: (i) modify or forge $\mathcal{CF}_{\mathsf{Log}}$; (ii) deactivate the CFA engine by removing $\mathcal{A}$pp's instrumentation; or (iii) generate a CFA report that is accepted by $\mathcal{V}$rf as authentic when it does not correspond to a timely (i.e., after issuance of the latest $\mathcal{C}$hl) execution of $\mathcal{A}$pp on $\mathcal{P}$rv.

Forging $\mathcal{CF}_{\mathsf{Log}}$. The trivial approach is directly modifying $\mathcal{CF}_{\mathsf{Log}}$ in $\mathcal{P}$rv's memory. However, ISC-FLAT keeps $\mathcal{CF}_{\mathsf{Log}}$ inside the Secure World, which is inaccessible to the Normal World. Another possibility is to call the trampoline function directly to add arbitrary entries to $\mathcal{CF}_{\mathsf{Log}}$. However, the code to perform these illegal trampoline calls cannot be injected in $\mathcal{A}$pp's program memory, since this region is made immutable by the Initialization Routine. Therefore, $\mathcal{A}$dv must jump from $\mathcal{A}$pp to malicious code that calls the trampoline (outside of $\mathcal{A}$pp) by corrupting the destination of an indirect branch. Since all indirect branches within $\mathcal{A}$pp are instrumented, this attempt would be appended to $\mathcal{CF}_{\mathsf{Log}}$ and thus detectable by $\mathcal{V}$rf.

Removing $\mathcal{A}$pp **instrumentation**. Instrumentation integrity is ensured through $H(\mathcal{A}\mathsf{pp})$, included in $\mathcal{R}(\mathcal{A}\mathsf{pp})$. Once the ISM Initialization Routine is called, **before** $\mathcal{A}$pp execution, it computes $H(\mathcal{A}\mathsf{pp})$. $\mathcal{A}$pp's binary is unmodifiable thereafter (until ISM finalization routine execution) due to the NS-MPU and SAU protections enforced by ISC-FLAT initial configurations. Thus, any attempt to modify $\mathcal{A}$pp must be made before the ISM Initialization Routine call, resulting in an incorrect $H(\mathcal{A}\mathsf{pp}^{\mathcal{A}\mathsf{dv}})$ value that is detected by $\mathcal{V}$rf.

Forging CFA Report ($\mathcal{R}(\mathcal{A}\mathsf{pp})$**).** $\mathcal{A}$dv may attempt to forge $\mathcal{R}(\mathcal{A}\mathsf{pp})$ to reflect the expected execution of $\mathcal{A}$pp when that execution did not happen. This requires finding $\sigma^{\mathcal{A}\mathsf{dv}}$ such that $Vrfy^{pk}(\sigma^{\mathcal{A}\mathsf{dv}}) = (H(\mathcal{A}\mathsf{pp})||H(\mathcal{CF}_{\mathsf{Log}})||\mathcal{C}\mathsf{hl})$. sk is securely stored within the Secure world. Therefore, this forgery is computationally infeasible as long as $(Sig, Vrfy)$ is implemented using a cryptographically secure signature.

B. Security Argument – Interrupt Safety Module

ISM supports the CFA engine by preserving the integrity of CFA reports irrespective of interrupts. To bypass ISM, $\mathcal{A}$dv must (i) deactivate the ISM Dispatcher; (ii) replace an ISR address in the Non-Secure IVT; or (iii) corrupt $\mathcal{A}$pp's data in the Normal World stack.

Deactivating ISM Dispatcher. $\mathcal{A}$dv could attempt to remove the dispatcher integrity checks that must occur for each interrupt to safeguard $\mathcal{A}$pp's context. However, the Secure World controls the NVIC interrupt configuration, which cannot be altered by an $\mathcal{A}$dv that compromises the Normal World.

Modifying Interrupt Control Flow. When the Dispatcher redirects the execution to an untrusted ISR, it must eventually receive the control back to resume $\mathcal{A}$pp's context and execution. $\mathcal{A}$dv could attempt to modify the ISR's return address and redirect the control flow, instead of returning to the Dispatcher or even add bogus entries to $\mathcal{CF}_{\mathsf{Log}}$ by calling the trampoline function directly. ISC-FLAT disables logging of new entries to $\mathcal{CF}_{\mathsf{Log}}$ (through the LAC flag) and blocks $\mathcal{A}$pp's stack to the NS-World, pausing attestation until any ISRs return control to the Dispatcher. While attestation is paused, nothing can affect the $\mathcal{A}$pp's context. If the execution returns to the Dispatcher, it will recover the $\mathcal{A}$pp's saved context and resume execution appropriately. Otherwise, ISC-FLAT will not produce a signed CFA report, and $\mathcal{V}$rf would conclude that $\mathcal{A}$pp execution failed.

Corrupting $\mathcal{A}$pp **Stack.** When ISC-FLAT Dispatcher is triggered (due to an ISR) while $\mathcal{A}$pp executes in the Normal World, a hardware routine pushes the contents of several registers onto the Non-Secure stack. After the ISR execution ends, the Dispatcher loads a specific value into the program counter (PC) called EXEC_RETURN. This particular value triggers an "end of interrupt" routine, where the hardware resumes the execution state to what it was before the interrupt was triggered, including its return address. Since the Non-Secure stack is accessible to the untrusted ISR, the saved return address could be corrupted, leading the Dispatcher to return to the wrong location. $\mathcal{A}$dv could also change another context in the stack, such as saved register values and variables. By locking the Non-Secure stack belonging to $\mathcal{A}$pp and ensuring the integrity of the stack pointer, ISC-FLAT guarantees that $\mathcal{A}$pp's context is appropriately resumed, including the ISR return address and the $\mathcal{A}$pp's registers and data memory values.

VII. Evaluation

We implement ISC-FLAT's proof-of-concept prototype on a NUCLEO-L552ZE-Q [4] development board that is equipped with an STM32L552ZE MCU. This MCU is based on the ARM Cortex-M33 (v8) architecture and operates at a clock frequency of 110 MHz. The MCU does not feature a memory cache and supports Arm TrustZone-M technology. Our implementation includes a python script to instrument the $\mathcal{A}$pp's assembly code, as required by the CFA engine. The trusted components, ISM and CFA engine, are implemented in `C` and run in the Secure World. An interface module that facilitates communication between the Secure and Normal World is also implemented in `C`. The CFA verification engine, executed by $\mathcal{V}$rf, is implemented in Python. In our experiments, we use a fixed $\mathcal{CF}_{\mathsf{Log}}$ buffer of size 4096 Bytes, which is configurable to match the $\mathcal{A}$pp's requirements and available resources. Assuming a bare-metal system, our prototype features ISC-FLAT running in the Secure World and $\mathcal{A}$pp running in the Normal World. In the Secure World, ISC-FLAT operates in the Thread privileged mode, except for the Dispatcher, which runs in the Handler mode. Within the Normal World, $\mathcal{A}$pp runs in the Thread privileged mode, while its interrupts operate in the Handler mode. Section VIII provides additional information on running $\mathcal{A}$pp or ISC-FLAT using unprivileged mode within the context of a Rich/Trusted OS context.

We evaluate ISC-FLAT prototype to (i) determine the time,

memory and energy consumption overheads of each individual ISC-FLAT module; (ii) determine what ISC-FLAT's impact on real applications is; and (iii) test the approach against possible attacks.

A. Runtime Overhead

Four modules contribute to ISC-FLAT overhead: Four modules contribute to ISC-FLAT overhead: (1) ISM Initialization Routine, to initialize attestation and hash the binary memory; (2) ISM Dispatcher, which handles interrupts during the attested execution; (3) ISM finalization routine, to finalize attestation and generate the CFA report; and (4) CFA engine, which appends new entries to $\mathcal{CF}_{\mathsf{Log}}$.

The runtime of each of the four modules is measured as follows:

- ISM Initialization Routine: elapsed time measured from the instruction that calls the attestation (ISM Initialization Routine) until the point where the first instruction of $\mathcal{A}\mathsf{pp}$ is reached.
- ISM Finalization Routine: elapsed time measured from $\mathcal{A}\mathsf{pp}$'s last instruction until the first instruction after finishing attestation.
- CFA engine: elapsed time measured from when the trampoline instruction is called until the execution of the first instruction after returning to the Normal World.

The ISM Dispatcher module is evaluated using two different metrics:

- Interrupt Latency: time between interrupt triggering and execution of the corresponding ISR, including the trusted dispatcher execution time.
- Interrupt Backtrip Latency: time to resume $\mathcal{A}\mathsf{pp}$ execution after an ISR returns, including the dispatcher execution to assure $\mathcal{A}\mathsf{pp}$'s integrity and interrupt flag resets.

We measured the runtime of each ISC-FLAT module 10^5 times. The results are shown in Table I. The overhead due to the initialization and finalization routines is fixed as these modules are only executed once per attested execution. The primary source of overhead in these modules is the cryptographic operations (this cost can potentially be reduced significantly on devices with hardware-accelerated cryptographic instructions). The majority of the runtime in the ISM Initialization Routine comes from the hash computation of $H(\mathcal{A}\mathsf{pp})$. In our measurements, we employed the blake2s[1] hash function and measured that a binary size of 1 KByte required 211864 CPU cycles, while the finalization routine took approximately 843816 CPU cycles for a $\mathcal{CF}_{\mathsf{Log}}$ of size 4kBytes. The combined runtime of both modules increases linearly with the total size of $\mathcal{CF}_{\mathsf{Log}}$ plus the binary ($\mathcal{A}\mathsf{pp}$) size at a rate of 182103 CPU cycles per additional 1 KByte. This cost is associated to the hash and signature operations used to generate the CFA report.

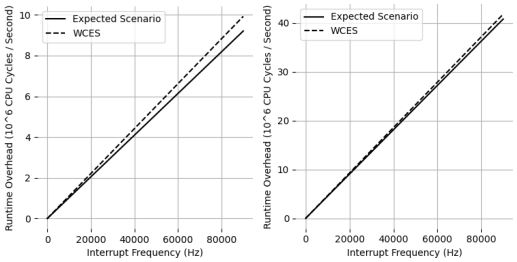
As depicted in Table I, each control flow logging event generated by the CFA engine requires approximately 491 CPU cycles, primarily due to the context switches between the Normal and Secure World. The CFA engine is triggered every time the execution reaches an instrumentation point

TABLE I
RUNTIME AND ENERGY MEASUREMENTS OF ISC-FLAT

Module	CPU Cycles		
	Mean	**Std**	**Max**
ISM Initialization Routine	211865	13	211878
ISM Finalization Routine	843816	13	843831
CFA engine	491	8	502
Without Dispatcher			
Interrupt Latency	29	2	32
Interrupt Latency Backtrip	405	8	415
With Dispatcher			
Interrupt Latency	103	5	110
Interrupt Latency Backtrip	452	8	463

(the beginning or end of each CFG node). Consequently, the relative overhead (%) is proportional to the ratio between branching and non-branching instructions. Instructions that do not branch do not require instrumentation and thus do not incur additional execution time. Thus, a greater number of non-branching instructions reduces the relative execution overhead. The worst case execution time (WCET) for the CFA engine occurs when all of the instructions are branching instructions (a theoretical case that would not occur in real programs). This case would introduce an overhead of 491 CPU cycles for each instruction.

To assess the efficiency of the ISM Dispatcher, we measured the interrupt latency with and without it. Our results show that the interrupt latency introduced by the ISM Dispatcher was approximately 103 CPU cycles, which results in 74 CPU cycles overhead compared to the baseline interrupt. However, we believe that this overhead is acceptable for most applications, considering the typical application latency requirements outlined in [50]. Additionally, the interrupt backtrip latency was around 452 CPU cycles, which represents an increase of 46 CPU cycles over the baseline. These values are not affected by the ISR code size or execution time, as they occur before and after the ISR execution. Therefore, the total runtime overhead introduced by ISC-FLAT during an attested execution of $\mathcal{A}\mathsf{pp}$ depends on the frequency of interrupts. The expected overhead generated by the Dispatcher and the WCET for different interrupt frequencies are shown in Figure 6.



(a) Interrupt Latency (b) Interrupt Backtrip Latency

Fig. 6. ISC-FLAT Dispatcher runtime vs. interrupt frequency

B. TCB Size

Table 1 presents the number of lines of C/Assembly code (calculated using "cloc v1.90") implemented for each ISC-FLAT module and the corresponding compiled binary size (optimization flag "-O0") that is incorporated into the TCB

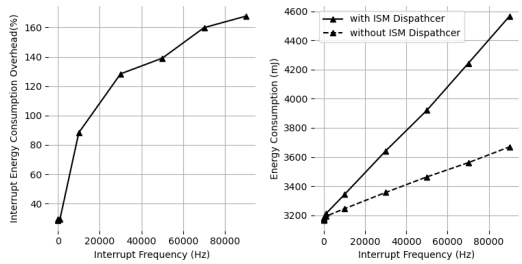
(including dependencies). ISC-FLAT implementation in the Secure World is written in a total of 542 lines of code (including an external Blake2s Library). As anticipated, the CFA engine is a small portion of the $\mathcal{A}$pp binary due to its limited role of adding new entries to $\mathcal{CF}_{\mathsf{Log}}$. On the other hand, the ISM Initialization and Finalization have larger code sizes, as they implement a range of operations, including cryptographic functions and register assignments. The Dispatcher, with an intermediate code size, performs fewer system configurations. For a reference point, OAT [67] reports 916 lines of C and assembly code.

TABLE II
TCB SIZE OF ISC-FLAT MODULES.

Module	Lines of Code	Binary Size (Kb)
ISM Initialization Routine	242	6.31
ISM Finalization Routine	73	1.78
ISM Dispatcher	32	1.22
CFA engine	20	0.38
Blake2s Library	177	3.34
Total	562	13.03

C. Energy Consumption

To evaluate the energy consumption of ISC-FLAT, we used a X-NUCLEO-LPM01A [9] evaluation board. This board powers the MCU with 3.3V, isolating it from the energy source of the other components on the MCU prototyping board. It measures the MCU's current usage with a 100kHz acquisition rate. To determine the energy consumption introduced by the Dispatcher at varying interrupt frequencies, we measured the MCU's current consumption during 100s of "Sleep Mode" with active interrupts. The "Sleep Mode" is deactivated when the interrupt handler executes and reactivated when it ends. We consider three configurations for the experiment: (i) no interrupts, (ii) with a single interrupt at different frequencies, and (iii) with the Dispatcher activated. The results of setups (ii) and (iii) are shown in Figure 7b. To isolate the Dispatcher overhead, we subtracted the consumption of setup (i) (without interrupts) from setups (ii) and (iii). The increment in (iii) compared to (ii) is shown in Figure 7a. Our results indicate that ISC-FLAT introduces an overhead of 29% at low-frequency interrupts, which increases to 167.7% as the frequency approaches 90kHz.It is important to note that the results in this subsection offer a micro-level perspective of interrupt energy consumption. The broader impact on the overall system will be assessed through case studies in Section VII-D.



(a) ISM Dispatcher (b) Sleep Mode Consumption

Fig. 7. ISM Dispatcher energy consumption.

D. Case Study Applications

In this section, we evaluate the performance of ISC-FLAT interrupts on real embedded applications, aiming to demonstrate the effects of ISC-FLAT overhead due to the ISM module. ISC-FLAT is the first interrupt-safe CFA, thus there are no approaches that achieve the same functionality. Therefore, we compare it to regular CFA to account for the additional cost of supporting interrupts safely. Our evaluation considers the overhead introduced by the CFA engine (required by any TEE-based CFA method, irrespective of interrupt-safety) over the baseline application without any ISC-FLAT modules. Most importantly, it isolates the overhead due to the ISM module (our main contribution) over the baseline CFA engine. Finally, we also experiment with ISC-FLAT defenses against different interrupt attack modes.

In order to assess ISC-FLAT in real-world scenarios, we customized three open-source applications commonly utilized for evaluating previous research on CFA. These include: ***[E1]*** a Syringe Pump [5] that manages a fluid injection medical device, ***[E2]*** an Ultrasonic Ranger [8] used to measure the proximity of obstacles for parking assistance applications, and ***[E3]*** a Fire Sensor [6] implemented based on a combination of temperature and humidity measurements.

In our experiments, we incorporated a timer interrupt that operates independently from the application. This interrupt retrieves the sensor readings at a frequency of f, and stores them in a buffer. For each frequency of f = 100Hz, 1kHz, 10kHz, 30kHz, 50kHz, 70kHz, we repeated the experiment 100 times. For each test, we evaluated the performance of the three applications, measuring their runtime and energy consumption under different conditions: without any CFA (baseline), with the CFA engine only, and with both the ISM and CFA engines activated. Afterward, we calculated the overall time and energy consumption overhead incurred by the modules compared to the baseline experiment. Table III presents the results of our experiments. Consult [58] as a reference on the worst-case interrupt frequency across various real-life applications.

Runtime Overhead. The CFA engine incurs non-negligible runtime overhead primarily due to the frequent invocations of the Secure World (twice per execution of a CFG node). The overhead observed for the CFA engine is consistent with prior research on TEE-based CFA. On the other hand, the ISM overhead added atop the CFA engine is negligible when utilizing low interrupt frequencies (ranging from 10 to 1kHz). Our results show an overhead of $< 0.1\%$. The overhead becomes noticeable when the frequency reaches 10kHz, where it slowly rises from a range of 0.4% ($E3$) to 1.4% ($E1$) and reaches a maximum overhead between 5.1 % ($E2$) to 6.4% ($E1$) at around 70kHz. In sum, these results indicate that ISC-FLAT is well-suited for systems with low-frequency interrupt requirements (below 10kHz) and incurs modest overhead at higher frequencies.

Energy Consumption Overhead. The energy usage of each application is evaluated using the same configuration and

TABLE III
OVERHEAD GENERATED BY ISC-FLAT MODULES OVER THE BASELINE.

	E1 – Syringe Pump				E2 – UltrasonicRanger				E3 – Fire Sensor			
	CFA Engine		CFA Engine + ISM		CFA Engine		CFA Engine + ISM		CFA Engine		CFA Engine + ISM	
Interrupt Frequency	Runtime	Energy	Runtime	Energy	Runtime	Energy	Runtime	Energy	Runtime	Energy	Runtime	Energy
100Hz	19.3%	9.1%	19.3%	9.1%	4.5%	2.4%	4.5%	2.4%	10.2%	4.9%	10.2%	4.9%
1kHz	19.3%	9.1%	19.3%	9.1%	4.5%	2.4%	4.5%	2.4%	10.2%	4.9%	10.3%	4.9%
10kHz	19.5%	9.2%	20.9%	10.1%	4.6%	2.4%	5.7%	2.6%	10.5%	4.9%	10.9%	5.1%
30kHz	21.3%	9.7%	25.2%	12.9%	6.5%	3.6%	8.4%	3.7%	11.7%	5.4%	13.1%	6.7%
50kHz	24.8%	11.5%	29.6%	14.7%	10.3%	5.1%	16.1%	6.4%	13.0%	6.1%	18.8%	8.6%
70kHz	28.7%	14.0%	35.1%	17.5%	13.7%	6.3%	18.8%	9.0%	15.8%	7.3%	21.6%	9.9%

board as specified in Section VII-C. As shown in Table III, the relative energy consumption overhead is roughly half of the runtime overhead. These findings demonstrate the suitability of ISC-FLAT for systems with low frequency (below 10kHz) and low-power devices while still allowing an acceptable energy expenditure for a diverse range of applications on high interrupt frequency settings.

Security Tests. We design malicious interrupts to launch control flow hijacks, including the examples presented in Section 3, within the Syringe Pump execution and validate the effectiveness of ISC-FLAT by analyzing its measurements. Specifically, we analyzed the following attack vectors:

– *Attacking the return address to redirect interruptions:* our malicious interrupts attempt to redirect the interrupts by changing the IRS return address and the Dispatcher address. When it tries to modify the Dispatcher return addresses, a fault exception is generated due to illegal MPU area access. By changing the return address of the ISR, we could redirect the execution. However, $\mathcal{A}$dv is unable to resume $\mathcal{A}$pp execution. Therefore, it can not generate a signed CFA report and it runs in a different context, unrelated to $\mathcal{A}$pp.

– *Deactivating ISC-FLAT configurations:* This case attempts to misconfigure the MPU and the NS-IVT to bypass ISC-FLAT. This case always generates a fault exception due to an illegal access to a Secure World region and illegal access to MPU-protected region, respectively.

Unfortunately, there are no public/widely used benchmarks for this kind of attack yet. Thus we were unable to perform an independent analysis and had to implement our own attack vectors. Nonetheless, we hope that these attack vectors serve as a proof of concept to show ISC-FLAT security in practice. They are also included in our public release of ISC-FLAT's implementation [3].

VIII. DISCUSSION & LIMITATIONS

Memory and Performance Optimization. CFA approaches may generate large $\mathcal{CF}_{\mathsf{Log}}$-s, depending on $\mathcal{A}$pp's behavior. Instrumenting every node in $\mathcal{A}$pp can also lead to many TrustZone calls and added runtime overhead. The CFA engine in ISC-FLAT is generic and replaceable. Therefore, new techniques to optimize $\mathcal{CF}_{\mathsf{Log}}$ construction can be merged into ISC-FLAT without interfering with the interrupt-safety provided by ISM. In this work, we do not focus on optimizing the CFA engine. We refer the reader to [67] for a comprehensive treatment of optimization issues.

Non-Control Data Attacks. CFA detects and reports all control flow deviations. Nonetheless, non-control data-only attacks [25], [36] can exploit specific vulnerabilities to corrupt application data without modifying $\mathcal{A}$pp's control flow path. While these attacks require less likely vulnerabilities (e.g., "write anywhere" bugs), they are still possible. To detect such cases, future work should consider interrupt-safe attestation of both control and data flows.

Coexistence with RTOS. It is worth noting that RTOS-s rely on interrupts to control task scheduling. Thus, by isolating $\mathcal{A}$pp against interrupts, ISC-FLAT can isolate $\mathcal{A}$pp from other concurrent applications (or the RTOS) on the same device. The dispatcher would consider interrupts from the RTOS as untrusted ISRs and preserve $\mathcal{A}$pp state across them. Nonetheless, to support ISC-FLAT implementation along with an RTOS, certain compatibility challenges must be addressed. The simplest version of ISC-FLAT precludes the (Normal-World) OS from reconfiguring NS-MPU during the attested process execution. In this case, the (Normal-World) OS should request NS-MPU re-configuration to the Secure-World. Furthermore, the initialization interface must account for OS-specific process management policies.

Coexistence with Trusted OS. ISC-FLAT's proof-of-concept prototype currently runs on bare-metal. However, it could also be implemented within a Secure-World firmware or operating system, such as TF-M [7]. If implemented as a software module, ISC-FLAT would become part of the trusted firmware's or operating system's TCB, increasing its size accordingly. Alternatively, running ISC-FLAT as a trusted service or application in an unprivileged Thread mode would avoid increasing the TCB size but could potentially result in increased runtime overhead due to limited access to resource for the Dispatcher.

IX. CONCLUSION

This paper characterizes the conflict between real-time application needs and existing TEE-based CFA methods. We also demonstrate interrupt-based attacks on typical CFA designs. Motivated by this problem, we propose ISC-FLAT: a TEE-based interrupt-safe CFA scheme. ISC-FLAT protects the integrity of interrupted attested programs against vulnerable/malicious interrupts. It tracks and controls interrupt access within the TEE's secure world to ensure that interrupt service routines cannot tamper with the integrity of CFA reports. We implement and evaluate ISC-FLAT and make its implementation publicly available.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and designated shepherd for their guidance and feedback. This work was supported by the National Science Foundation (Award #2245531) as well as a Meta Research Award (2022 Towards Trustworthy Products in AR, VR, and Smart Devices RFP).

REFERENCES

[1] "Blake2s Implementation," github.com/mjosaarinen/blake2_mjosref/blob/master/blake2s.c.

[2] "CFA Interrupt Attack Example," https://github.com/RIT-CHAOS-SEC/ISC-FLAT_open_source/examples/attacks.

[3] "ISC-FLAT Prototype Implementation," https://github.com/RIT-CHAOS-SEC/ISC-FLAT_open_source.

[4] "NUCLEO-L552ZE-Q board," https://www.st.com/en/microcontrollers-microprocessors/stm32l552ze.html.

[5] "Open Syringe Pump," https://github.com/naroom/OpenSyringePump.

[6] "Temperature and Humidity Sensor," https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/temp_humi_sensor.

[7] "TrustedFirmware-M (TF-M)," https://www.trustedfirmware.org/projects/tf-m/.

[8] "Ultrasonic Ranger," https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/ultrasonic_ranger.

[9] "X-NUCLEO-LPM01A board," https://www.st.com/en/evaluation-tools/x-nucleo-lpm01a.html.

[10] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity Principles, Implementations, and Applications," *Proceedings of ACM Transactions on Information and System Security*, vol. 13, no. 1, pp. 1–40, 2009.

[11] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: Control-Flow Attestation for Embedded Systems Software," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 743–754.

[12] J. Ahn, I.-G. Lee, and M. Kim, "Design and Implementation of Hardware-based Remote Attestation for a Secure Internet of Things," *Wireless Personal Communications*, vol. 114, no. 1, pp. 295–327, 2020.

[13] M. Ambrosin, M. Conti, R. Lazzeretti, M. M. Rabbani, and S. Ranise, "PADS: Practical Attestation for Highly Dynamic Swarm Topologies," in *Proceedings of IEEE International Workshop on Secure Internet of Things*, 2018, pp. 18–27.

[14] M. Ammar, B. Crispo, and G. Tsudik, "SIMPLE: A Remote Attestation Approach for Resource-constrained IoT devices," in *Proceedings of ACM/IEEE International Conference on Cyber-Physical Systems*, 2020, pp. 247–258.

[15] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the Mirai Botnet," in *Proceedings of USENIX Security Symposium*, 2017, pp. 1093–1110.

[16] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann, "Seda: Scalable Embedded Device Attestation," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 964–975.

[17] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "Tytan: Tiny trust anchor for tiny devices," in *Proceedings of ACM/IEEE Design Automation Conference*, 2015, pp. 1–6.

[18] F. Brasser, A.-R. Sadeghi, and G. Tsudik, "Remote Attestation for Low-End Embedded Devices: the Prover's Perspective," in *Proceedings of ACM/IEEE Design Automation Conference*, 2016.

[19] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens, "Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors," in *Proceedings of IEEE Computer Security Foundations Symposium*, 2020, pp. 262–276.

[20] X. Carpent, K. ElDefrawy, N. Rattanavipanon, and G. Tsudik, "Lightweight Swarm Attestation: a Tale of Two Lisa-s," in *Proceedings of ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 86–100.

[21] X. Carpent, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "Temporal Consistency of Integrity-ensuring Computations and Applications to Embedded Systems Security," in *Proceedings of ACM Asia Conference on Computer and Communications Security*, 2018, pp. 313–327.

[22] X. Carpent, N. Rattanavipanon, and G. Tsudik, "ERASMUS: Efficient Remote Attestation via Self-Measurement for Unattended Settings," in *Proceedings of Design, Automation and Test in Europe Conference*, 2018.

[23] ——, "Remote Attestation of IoT devices via SMARM: Shuffled Measurements Against Roving Malware," in *Proceedings of IEEE International Symposium on Hardware Oriented Security and Trust*, 2018, pp. 9–16.

[24] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented Programming Without Returns," in *Proceedings of ACM Conference on Computer and Communications Security*, 2010, pp. 559–572.

[25] S. Chen, J. Xu, and E. C. Sezer, "Non-Control-Data Attacks Are Realistic Threats," in *Proceedings of USENIX Security Symposium*, Jul. 2005.

[26] Y.-G. Choi, J. Kang, and D. Nyang, "Proactive Code Verification Protocol in Wireless Sensor Network," in *Proceedings of International Conference on Computational Science and Its Applications*. Springer, 2007, pp. 1085–1096.

[27] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the TOCTOU Problem in Remote Attestation," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2921–2936.

[28] I. De Oliveria Nunes, S. Jakkamsetti, and G. Tsudik, "Tiny-CFA: A Minimalistic Approach for Control Flow Attestation Using Verified Proofs of Execution," 2021.

[29] J. Deogirikar and A. Vidhate, "Security attacks in IoT: A survey," in *Proceedings of IEEE International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)*, 2017, pp. 32–37.

[30] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: Lightweight Hardware-assisted Attestation of Program Execution," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, 2018, pp. 1–8.

[31] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "Lo-fat: Low-overhead Control Flow Attestation in Hardware," in *Proceedings of ACM/IEEE Design Automation Conference*, 2017, pp. 1–6.

[32] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust," in *Proceedings of Network and Distributed System Security Symposium*, vol. 12, 2012, pp. 1–15.

[33] K. Eldefrawy et al., "HYDRA: Hybrid Design for Remote Attestation (Using a Formally Verified Microkernel)," in *Proceedings of ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2017.

[34] J. Giraldo, A. Cárdenas, and N. Quijano, "Integrity Attacks on Real-time Pricing in Smart Grids: Impact and Countermeasures," *IEEE Transactions on Smart Grid*, vol. 8, no. 5, pp. 2249–2257, 2016.

[35] M. Grisafi, M. Ammar, M. Roveri, and B. Crispo, "PISTIS: Trusted Computing Architecture for Low-end Embedded Systems," in *Proceedings of USENIX Security Symposium*, 2022.

[36] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *Proceedings of IEEE Symposium on Security and Privacy*, 2016, pp. 969–986.

[37] A. Ibrahim, A.-R. Sadeghi, and S. Zeitouni, "SeED: Aecure Non-interactive Attestation for Embedded Devices," in *Proceedings of ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2017.

[38] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A Security Architecture for Tiny Embedded Devices," in *Proceedings of ACM European Conference on Computer Systems*, 2014.

[39] B. Kuang, A. Fu, W. Susilo, S. Yu, and Y. Gao, "A Survey of Remote Attestation in Internet of Things: Attacks, Countermeasures, and Prospects," *Computers & Security*, vol. 112, p. 102498, 2022.

[40] M. Kucab, P. Boryło, and P. Chołda, "Remote Attestation and Integrity Measurements with Intel SGX for Virtual Machines," *Computers & Security*, vol. 106, p. 102300, 2021.

[41] R. Langner, "Stuxnet: Dissecting a Cyberwarfare Weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[42] Y. Li, J. M. McCune, and A. Perrig, "VIPER: Verifying the Integrity of PERipherals' Firmware," in *Proceedings of ACM Conference on Computer and Communications Security*, 2011.

[43] J. Liu, Q. Yu, W. Liu, S. Zhao, D. Feng, and W. Luo, "Log-based Control Flow Attestation for Embedded Devices," in *Proceedings of International Symposium of Cyberspace Safety and Security*. Springer, 2019, pp. 117–132.

[44] L. Luo, X. Shao, Z. Ling, H. Yan, Y. Wei, and X. Fu, "fASLR: Function-Based ASLR via TrustZone-M and MPU for Resource-Constrained IoT Systems," *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 17 120–17 135, 2022.

[45] I. D. O. Nunes, G. Dessouky, A. Ibrahim, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "Towards Systematic Design of Collective Remote Attestation Protocols," in *Proceedings of IEEE International Conference on Distributed Computing Systems*, 2019, pp. 1188–1198.

[46] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A Verified Hardware/Software Co-Design for Remote Attestation," in *Proceedings of USENIX Security Symposium*, 2019, pp. 1429–1446.

[47] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise," in *Proceedings of USENIX Security Symposium*, 2020, pp. 771–788.

[48] I. D. O. Nunes, S. Jakkamsetti, and G. Tsudik, "Dialed: Data Integrity Attestation for Low-end Embedded Devices," in *Proceedings of ACM/IEEE Design Automation Conference*, 2021, pp. 313–318.

[49] D. Oliveira, T. Gomes, and S. Pinto, "uTango: an Open-source TEE for the Internet of Things," *arXiv preprint arXiv:2102.03625*, 2021.

[50] R. Pan and G. Parmer, "SBIs: Application Access to Safe, Baremetal Interrupt Latencies," in *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium*, 2022, pp. 82–94.

[51] D. Perito and G. Tsudik, "Secure Code Update for Embedded Devices via Proofs of Secure Erasure," in *Proceedings of European Symposium on Research in Computer Security*. Springer, 2010, pp. 643–662.

[52] L. Petzi, A. E. B. Yahya, A. Dmitrienko, G. Tsudik, T. Prantl, and S. Kounev, "{SCRAPS}: Scalable Collective Remote Attestation for {Pub-Sub}{IoT} Networks with Untrusted Proxy Verifier," in *Proceedings of USENIX Security Symposium*, 2022, pp. 3485–3501.

[53] S. Pinto and C. Garlati, "User Mode Interrupts: A must for Securing Embedded Systems," in *Proceedings of the Embedded World Conference*, 2019.

[54] S. Pinto and N. Santos, "Demystifying Arm Trustzone: A Comprehensive Survey," *ACM computing surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[55] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on Trustzone-enabled Microcontrollers? voilà!" in *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium*, 2019, pp. 293–304.

[56] L. Pycroft and T. Z. Aziz, "Security of Implantable Medical Devices with Wireless Connections: The Dangers of Cyber-Attacks," *Expert Review of Medical Devices*, vol. 15, no. 6, pp. 403–406, 2018.

[57] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper Resistance Mechanisms for Secure Embedded Systems," in *Proceedings of IEEE International Conference on VLSI Design*, 2004, pp. 605–611.

[58] J. Regehr and U. Duongsaa, "Preventing interrupt overload," *ACM SIGPLAN Notices*, vol. 40, no. 7, pp. 50–58, 2005.

[59] T. Ring, "Connected Cars - The Next Target for Hackers," *Network Security*, vol. 2015, no. 11, pp. 11–16, 2015.

[60] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented Programming: Systems, Languages, and Applications," *Proceedings of ACM Transactions on Information and System Security*, vol. 15, no. 1, pp. 1–34, 2012.

[61] A. Seshadri, M. Luk, A. Perrig, L. Van Doorn, and P. Khosla, "SCUBA: Secure Code Update by Attestation in sensor networks," in *Proceedings of ACM Workshop on Wireless Security*, 2006, pp. 85–94.

[62] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *Proceedings of IEEE Symposium on Research in Security and Privacy*, 2004, pp. 272–282.

[63] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proceedings of ACM Conference on Computer and Communications Security*, 2007, pp. 552–561.

[64] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of IEEE Symposium on Security and Privacy*, 2016.

[65] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization," in *Proceedings of IEEE Symposium on Security and Privacy*, 2013, pp. 574–588.

[66] R. V. Steiner and E. Lupu, "Towards More Practical Software-based Attestation," *Computer networks*, vol. 149, pp. 43–55, 2019.

[67] Z. Sun, B. Feng, L. Lu, and S. Jha, "OAT: Attesting Operation Integrity of Embedded Devices," in *Proceedings of IEEE Symposium on Security and Privacy*, 2020, pp. 1433–1449.

[68] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal War in Memory," in *Proceedings of IEEE Symposium on Security and Privacy*, 2013, pp. 48–62.

[69] F. Toffalini, E. Losiouk, A. Biondo, J. Zhou, and M. Conti, "{ScaRR}: Scalable Runtime Remote Attestation for Complex Systems," in *Proceedings of International Symposium on Research in Attacks, Intrusions and Defenses*, 2019, pp. 121–134.

[70] F. Vahid and T. D. Givargis, *Embedded System Design: a Unified Hardware/Software Introduction*. John Wiley & Sons, 2001.

[71] 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 *Proceedings of IEEE Symposium on Security and Privacy*, 2022, pp. 1573–1573.

[72] J. Wang, Z. Hong, Y. Zhang, and Y. Jin, "Enabling Security-enhanced Attestation with Intel SGX for Remote Terminal and IoT," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 88–96, 2017.

[73] W. Xu, X. Zhang, H. Hu, G.-J. Ahn, and J.-P. Seifert, "Remote Attestation with Domain-based Integrity Model and Policy Analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 3, pp. 429–442, 2011.

[74] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi, "Atrium: Runtime Attestation Resilient Under Memory Attacks," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2017, pp. 384–391.

[75] Y. Zhang, X. Liu, C. Sun, D. Zeng, G. Tan, X. Kan, and S. Ma, "ReCFA: Resilient Control-Flow Attestation," in *Proceedings of Computer Security Applications Conference*, 2021, pp. 311–322.