ACFA: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation

Adam Caulfield, *Rochester Institute of Technology;* Norrathep Rattanavipanon, *Prince of Songkla University, Phuket Campus;* Ivan De Oliveira Nunes, *Rochester Institute of Technology*

https://www.usenix.org/conference/usenixsecurity23/presentation/caulfield

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

ACFA: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation

Adam Caulfield
Rochester Institute of Technology

Norrathep Rattanavipanon
Prince of Songkla University, Phuket Campus

Ivan De Oliveira Nunes
Rochester Institute of Technology

Abstract

Embedded devices are increasingly used in a wide range of "smart" applications and spaces. At the lower-end of the scale, they are implemented under strict cost and energy budgets, using microcontroller units (MCUs) that lack security features akin to those available in general-purpose processors. In this context, Remote Attestation (*RA*) was proposed as an inexpensive security service that enables a verifier ($\mathcal{V}$rf) to remotely detect illegal modifications to the software binary installed on a prover MCU ($\mathcal{P}$rv). Despite its effectiveness to validate $\mathcal{P}$rv's binary integrity, attacks that hijack the software's control flow (potentially leading to privilege escalation or code reuse attacks) cannot be detected by classic *RA*.

Control Flow Attestation (*CFA*) augments *RA* with information about the exact order in which instructions in the binary are executed. As such, *CFA* enables detection of the aforementioned control flow attacks. However, we observe that current *CFA* architectures cannot guarantee that $\mathcal{V}$rf ever receives control flow reports in case of attacks. In turn, while they support detection of exploits, they provide no means to pinpoint the exploit origin. Furthermore, existing CFA requires either (1) binary instrumentation, incurring significant runtime overhead and code size increase; or (2) relatively expensive hardware support, such as hash engines. In addition, current techniques are neither continuous (they are only meant to attest small and self-contained operations) nor active (once compromises are detected, they offer no secure means to remotely remediate the problem).

To jointly address these challenges, we propose *ACFA*: a hybrid (hardware/software) architecture for <u>A</u>ctive <u>CFA</u>. *ACFA* enables continuous monitoring of all control flow transfers in the MCU and does not require binary instrumentation. It also leverages the recently proposed concept of "active roots-of-trust" to enable secure auditing of vulnerability sources and guaranteed remediation, in case of compromise detection. We provide an open-source reference implementation of *ACFA* on top of a commodity low-end MCU (TI MSP430) and evaluate it to demonstrate its security and cost-effectiveness.

1 Introduction

Embedded devices are crucial components of modern systems. A large portion of these devices are implemented using low-end and bare-metal microcontroller units (MCUs), specifically designed for energy, cost, and spatial efficiency. They are well-suited for and commonly used in safety-critical sensor-based applications such as medical devices, vehicular sensors/actuators, and sensor/alarm systems. Due to cost/energy budgets, MCUs often lack common hardware features used to secure higher-end systems (e.g., memory management units, strong privilege separation, and inter-process isolation). Unsurprisingly, the absence of these features makes them attractive targets to a wide range of software attacks [22, 31, 61].

In this context, Remote Attestation (*RA*) [5, 9, 18, 20, 26–28, 33, 37, 40, 44, 45, 62], as well Proofs of Execution (*PoX*) [13, 19], have been proposed as means for a verifier ($\mathcal{V}$rf) to ascertain the software state of a remote prover MCU ($\mathcal{P}$rv). While these techniques can prove software integrity and its execution on $\mathcal{P}$rv, they cannot detect runtime attacks that tamper with the program's control flow without modifying its code. For instance, an adversary ($\mathcal{A}$dv) can leverage a buffer overflow vulnerability to hijack the program's control flow by overwriting the return address of the executing function. This vulnerability can in turn be used to jump to an arbitrary instruction within the binary, potentially skipping security checks or launching Return-Oriented Programming (ROP) [48] attacks.

Control Flow Integrity (CFI) methods [6, 17, 29] aim to address these vulnerabilities by proactively checking the program's control flow at runtime, locally at $\mathcal{P}$rv. However, CFI methods typically rely on hardware features and/or computational requirements (e.g., instrumentation, storage for large control flow graphs, and/or shadow stacks [58]) that are prohibitively expensive for MCUs [2]. In addition, the general problem of enumerating valid and invalid control flow paths is often intractable [14, 46].

Due to the challenges associated with CFI, Control Flow Attestation (*CFA*) was proposed in C-FLAT [2]. The key idea in *CFA* is to outsource the detection of control flow violations to the computationally resourceful $\mathcal{V}$rf (e.g., a back-end server). To support this remote verification, $\mathcal{P}$rv builds an authenticated log containing all control flow transfers, i.e., the source and destination of all branching instructions (e.g `jumps`, `returns`, `calls`, etc.) within the execution of a given operation. This log is obtained by either (1) instrumenting each branching instruction with additional instructions to securely save branch destinations in protected memory [2, 21, 56]; or (2) using customized hardware to detect branches and save their destinations [23, 24, 65]. The produced "control flow log ($\mathcal{CF}_{Log}$)" is authenticated – usually MAC-ed or signed by a Root-of-Trust (RoT) in $\mathcal{P}$rv– and sent to $\mathcal{V}$rf along with an *RA* report. In possession of both the attested binary and the log of all control flow transfers, $\mathcal{V}$rf can check if the reported control flow path is valid and is even able to emulate the reported execution (if data inputs are provided, as in [41]).

1.1 *CFA* Limitations: Auditing & Healing

Following C-FLAT [2], additional *CFA* designs were presented [21, 23, 24, 56, 65] under various assumptions and guarantees. Despite substantial progress, current *CFA* techniques share several limitations. Due to their passive nature, they offer no guarantee that a $\mathcal{CF}_{Log}$ is ever received by $\mathcal{V}$rf in case of $\mathcal{P}$rv compromise. While this suffices to detect compromises (in general, absence of an *RA* report indicates that something is wrong), it precludes *auditing* $\mathcal{CF}_{Log}$ to pinpoint the source of compromises (i.e., to determine what is wrong). The latter is non-trivial to obtain, since a compromised $\mathcal{P}$rv might ignore the protocol and simply refuse to send back reports that indicate a compromise. Furthermore, current techniques cannot guarantee $\mathcal{P}$rv's remediation when a compromise is detected.

In addition, current techniques manage $\mathcal{CF}_{Log}$ in ways that introduce non-trivial challenges. A typical approach is to compute an in-order hash-chain of all entries in $\mathcal{CF}_{Log}$. While this reduces the required storage (only the latest hash value needs to be maintained by $\mathcal{P}$rv), it requires $\mathcal{V}$rf to have *a priori* knowledge of all valid control flow paths. Without this knowledge, $\mathcal{V}$rf cannot compute the correct hash result during the verification of *CFA* report. Similar to the CFI case, determining all valid control paths is non-trivial and often infeasible. An alternative approach is to store $\mathcal{CF}_{Log}$ in its entirety and send it verbatim to $\mathcal{V}$rf [56], once the attested execution is over. This eases the verification process. However, as $\mathcal{CF}_{Log}$ grows rapidly, it can quickly fill up $\mathcal{P}$rv's limited memory. Due to this limitation, some *CFA* techniques (e.g., OAT [56] and Tiny-CFA [21]) are only envisioned for small and self-contained operations. In SCaRR [59], this limitation is resolved by requiring $\mathcal{P}$rv to transmit a series of intermediate logs of reduced size, rather than the entire $\mathcal{CF}_{Log}$. This allows continuous verification of the program's control flow using a series of fine-grained reports. However, since SCaRR was designed for high-end cloud systems, its applicability for low-end MCUs remains unclear.

Finally, existing *CFA* architectures either (1) rely on code instrumentation, resulting in substantial runtime and binary size overhead; or (2) rely fully on hardware features that are prohibitively expensive to low-end MCUs.

1.2 Contributions: Efficient Control Flow Auditing & Active Compromise Remediation

This paper proposes *ACFA*: an Active Control Flow Attestation architecture. *ACFA* addresses aforementioned limitations by composing concepts from *CFA* and Active RoTs (see Section 2.4) to guarantee that $\mathcal{V}$rf always receives $\mathcal{CF}_{Log}$ and is able to remotely remediate $\mathcal{P}$rv's state in case of compromise detection. *ACFA* architecture is implemented as an inexpensive and open-source hardware/software co-design. In sum, *ACFA* anticipated contributions are threefold:

- We propose *ACFA*, the first architecture to guarantee $\mathcal{V}$rf eventually receives *CFA* reports ($\mathcal{CF}_{Log}$-s) containing $\mathcal{P}$rv's execution trace. *ACFA* also supports guaranteed healing of $\mathcal{P}$rv when a compromise is detected. These features are obtained through a synergic combination of Active RoTs, *CFA*, and novel *CFA*-specific non-maskable interrupts, realized as a hybrid (HW/SW) design.
- While prior hybrid approaches exist in *RA*, current *CFA* techniques relied either on customized (and relatively expensive) hardware or on software instrumentation. We present the first hybrid design for *CFA* that eliminates any software instrumentation requirements and minimizes hardware cost, making it affordable even to simple MCUs. *ACFA* also demonstrates the feasibility of secure $\mathcal{CF}_{Log}$ slicing (introduced by SCaRR [59]) in MCUs and leverages this feature to support fine-grained control flow auditing of arbitrarily sized software operations.
- We propose a continuous *ACFA* protocol aimed at on-demand sensing/actuation use cases. The protocol integrates *ACFA* with a typical on-demand MCU application: MCU awaits for command(s) $\rightarrow$ performs action(s) $\rightarrow$ reports result(s) $\rightarrow$ returns to idle/waiting state. We provide open-source end-to-end implementations and demonstrative videos of such use-cases (including *ACFA* implementation) on an FPGA-based deployment in [12].

2 Background

2.1 Scope

This work focuses on simple MCUs and aims for minimality of hardware requirements. We argue that a design that is cost-effective enough for the lowest-end MCUs could also be adapted and potentially enriched for higher-end devices, with less strict hardware budgets (we discuss alternative designs in Section 5). Adapting designs in the other direction is more challenging. The choice of a simple device also facilitates reasoning and presenting *ACFA* concepts systematically.

Following these premises, we present a design for low-end MCUs based on low-power single-core platforms with only a few kilobytes (KB) of program and data memory (such as Atmel AVR ATmega and TI MSP430). They feature 8- and 16-bit CPUs, typically running at 1-16 MHz clock frequencies, with $\approx$ 64 KB of addressable memory. SRAM is used as data memory (*DMEM*) ranging in size between 4 and 16 KB, while the rest of the address space is available for program memory (*PMEM*). They run software at "bare metal", executing instructions in place (physically from *PMEM*), and have no memory management unit (MMU) to support virtual memory.

ACFA prototype is implemented atop a representative of this class of devices: the well-known TI MSP430 ultra low-energy MCU. This choice is simply due to the availability of an open-source version of the MSP430 hardware from OpenCores [32]. Nevertheless, we expect *ACFA* design to generalize to other bare-metal MCUs (e.g., ARM Cortex-M). See Section 5 for future work discussion on adapting *ACFA* to higher-end devices (e.g., those featuring virtual memory).

2.2 Remote Attestation (*RA*)

RA is a challenge-response protocol between $\mathcal{V}\mathsf{rf}$ and $\mathcal{P}\mathsf{rv}$. It allows $\mathcal{V}\mathsf{rf}$ to remotely assess $\mathcal{P}\mathsf{rv}$'s trustworthiness by measuring the content of $\mathcal{P}\mathsf{rv}$'s memory. As depicted in Figure 1, a typical *RA* interaction involves the following steps:

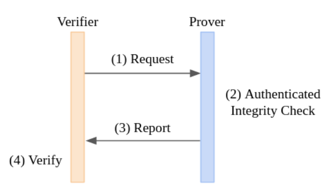


Figure 1: A typical *RA* interaction

1. $\mathcal{V}\mathsf{rf}$ requests *RA* from $\mathcal{P}\mathsf{rv}$ by sending a cryptographic challenge *Chal*.
2. Upon receiving *Chal*, $\mathcal{P}\mathsf{rv}$ computes an authenticated integrity-ensuring function over its own memory and *Chal*, producing report *H*.
3. $\mathcal{P}\mathsf{rv}$ sends the report *H* back to $\mathcal{V}\mathsf{rf}$.
4. $\mathcal{V}\mathsf{rf}$ checks *H* against an expected value to determine if $\mathcal{P}\mathsf{rv}$ has been compromised.

The authenticated integrity-ensuring function in step 2 is implemented using a message authentication code (MAC) or a digital signature. The secret key used in this operation must be securely stored to ensure that it is inaccessible to any untrusted software on $\mathcal{P}\mathsf{rv}$. *RA* threat models (including the one considered in this paper) assume that $\mathcal{P}\mathsf{rv}$ is susceptible to full software compromise. Therefore, secure storage for the *RA* secret key implies some level of hardware support.

RA architectures are generally classified in three types: software-based (a.k.a. "keyless"), hardware-based, or hybrid. Software-based architectures [36, 52–54] require no hardware support. However, *RA* must be local (e.g., over a one-hop wired communication) and requires several strong assumptions about $\mathcal{A}\mathsf{dv}$ capabilities, implementation optimality, and fixed communication delays that are often infeasible in practice [11]. Hardware-based architectures [38, 43, 49] rely on standalone cryptographic coprocessors (e.g., TPMs [60]) or complex support from the CPU instruction set architecture (e.g., Intel SGX [35]). Although these approaches provide strong security guarantees for *RA*, their hardware cost is often too expensive and unrealistic for MCUs. Hybrid architectures [18, 26, 28] focus on low-cost MCUs. They leverage minimal hardware support to store cryptographic secret(s) and to support secure execution of a software implementation of the integrity-ensuring function (MAC or signature) computed during the *RA* protocol. Hybrid architectures aim to combine the low hardware cost of software-based approaches with the security guarantees offered by hardware-based approaches.

VRASED [18] is a formally verified hybrid *RA* architecture. It implements the authenticated integrity-ensuring function in software while introducing small trusted hardware to enforce the correct execution of this software and confidentiality of the *RA* secret key. In addition, VRASED guarantees that the attested memory is temporally consistent, i.e., not modifiable during the memory measurement. We further elaborate on *RA* related work in Section 7. As we discuss in Sections 3 and 4, *ACFA* hybrid design leverages VRASED to replace relatively costly hardware-based hash engines and to authenticate *CFA* reports.

VRASED also provides an optional design extension that supports authentication of $\mathcal{V}\mathsf{rf}$ requests. In this case, an authentication token accompanies $\mathcal{V}\mathsf{rf}$ requests. $\mathcal{V}\mathsf{rf}$ computes this token as a MAC over *Chal*, using the *RA* key. To prevent replays, *Chal* must be a monotonically increasing counter, and the latest *Chal* used to successfully authenticate $\mathcal{V}\mathsf{rf}$ must be stored in $\mathcal{P}\mathsf{rv}$'s persistent and protected memory. In each *RA* request, incoming *Chal* must be greater than the stored value. Once an *RA* request is successfully authenticated, the stored value is updated accordingly. As discussed later in Section 4, *ACFA* also uses this VRASED extension to authenticate remediation decisions made by $\mathcal{V}\mathsf{rf}$.

2.3 Control Flow Attestation (*CFA*)

In addition to the *RA* result, *CFA* also provides $\mathcal{V}\mathsf{rf}$ with an authenticated $\mathcal{CF}_{Log}$ that contains the order in which the instructions in the attested binary were executed. $\mathcal{CF}_{Log}$ is either produced by dedicated hardware or obtained by instrumenting each branching instruction with additional instructions to securely save their source and destination addresses in protected memory. Once the execution of the attested operation completes, $\mathcal{CF}_{Log}$ is authenticated (usually MAC-ed or signed by the *RA* RoT in $\mathcal{P}$rv) and reported to $\mathcal{V}$rf along with the *RA* result. In possession of both the attested binary and $\mathcal{CF}_{Log}$, $\mathcal{V}$rf can decide if the reported control flow path is valid, and thus if $\mathcal{P}$rv has been compromised.

Prior *CFA* designs (and *RA*/*PoX* architectures, more broadly) consider absence of a valid report (step (3) in Figure 1) as a sign that $\mathcal{P}$rv is compromised, as the honest $\mathcal{P}$rv would have followed the protocol. Such an assumption is sensible from a *detection* perspective. However, it prevents $\mathcal{V}$rf from *securely auditing* the source of an exploit – the control flow violation leading to the exploit may never be received by $\mathcal{V}$rf; thus $\mathcal{V}$rf cannot easily pinpoint the vulnerability. One of *ACFA* core contributions is to enable *secure runtime auditing*, i.e., guaranteeing delivery of $\mathcal{CF}_{Log}$, even when $\mathcal{P}$rv is compromised by malware that prevents $\mathcal{P}$rv from sending *CFA* reports to $\mathcal{V}$rf. Naturally, this guarantee holds when $\mathcal{A}$dv is unable to jam the network indefinitely. In the case where $\mathcal{A}$dv has such capability and $\mathcal{V}$rf never receives reports, *ACFA* can optionally halt execution on $\mathcal{P}$rv.

C-FLAT [2] was the first proposal for *CFA*. It uses ARM TrustZone's secure world as an RoT to build and store $\mathcal{CF}_{Log}$. In a pre-processing phase, a control flow graph (CFG) is constructed and each node in the CFG is assigned a unique Node ID. The executable is instrumented with secure-monitor calls to TrustZone secure world to save Node IDs whenever a node transition occurs. C-FLAT measurement engine, implemented within the secure world, extends a hash-chain with the Node ID on each call. Once execution of the attested task completes, the hash-chain uniquely identifies the control flow path. Subsequent *CFA* architectures [21,23,24,56,59,65,66] built upon C-FLAT (see Section 7). Despite substantial progress, to the best of our knowledge, the problems identified in Section 1 remain common to all of these *CFA* architectures.

2.4 Active RoTs

Classic attestation methods (including *RA* and *CFA*) have been designed as passive RoTs. As such, they can detect compromises to $\mathcal{P}$rv integrity. However, they cannot guarantee actions will be taken beyond detection. Recently proposed *active* RoTs [3,4,34,39,64], on the other hand, focus on availability under software compromise. In particular, GAROTA [4] is a generalized interrupt-based active RoT designed as a hardware monitor for low-end MCUs. It supports a secure association between a trigger event (e.g., "temperature exceeds a threshold") and the correct execution of a software function responsible for a safety-critical action (e.g., "sound the alarm"), whenever the trigger event occurs. This guaranteed trigger-action association must hold even when the MCU software is compromised.

To achieve this goal, GAROTA provides two core features: *guaranteed triggering* and *re-triggering on failure*. *Guaranteed triggering* ensures that a predefined trusted software function ($\mathcal{F}$) always takes over execution when a corresponding safety-critical interrupt of interest – the trigger – occurs. After the trigger, $\mathcal{F}$ execution cannot be tampered with or interrupted until its completion (i.e., reaching its pre-defined exit instruction). Any attempt to interfere with $\mathcal{F}$ execution causes an immediate MCU reset. The reset brings the MCU back to a clean state where interrupts and Direct Memory Access (DMA) controllers are disabled. Immediately after any reset, the *re-triggering on failure* property ensures that $\mathcal{F}$ is always the first software to execute. Therefore, malware on $\mathcal{P}$rv is unable to prevent $\mathcal{F}$ from executing in its entirety once a trigger has occurred. At best, malware can cause a reset by attempting to interrupt $\mathcal{F}$. The reset will, in turn, lead to a secure re-execution of $\mathcal{F}$ with interrupts and DMA disabled.

GAROTA is a general architecture that supports any preexistent interrupt source to be configured as a trigger, including GPIO inputs (i.e., external inputs from sensors, buttons, etc.), timers, and (UART-based) network events. As $\mathcal{F}$ is a software function, it can implement any desired action that should take place following the trigger event. To obtain this secure trigger-action association, GAROTA hardware monitors execution and protects the initial configuration of the trigger interrupt from illegal modifications or disablement. This protection includes preserving interrupt configuration registers, interrupt handlers, and the interrupt vector table. This way, GAROTA guarantees that a trigger always results in an invocation of $\mathcal{F}$. However, guaranteed invocation of $\mathcal{F}$ upon occurrence of a trigger is not sufficient to claim that $\mathcal{F}$ is properly performed, since the $\mathcal{F}$ code (and execution thereof) could be tampered with. To this end, GAROTA hardware also provides runtime protections that prevent any unprivileged/untrusted program from modifying $\mathcal{F}$ code. GAROTA monitors the execution of $\mathcal{F}$ to ensure:

1. **Atomicity:** $\mathcal{F}$ executes uninterrupted, from its first instruction (legal entry), to its last instruction (legal exit);
2. **Non-malleability:** *PMEM* region storing $\mathcal{F}$ implementation is unmodifiable at runtime. During $\mathcal{F}$ execution, *DMEM* can only be modified by $\mathcal{F}$ itself, e.g., no modifications by DMA controllers.

These properties ensure that any malware potentially residing on the MCU (i.e., compromised software outside $\mathcal{F}$ or compromised DMA controllers) cannot tamper with $\mathcal{F}$ execution.

ACFA builds atop active RoT concepts as one of its features to guarantee secure control flow auditing and device healing. Unlike GAROTA, *ACFA* creates a new trigger, based on *CFA*-specific events, implemented as a non-maskable interrupt that is controlled only by *ACFA*, in hardware. The associated $\mathcal{F}$ in *ACFA* implements a sequence of actions to guarantee that $\mathcal{CF}_{Log}$ is always received by $\mathcal{V}$rf and that a $\mathcal{V}$rf-initiated remediation function is properly invoked, when applicable.

3 *ACFA* High Level Overview

This section presents *ACFA* high level ideas, before going into its details in Section 4. To construct $\mathcal{CF}_{Log}$, *ACFA* implements a hardware *CFA* monitor that detects and saves all control flow transfers that happen during the attested execution to a fixed dedicated *DMEM* region. The monitor also ensures that this region is read-only to all software. Therefore, compromised $\mathcal{P}rv$ software is unable to modify $\mathcal{CF}_{Log}$. When reporting the *CFA* result (including both $\mathcal{P}rv$ binary and $\mathcal{CF}_{Log}$) to $\mathcal{V}rf$, *ACFA* offers the following key features:

[F1] Secure Control Flow Auditing: it guarantees that any $\mathcal{CF}_{Log}$ (or partial $\mathcal{CF}_{Log}$, when $\mathcal{CF}_{Log}$ is sliced and streamed due to limited storage) generated by the *CFA* hardware monitor must be received, successfully authenticated, and accepted by $\mathcal{V}rf$. The active *CFA* RoT in $\mathcal{P}rv$ assures that execution remains paused until a confirmation of receipt from $\mathcal{V}rf$ reaches $\mathcal{P}rv$. In the interim, the report can be periodically re-transmitted to $\mathcal{V}rf$ to cope with occasional network losses. Optionally, if an (application-specific) upper bound on the wait time is reached without receiving $\mathcal{V}rf$ confirmation, $\mathcal{P}rv$ can automatically switch to the remediation phase (see below) or resume execution, depending on the desired policy (strict vs. best-effort). Our discussion focuses on a strict version, where software integrity is more important than minimizing disruption. In this case, $\mathcal{P}rv$ must always wait for $\mathcal{V}rf$ confirmation. Thus, $\mathcal{P}rv$ execution halts if messages are discarded indefinitely. We revisit alternative designs for the best-effort case in Section 5.

[F2] Guaranteed Remediation: as a part of its confirmation, $\mathcal{V}rf$ can indicate whether $\mathcal{P}rv$ execution is allowed to proceed normally, i.e., when the *CFA* verification indicates a benign and expected state. In case of compromise detection, $\mathcal{V}rf$ can indicate that $\mathcal{P}rv$ must switch to the remediation phase. *ACFA* ensures that $\mathcal{V}rf$ command is processed, irrespective of a compromised software state on $\mathcal{P}rv$. The specific remediation action is configurable, depending on the desired policy for each particular application domain. For instance, it might include remotely updating the binary in *PMEM*, erasing all memory, or shutting $\mathcal{P}rv$ down.

At its core, *ACFA* implements an Active RoT (recall Section 2.4) with a trigger used to take over $\mathcal{P}rv$ execution whenever $\mathcal{CF}_{Log}$ (or a slice of $\mathcal{CF}_{Log}$, if $\mathcal{CF}_{Log}$ sliced and streamed) must be sent to $\mathcal{V}rf$. To that end, *ACFA* hardware monitor implements a new secure interrupt occurring in three cases (whichever comes first):

[T1] when a timer expires, imposing periodic reports to $\mathcal{V}rf$;

[T2] when the $\mathcal{CF}_{Log}$ designated memory is full, implying that its contents must be received by $\mathcal{V}rf$ and flushed before new control flow transfers can be stored;

[T3] when $\mathcal{P}rv$ resets/boots or when the attested operation concludes its execution.

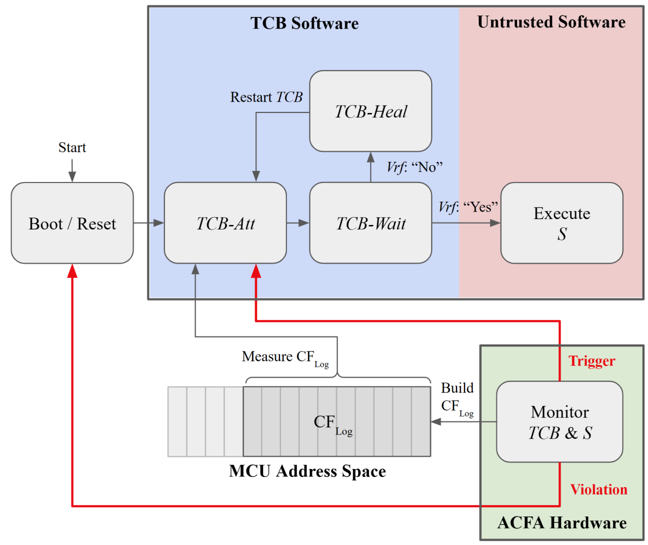


Figure 2: *ACFA* Execution Workflow

We note that trigger case **[T2]** implies that whenever $\mathcal{P}rv$ runs out of dedicated memory to store $\mathcal{CF}_{Log}$, the partial snapshot of the control flow transfers in $\mathcal{CF}_{Log}$ is automatically authenticated and transmitted to $\mathcal{V}rf$ for verification. After this step, the same memory region can be re-used to store subsequent control flow transfers.

In *ACFA*, the associated trigger-handling function $\mathcal{F}$ is referred to as Trusted Computing Base (*TCB*) Software. It implements three steps within itself:

- ***TCB*-Att:** is an *RA* RoT implemented using VRASED and is always called upon trigger to measure (i.e., compute a MAC of) $\mathcal{P}rv$ binary and the current $\mathcal{CF}_{Log}$;
- ***TCB*-Wait:** always follows *TCB*-Att and is called to send the report (computed by *TCB*-Att) to $\mathcal{V}rf$ and wait for $\mathcal{V}rf$ decision on whether a remediation phase should follow (in case of compromise detection);
- ***TCB*-Heal:** implements the remediation action that may occur based on $\mathcal{V}rf$ decision after analyzing the report.

Figure 2 illustrates *ACFA* execution workflow alongside *ACFA* HW module responsible for (1) generating and protecting $\mathcal{CF}_{Log}$; and (2) issuing the trigger interrupt when a *CFA* report must be sent to $\mathcal{V}rf$. We highlight two important consequences of *ACFA* design and execution workflow. Even compromised software on $\mathcal{P}rv$ is unable to preclude sending of $\mathcal{CF}_{Log}$ to $\mathcal{V}rf$, as trigger cannot be disabled due to the active RoT guarantees and the sending function is implemented within the (atomically executed) *TCB*. Therefore, $\mathcal{V}rf$ receives $\mathcal{CF}_{Log}$ even in case of $\mathcal{P}rv$ compromise, enabling auditing of the exploit's control flow path to identify the vulnerability source. Similarly, *TCB*-Heal is also implemented within *TCB* and cannot be avoided by any external attempts originating from potentially compromised software on $\mathcal{P}rv$.

With this design, we envision *ACFA* to be particularly use-

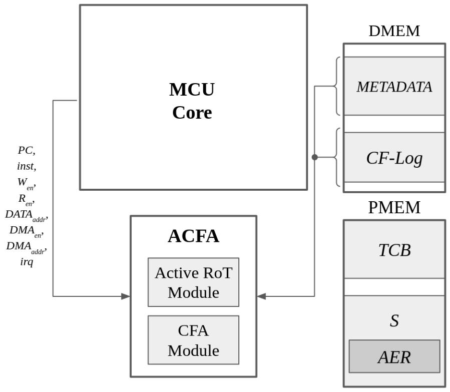


Figure 3: *ACFA* Architecture

ful in security-critical on-demand sensing applications, where $\mathcal{P}$rv is expected to perform sensor readings upon receiving $\mathcal{V}$rf commands. An end-to-end *ACFA* implementation with a sample application is presented in Section 6.2.

4 *ACFA* in Detail

This section presents *ACFA* details. We start by defining the adversary model, *ACFA* architecture, and *ACFA* protocol. We then deconstruct *ACFA* design into multiple required security properties and associated design elements that enforce each required property. Finally, we analyze the security of the overall construction according to the adversary model.

4.1 Adversary ($\mathcal{A}$dv) Model

We consider that $\mathcal{A}$dv can exploit software vulnerabilities in $\mathcal{P}$rv software to (1) modify any writable memory that is not explicitly protected by hardware-enforced access controls; (2) cause malicious control flow transfers on untrusted software; and (3) attempt to hide their malicious actions (in the form of injected code or hijacked control flows). Unless prevented, modifications to program memory can change instructions, and modifications to data memory can corrupt intermediate computation results, affecting the program's intended control flow. $\mathcal{A}$dv may also attempt to trigger interrupts or re-program any interrupt handler to achieve similar goals; re-programming an interrupt handler can be done by either modifying its software directly or modifying an entry in the interrupt vector table to point to any other (potentially malicious) software. In addition, $\mathcal{A}$dv has a Dolev-Yao [25] capability with respect to the network. Therefore, it may discard, inject, or attempt to modify messages between $\mathcal{P}$rv and $\mathcal{V}$rf. Hardware attacks that require physical access to circumvent $\mathcal{P}$rv hardware protections (or hardware-protected software) are out-of-scope in this paper. Protection against the latter involves orthogonal physical access control measures [42,47].

Table 1: Notation Summary

Symbol	Definition
PC	Program Counter (points to the instruction currently being executed).
inst	Bits of the currently executing instruction that specify its operation type
W_{en}	MCU write enable bit (is set whenever the CPU writes to memory)
R_{en}	MCU read enable bit (is set whenever the CPU reads from memory)
D_{addr}	MCU data address signal (contains the that address being read – when R_{en} is set – or written – when W_{en} is set – at each execution cycle).
DMA_{en}	DMA enable bit (is set whenever the DMA reads or writes from/to memory)
DMA_{addr}	DMA data address signal MCU data address signal (contains the that address being read or written when DMA_{en} is set is set – at each execution cycle)
irq	MCU signal that is set when an interrupt is occurring
TCB	Trusted computing base (PMEM location storing *ACFA* trusted software)
S	*PMEM*, except for *TCB*, i.e., region storing all untrusted application code
AER	Region storing code whose execution is to be attested. Located within *S*.
METADATA	*ACFA*-reserved *DMEM* region used to store *ACFA*-associated data
$\mathcal{CF}_{Log}$	Log that stores control flow transfers during *AER* execution
CF_{size}	Current size of $\mathcal{CF}_{Log}$

Remark: *As noted in Section 3, ACFA aims to guarantee $\mathcal{CF}_{Log}$ delivery and $\mathcal{P}$rv remediation assuming eventual communication. In case of a Dolev-Yao $\mathcal{A}$dv that discards all messages indefinitely, ACFA (in its strict version) intentionally halts $\mathcal{P}$rv's compromised execution.*

4.2 *ACFA* Architecture

Figure 3 presents *ACFA* architecture. *ACFA* HW interacts with the MCU Core and with main memory. It is composed of two sub-modules: the Active RoT Module and the *CFA* Module.

The MCU address space consists of program memory (*PMEM*) and data memory (*DMEM*). In *ACFA*, *PMEM* is divided between the *TCB* software and other (untrusted) application software, denoted *S*. *TCB* is located in a fixed memory region. *ACFA* protects this region by checking MCU signals at runtime. The Attested Executable Region (*AER*) is a subset of *S* containing the software of interest that should be attested/audited by $\mathcal{V}$rf. *AER* location and size are configurable. Therefore, $\mathcal{V}$rf can define what should be attested/audited: the execution of a code segment, a single function, multiple functions, or the entirety of *PMEM*. *ACFA* also reserves regions *METADATA* and $\mathcal{CF}_{Log}$ in fixed physical locations of *DMEM*. *METADATA* is used to store *ACFA*-related variables: the cryptographic challenge *Chal* (received from $\mathcal{V}$rf), addresses defining the the boundaries of *AER* in memory (AER_{min}, AER_{max}), and the current size of $\mathcal{CF}_{Log}$ (CF_{size}).

ACFA hardware monitors several MCU signals in order to enforce security properties. Table 1 summarizes the notation used in the rest of this paper, including CPU signals monitored by *ACFA* HW. Among these signals, the program counter (*PC*) contains the address of the current instruction being executed. This signal tells *ACFA* HW which software region (*TCB*, *S*, or *AER*) is executing. The *inst* signal contains the "opcode" of the currently executing instruction (as a bit-string). *inst* is used by *ACFA* to determine if a branch instruction is occurring. *ACFA* also monitors signals related to memory accesses – the write and read enable bits (W_{en},R_{en}) and the data address (D_{addr}) being accessed by the MCU. This allows *ACFA* to determine if a read/write is occurring and the respective memory address of the read/write operation, enabling prevention of illegal reads/writes. Similarly, *DMA* access signals (DMA_{en}, DMA_{addr}) are also monitored to detect *DMA* reads/writes and their destinations. Finally, *ACFA* also monitors signals related to interrupts such as the interrupt bit (*irq*) and the global interrupt enable bit (*gie*) to detect when interrupts are triggered, accepted, and enabled.

ACFA HW is composed of two sub-modules: the Active RoT module and the *CFA* module. Based on the aforementioned HW signals, they enforce several required security properties that will be described in detail in Section 4.5.

The Active RoT Module is responsible for the *guaranteed triggering* and *re-triggering on failure* properties (see Section 2.4) which guarantee the correct execution of *ACFA* TCB Software. *ACFA* protects each *TCB*-trigger source (**[T1]-[T3]**) from malicious software by implementing them as Non-Maskable Interrupts (NMIs). As opposed to normal interrupts, NMIs cannot be disabled in software. *ACFA* also ensures that the deadline of the periodic timer (used by trigger **[T1]**) is only configurable by TCB Software (i.e., when $PC \in TCB$). Similarly, *ACFA* creates a new NMI that is triggered whenever $\mathcal{CF}_{Log}$ region is full (**[T2]**) or when *AER* execution is concluded (**[T3]**) by triggering the NMI when $PC = AER_{max}$.

To assure integrity of the *CFA* report, the *CFA* Module detects any illegal attempts to modify data associated with the execution of *AER* (such as $METADATA$, $\mathcal{CF}_{Log}$, and *AER* binary itself), as this data is included in the *CFA* report and used by $\mathcal{V}$rf to interpret such report. *CFA* Module also detects and logs all control flow transfers (due to branches or interrupts) onto $\mathcal{CF}_{Log}$ in an optimized fashion.

Any violation to *ACFA* properties (as detected by *ACFA* HW) triggers an MCU reset (recall Figure 2). A reset implies execution of *TCB*. Therefore, *ACFA* ensures that an exploit always leads to $\mathcal{V}$rf receiving a *CFA* report that contains the exploit's control flow information, allowing $\mathcal{V}$rf to pinpoint the source of this exploit.

Remark: *As shown in Figure 3, ACFA operates in parallel with the MCU Core's execution pipeline. Hence, the execution critical path delay is not affected by ACFA.*

4.3 *ACFA* Protocol

Figure 4 presents *ACFA* protocol. A protocol instance starts when *TCB* is invoked on $\mathcal{P}$rv due to one of *ACFA* triggers (**[T1], [T2], or [T3]**). Recall that *ACFA* triggers include, boot/program end, expiration of a timer, and $\mathcal{CF}_{Log}$ being full. The timeout parameter can be configured to meet application needs. For instance, to minimize disruption in case of on-demand sensing applications, the deadline can be set to give sufficient time for the sensing code to complete its execution while still assuring that the report is always received by $\mathcal{V}$rf in a timely manner.

ACFA protocol implements the execution workflow illustrated in Figure 2. Once *TCB* is invoked in $\mathcal{P}$rv, it executes *TCB*-Att to produce an *RA* measurement H, as in Step 1 of Figure 4. H is computed on $PMEM$, $METADATA$ and $\mathcal{CF}_{Log}$, using a key ($\mathcal{K}$) that is pre-shared between $\mathcal{V}$rf and the RoT in $\mathcal{P}$rv. Then, in Step 2, $\mathcal{P}$rv transitions to *TCB*-Wait, generating and sending an *ACFA* report to $\mathcal{V}$rf. This report consists of H, $METADATA$ and $\mathcal{CF}_{Log}$. It then awaits for $\mathcal{V}$rf response. Upon receiving *ACFA* report, $\mathcal{V}$rf performs the verification process (`Verify`), in Step 4, by:

1. Checking validity of H. As $\mathcal{V}$rf possesses $\mathcal{P}$rv expected binary (denoted $PMEM'$), this check can be done by computing the expected H, i.e:
$$H \stackrel{?}{=} \texttt{HMAC}_{\mathcal{K}}(PMEM', METADATA, \mathcal{CF}_{Log})$$
2. Checking if $METADATA$ matches *Chal* and *AER* boundary as requested by $\mathcal{V}$rf in the previous instance of *ACFA* protocol. We note that when the protocol runs for the first time, $\mathcal{P}$rv has yet to receive any challenge or *AER* boundary from $\mathcal{V}$rf. In this case, $\mathcal{V}$rf instead compares $METADATA$ with a default value, i.e., 0.
3. Evaluating $\mathcal{P}$rv reported execution trace based on $\mathcal{CF}_{Log}$ and its size (CF_{size}), where CF_{size} is located inside $METADATA$. This step can employ a variety of techniques, such as evaluating $\mathcal{CF}_{Log}$ on *AER* control flow graph or emulating a shadow stack for *AER* execution. We discuss instantiations of $\mathcal{V}$rf in Section 6.1.

If verification succeeds, $\mathcal{V}$rf approves the report and thus sets the approval flag ($app := 1$), indicating that $\mathcal{P}$rv is allowed to continue execution; otherwise, the approval flag is cleared ($app := 0$). In Step 5, $\mathcal{V}$rf creates an *ACFA* response by incrementing the challenge $Chal'$, defining contiguous region of PMEM [AER_{min}, AER_{max}] that determines the next operation to be audited (which could remain the same), and computing an authentication token $\mathcal{A}uth$. This response is forwarded to $\mathcal{P}$rv in Step 6.

Upon receiving the response, $\mathcal{P}$rv authenticates $\mathcal{V}$rf message in Step 7 (including whether $Chal' > Chal$, for freshness). If authentication fails, $\mathcal{P}$rv goes back to waiting in Step 2. Only when authenticity of the response is confirmed, $\mathcal{P}$rv determines whether $\mathcal{V}$rf approves the report. In case that the report is not approved ($app = 0$ in the response), $\mathcal{P}$rv enters *TCB*-Heal to perform a remediation operation (e.g., system reset, software update) in Step 8. After the remediation finishes, it restarts the whole process from Step 1 in order to convince $\mathcal{V}$rf that the remediation was indeed performed successfully by attesting the new system state.

When $\mathcal{V}$rf approves ($app = 1$), $\mathcal{P}$rv is authorized to exit *TCB* and continue to Step 9 where $\mathcal{P}$rv begins executing the sensor application or resumes execution from where it left off before the *TCB*-trigger. While executing *AER*, *ACFA* hardware monitors execution and constructs $\mathcal{CF}_{Log}$. This continues until the occurrence of a new trigger, which in turn initiates a new instance of the *ACFA* protocol from Step 1.

To support the computation of H as well as the authentication of $\mathcal{V}$rf message in Step 7, *ACFA* leverages VRASED for *RA* (recall VRASED description from Section 2.2) which en-

Verifier ($\mathcal{V}$rf)		**Prover** ($\mathcal{P}$rv)
		1) When *TCB* is invoked (either by trigger **[T1]-[T3]** or by a manual call in software), $\mathcal{P}$rv executes *TCB*-Att to compute *RA* measurement: $$H := \texttt{HMAC}_{\mathcal{K}}(PMEM, METADATA, \mathcal{CF}_{Log})$$ where $\mathcal{K}$ is the *RA* key pre-shared between $\mathcal{V}$rf and VRASED *RA* RoT in $\mathcal{P}$rv. Then enter *TCB*-Wait.
3) Receive (H, *METADATA* and $\mathcal{CF}_{Log}$) and extract *Chal* from *METADATA*	← *ACFA* report	2) In *TCB*-Wait: Create and send *ACFA* report := $H \| METADATA \| \mathcal{CF}_{Log}$ and wait for approval.
4) Run verification (including analysis of $\mathcal{CF}_{Log}$) to determine whether to approve the report: $$app := \texttt{Verify}(H, \mathcal{K}, PMEM', METADATA, \mathcal{CF}_{Log})$$ where $PMEM'$ is the expected software for $\mathcal{P}$rv *PMEM* and $app \in \{0,1\}$ is an approval bit.		
5) Generate a new challenge $Chal'$, a memory region to be monitored (AER_{min}, AER_{max}) and an authentication token $\mathcal{A}uth$, where: $$\mathcal{A}uth := \texttt{HMAC}_{\mathcal{K}}(Chal', AER_{min}, AER_{max}, app)$$ $$Chal' := Chal + 1$$		
6) Create and send *ACFA* response $response := app \| Chal' \| AER_{min} \| AER_{max} \| \mathcal{A}uth$	*ACFA* response →	7) In *TCB*-Wait: Authenticate the response, producing a one-bit output: $$out := \texttt{Authenticate}(\mathcal{K}, ACFA \text{ response})$$ Based on *out* and *app*, it decides the next transition: • If $out = 0$: Re-enter *TCB*-Wait. *Jump to Step 2.* • Else if $app = 0$: Save ($Chal'$, AER_{min} and AER_{max} to *METADATA*) and enter *TCB-Heal*. *Jump to Step 8.* • Else: Save ($Chal'$, AER_{min} and AER_{max}) to *METADATA*, exit *TCB* and resume execution of *AER*. *Jump to Step 9.*
		8) In *TCB*-Heal: Execute remediation software (e.g., reboot, reset, software update), then re-start *TCB*-Att. *Jump to Step 1.*
		9) Resume Application Execution: • Whenever executing *AER*: append control-flow transfers to $\mathcal{CF}_{Log}$. • Whenever a trigger occurs, *ACFA* causes execution to enter *TCB*-Att. *Jump to Step 1.*

Figure 4: *ACFA* protocol.

sures that the secret key $\mathcal{K}$ used for *RA* and for authentication of $\mathcal{V}$rf message is not leaked, even in case of a compromised software state on $\mathcal{P}$rv. We also note that, to deal with network failures, *ACFA* report and response messages can be re-transmitted periodically, if the subsequent message in the protocol is not received from the respective communication end-point after a given time.

4.4 Required Security Properties

To support the correct execution of *ACFA* protocol defined in Section 4.3, irrespective of a potentially compromised software state in $\mathcal{P}$rv, *ACFA* enforces multiple properties to assure $\mathcal{CF}_{Log}$ Integrity (Properties **[P1-P3]**) and *TCB* Execution Integrity (Properties **[P4-P5]**).

[P1] Read-Only $\mathcal{CF}_{Log}$: $\mathcal{CF}_{Log}$ is read-only to *all* software. This property is necessary to ensure $\mathcal{CF}_{Log}$ integrity, i.e., $\mathcal{A}$dv cannot tamper with the content in $\mathcal{CF}_{Log}$. Without this property, $\mathcal{A}$dv could forge a valid control flow log without executing the intended software by simply overwriting the $\mathcal{CF}_{Log}$ region.

[P2] *METADATA* Integrity: The `Verify` algorithm (Step 4, in Figure 4) depends on *METADATA*. For this reason, *ACFA* guarantees *METADATA* can only be overwritten by *TCB* Software, which sets *METADATA* according to *ACFA* response (sent by $\mathcal{V}$rf in Step 6 of Figure 4). *METADATA* stores the bounds (AER_{min}, AER_{max}), defining *AER* region. *ACFA* detects and logs control flow transfers based on these boundaries. In addition, the *METADATA* contains the current size of the log (CF_{size}) and the cryptographic challenge (*Chal*). CF_{size} determines the total control flow transfers that happened since the last *ACFA* response message (as trigger may occur before $\mathcal{CF}_{Log}$ is full, due to a timer expiration or a violation of *ACFA* rules). *Chal* assures that subsequent *ACFA* reports cannot be replayed. Thus, *METADATA* must be write-protected from $\mathcal{P}$rv untrusted software.

Machine Model:

1. Memory Modification:

$$modifyMem(X) \equiv (W_{en} \wedge D_{addr} \in X) \vee (DMA_{en} \wedge DMA_{addr} \in X)$$

2. *inst* signal contains opcode of the instruction pointed by *PC*

3. $call_{irq}$ bit is set whenever a control flow transfer occurs due to interrupt handling.

Figure 5: Machine Model

[P3] $\mathcal{CF}_{Log}$ **Correctness:** All control flow transfers within *AER* (including any external jump into *AER*, e.g., to invoke *AER*) must be correctly detected and accurately recorded to $\mathcal{CF}_{Log}$. In other words, $\mathcal{CF}_{Log}$ must reflect the exact sequence of control flow transfers that have happened during *AER* latest execution (since receipt of the latest *Chal*).

[P4] Guaranteed *TCB* Triggering/Re-Triggering: trigger must result in guaranteed *TCB* execution upon occurrence of **[T1]**, **[T2]**, and **[T3]** cases (defined in Section 3). In case of **[T1]**, *TCB* must be triggered periodically, to enable auditing of time sensitive tasks. The period is configurable from within *TCB*. *TCB* must also be triggered when $\mathcal{CF}_{Log}$ is full **[T2]**, to free $\mathcal{CF}_{Log}$ for new control flow transfers. Lastly, **[T3]** requires *TCB* execution to be triggered on a reset/boot or when *AER* execution completes. The former is necessary to prevent $\mathcal{A}$dv from triggering resets to avoid auditing of *ACFA* report by $\mathcal{V}$rf or to avoid the active remediation phase. The latter is required since the reaching of AER_{max} may occur before $\mathcal{CF}_{Log}$ is full and before the expiration of the timer.

[P5] *TCB* Integrity: Since *TCB* is responsible for various security-critical operations in *ACFA*, its integrity is crucial. Its instructions must be write-protected from all other software in $\mathcal{P}$rv. Once called, *TCB* must execute atomically, i.e., it faithfully executes the sequence *TCB*-Att $\rightarrow$ *TCB*-Wait $\rightarrow$ *TCB*-Heal, without interruptions and without interference from other software in $\mathcal{P}$rv. Atomicity should also prevent jumps to the middle of *TCB*, as they could be used to initiate out-of-order execution of *TCB* code. The *RA* result in *TCB*-Att (Step 1 in Figure 4) must be unforgeable to assure that *ACFA* report is authentic (in turn, this requires absolute confidentiality of the *RA* secret key). Similarly, *ACFA* response sent by $\mathcal{V}$rf (Step 6 in Figure 4) defines actions to be taken on $\mathcal{P}$rv and therefore must be authenticated by *TCB*. The *RA* RoT in *ACFA* must implement these functions securely, irrespective of any compromised software outside *TCB*.

4.5 Specification of *ACFA* Components

We now discuss how *ACFA* enforces **[P1]-[P5]**, presented in Section 4.4. In particular, we define the logic required to implement these properties based on the MCU signals monitored by *ACFA* HW (recall *ACFA* monitored signals from Table 1). This logic is then implemented by *ACFA* HW.

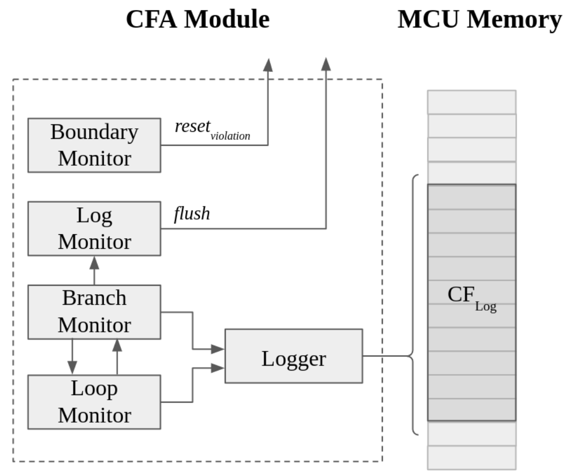


Figure 6: Sub-Modules within the *CFA* module

Figure 5 defines the MCU machine model based on how the MCU behavior is reflected in its hardware signals. First, $modifyMem(X)$ predicate models MCU signals whenever a given memory address X is modified by either CPU or DMA. In order for the CPU to modify memory region X, the W_{en} bit must be set and D_{addr} must point to a location within X. Similarly, in order for DMA to modify X, DMA_{en} must be set and DMA_{addr} must be within X. The *inst* signal contains the opcode determining the instruction that is currently being pointed by *PC* in *PMEM*. Each distinct instruction in the CPU instruction set architecture has a different opcode. Finally, $call_{irq}$ bit is set whenever a control flow transfer happens due to interrupt handling.

CFA Module

The *CFA* module in *ACFA* is depicted in Figure 6. It consists of five sub-modules.

- **Boundary Monitor:** Boundary Monitor enforces **[P1]** and **[P2]** as specified in the logic of Figure 7, preventing unauthorized modifications to *METADATA* and $\mathcal{CF}_{Log}$. If *METADATA* is modified by any software other than *TCB* or if there is a software-write attempt to $\mathcal{CF}_{Log}$, it resets $\mathcal{P}$rv.
- **Branch Monitor:** As specified in Figure 8, Branch Monitor detects control flow transfers by checking the *inst* signal to identify branching instructions, i.e., `call`, `jmp`, `ret`, and `reti`. Additionally, it monitors the $call_{irq}$ signal to detect branches due to interrupts. It outputs $branch_{detect} = 1$ if a control flow transfer is detected. Depending on the CPU state, a variable number of instructions may execute in between the moment when an interrupt is received (when $irq = 1$) and the moment when the jump to the ISR is triggered (i.e., the moment when the control flow transfer occurs). Therefore, $call_{irq}$ itself is determined based on a combination of the MCU signals *irq* and *gie*. We defer these implementation-specific details to Appendix A.

HW Specification: Monitor Boundaries of *METADATA* and $\mathcal{CF}_{Log}$

$$[modifyMem(METADATA) \land PC \notin TCB] \lor modifyMem(\mathcal{CF}_{Log}) \lor (DMA_{en} \land DMA_{addr} \in METADATA) \rightarrow reset \tag{1}$$

Figure 7: Hardware Spec.: Boundary Monitor Sub-Module

HW Specification: Detect branch due to instruction or interrupt

$$[(inst = call) \lor (inst = jmp) \lor (inst = ret) \lor (inst = reti) \lor call_{irq}] \rightarrow branch_{detect} \tag{2}$$

Figure 8: Hardware Spec.: Branch Monitor Sub-Module

• **Log Monitor:** The Log Monitor tracks the size of $\mathcal{CF}_{Log}$ (CF_{size}) and controls the **[T2]** trigger (activated when $\mathcal{CF}_{Log}$ is full). In Figure 6, **[T2]** is represented by the *flush* hardware signal, which is controlled by Log Monitor and used as an NMI input to the CPU. Thus, when $flush = 1$, it immediately launches *TCB* execution. Log Monitor is also responsible for indicating that a control flow transfer is ready to be written to $\mathcal{CF}_{Log}$, by setting the bit $hw_{en} = 1$. The hw_{en} bit is an internal wire of the *CFA* module used for communication between the Log Monitor and the Logger sub-modules. Figure 9 details the logic implemented to control CF_{size} and hw_{en}. It compares CF_{size} to the maximum size of $\mathcal{CF}_{Log}$ to determine if $\mathcal{CF}_{Log}$ is full, and the result of this comparison determines if log_{full} bit must be set. Whenever a branch is detected during *AER* execution ($PC \in AER$) and $\mathcal{CF}_{Log}$ is not full, it sets $hw_{en} = 1$; otherwise, hw_{en} remains 0. The same bit is set to 1 when *AER* is invoked, which is detected by monitoring the next *PC* value (PC_{next}). In addition, Log Monitor is responsible for incrementing CF_{size} to keep track of the next unused position in $\mathcal{CF}_{Log}$. It ensures to clear CF_{size} after *TCB* execution completes ($PC = TCB_{max}$). Therefore, upon returning from *TCB*, $\mathcal{CF}_{Log}$ will be overwritten by subsequent control flow transfers. The $loop_{detect}$ signal is used for optimization purposes and controlled by the Loop Monitor sub-module that will be discussed next.

• **Loop Monitor:** The Loop Monitor is used to reduce $\mathcal{CF}_{Log}$ size by optimizing repetitive $\mathcal{CF}_{Log}$ entries produced by static loops without internal branching instructions (such as delay loops, which are common in embedded system software). These loops can quickly fill $\mathcal{CF}_{Log}$ with redundant control flow transfers. Thus, *ACFA* follows a similar approach to prior *CFA* methods [23,24] by considering each repeated backward jump with the same source and destination addresses (*src, dest*) as a static loop and simply logging (*src, dest*) once, along with the number of iterations that occurred. To differentiate between $\mathcal{CF}_{Log}$ entries generated by static loops from regular entries, Loop Monitor controls the $loop_{detect}$ signal. Since this feature is strictly used for optimization purposes, we defer its details to Appendix B. All control flow transfers in a static loop ($loop_{detect} = 1$) do not increment CF_{size} but instead write the number of iterations in place. Once the control flow leaves the static loop, CF_{size} is incremented again.

HW Specification: Secure Logging

$$CF_{size} = maxSize(\mathcal{CF}_{Log}) \rightarrow log_{full} \tag{3}$$

$$(PC \in AER) \land branch_{detect} \land \neg log_{full} \rightarrow hw_{en} \tag{4}$$

$$\neg(PC \in AER) \land (PC_{next} \in AER) \land branch_{detect} \land \neg log_{full} \rightarrow hw_{en} \tag{5}$$

$$hw_{en} \land \neg log_{full} \land \neg loop_{detect} \rightarrow CF_{size}{++} \tag{6}$$

$$PC = TCB_{max} \rightarrow CF_{size} = 0 \tag{7}$$

HW Specification: *TCB* Trigger [T2]

$$log_{full} \lor reset \rightarrow flush \tag{8}$$

Figure 9: Hardware Spec.: Log Monitor Sub-Module

HW Specification: Log Entry Construction

$$src = \begin{cases} PC_{prev}, & \text{if } \neg loop_{detect} \\ ctr[31:16], & \text{if } loop_{detect} \end{cases} \tag{9}$$

$$dest = \begin{cases} PC, & \text{if } \neg loop_{detect} \\ ctr[15:0], & \text{if } loop_{detect} \end{cases} \tag{10}$$

HW Specification: Secure Log Update

$$hw_{en} \rightarrow \mathcal{CF}_{Log}[CF_{size}] = (src, dest) \tag{11}$$

Figure 10: Hardware Spec.: Logger Sub-Module

• **Logger:** The sole responsibility of the Logger module is to write the next entry into the $\mathcal{CF}_{Log}$. Each entry is a pair of source and destination addresses. Since MSP430 uses 16-bit addresses, entries are 32-bit values. Figure 10 shows the logic to append an entry to $\mathcal{CF}_{Log}$. Whenever $hw_{en} = 1$, Logger appends ($src = PC_{prev}, dest = PC$) to $\mathcal{CF}_{Log}$, where PC_{prev} denotes the previous *PC* value. If a static loop is detected ($loop_{detect} = 1$), the loop counter *ctr* is additionally logged ($src = ctr[31:16], dest = ctr[15:0]$) as a next $\mathcal{CF}_{Log}$ entry and incremented accordingly.

In summary, the CFA Module supports security properties **[P1-P4]**. The Boundary Monitor supports both **[P1-P2]** to ensure critical data cannot be modified maliciously. In combination, Log Monitor, Branch Monitor, Loop Monitor, and Logger detect and record all control flow transfers to $\mathcal{CF}_{Log}$, implementing **[P3]**. Furthermore, **[P4]** is supported by the Log Monitor ensuring **[T2]** will always cause TCB to execute.

Active RoT Module

Properties **[P4-P5]** also rely on the active RoT guarantees discussed in Section 2.4. The sequence *TCB*-Att→*TCB*-Wait→*TCB*-Heal is implemented within that active RoT handler function $\mathcal{F}$. To ensure that triggers **[T1]**, **[T2]**, and **[T3]** always result in the execution of $\mathcal{F}$, they are NMIs (that cannot be disabled in software) thus supporting **[P4]**. In addition, *ACFA* hardware makes use of some of the original hardware modules in GAROTA [4] implementation to ensure ***Non-malleability*** and ***Atomicity*** of *TCB*, thus supporting **[P5]**. VRASED is ported in its entirety into *ACFA*. It is required to implement the *TCB-Att* phase of the *TCB* sequence securely, i.e., securely producing an *RA* measurement and authenticating $\mathcal{V}$rf messages, per property **[P5]**. VRASED verified architecture enables secure *RA* by ensuring confidentiality of the attestation key and proper execution of the *RA* integrity-ensuring function. It assures that untrusted software in $\mathcal{P}$rv cannot access any trace of $\mathcal{K}$ before, during, or after *TCB*-Att computation. In addition, VRASED ensures that *TCB-Att* remains immutable, executes atomically, and has fixed entry and exit points (see [18] for details on VRASED internals).

4.6 Security Analysis

Recall from Section 4.1 that $\mathcal{A}$dv can exploit vulnerabilities in $\mathcal{P}$rv software *S*, to modify any code or data (including stack data to perform control flow hijacks). Similarly, $\mathcal{A}$dv may use this capability to disrupt any phase of *ACFA* protocol.

$\mathcal{A}$dv may attempt to forge/modify $\mathcal{CF}_{Log}$ to reflect the control flow path of *AER* faithful execution without truly executing it. One approach is to modify $\mathcal{CF}_{Log}$ directly. However, as *ACFA* prevents all CPU/DMA accesses to $\mathcal{CF}_{Log}$ (**[P1]**), any such attempt results in a system reset, triggering *TCB* to inform $\mathcal{V}$rf of this attack. In addition, $\mathcal{A}$dv may attempt to forge $\mathcal{CF}_{Log}$ by causing *ACFA* to track a different executable $AER_{\mathcal{A}dv}$, located elsewhere in *PMEM*, by modifying the bounds in *METADATA*. However, **[P2]** assures that such an attempt to modify *METADATA* is prevented. Similarly, $\mathcal{A}$dv cannot overwrite *Chal* in an attempt to replay the *CFA* report produced by a previous execution of *AER*.

If $\mathcal{A}$dv exploits vulnerabilities to diverge *AER* control flow, the exploit will be reflected in $\mathcal{CF}_{Log}$ (given **[P3]**). Then, to escape $\mathcal{V}$rf detection, $\mathcal{A}$dv must directly forge an authenticated *CFA* report containing a benign control flow path. In order to forge this report, $\mathcal{A}$dv must forge the result of the authenticated integrity-ensuring function (MAC) computed by *TCB*-Att, which in turn requires tampering with *TCB* execution. However, this is infeasible due to **[P5]**.

$\mathcal{A}$dv may try to cause a deadlock in the *CFA* protocol so that $\mathcal{V}$rf-issued remediation is never performed on $\mathcal{P}$rv. Since *TCB* is automatically triggered on *AER* completion, $\mathcal{A}$dv may continuously interrupt *AER* to prevent *TCB* from ever being called. Additionally, $\mathcal{A}$dv may overwrite data to cause an infinite loop within *AER* so that its execution never completes. However, this will be preempted by either: *ACFA* timer trigger or *ACFA* $\mathcal{CF}_{Log}$ full trigger. Since *TCB* is guaranteed to execute thereafter (**[P4]**), $\mathcal{A}$dv cannot avoid detection/remediation in this way.

$\mathcal{A}$dv may attempt to tamper with *TCB* behavior by changing its code or interrupting its execution (e.g., to prevent the remediation phase in *TCB-Heal*). However, *TCB* code is immutable at runtime and its execution is atomic (**[P5]**). Any attempt to interrupt *TCB* will cause $\mathcal{P}$rv to reset and re-execute *TCB* from the start (due to *re-triggering on failure* property). After a reset, interrupts and DMA are disabled by default. Therefore, no further interference by $\mathcal{A}$dv is possible during the new instance of *TCB* execution.

Finally, a network $\mathcal{A}$dv may discard messages between $\mathcal{V}$rf and $\mathcal{P}$rv. *ACFA* guarantees that $\mathcal{P}$rv remains in the *TCB-Wait* phase until an approval message from $\mathcal{V}$rf is eventually received. Therefore, even when $\mathcal{A}$dv controls both $\mathcal{P}$rv and the network, a compromised *AER* is prevented from executing.

5 Alternative Designs & Policies

This section discusses alternatives in *ACFA* design and policies, as well as their implications.

• ***ACFA* with Hardware Hash Engines:** *ACFA* standard design opts for a software implementation of the *RA* integrity-ensuring function (using VRASED). This choice significantly reduces the hardware cost. However, it also increases storage requirements in order to maintain $\mathcal{CF}_{Log}$ verbatim. It also increases the potential number of partial transmissions of $\mathcal{CF}_{Log}$ to $\mathcal{V}$rf, when this dedicated storage fills up. We note that, if $\mathcal{P}$rv is implemented with a less strict hardware budget, hardware-based hash engines can be utilized to reduce the storage/transmission overhead. In practice, trade-offs between hardware cost and other overheads should be considered when deciding for one approach over the other. Early *CFA* methods [2, 23, 24, 65] proposed to build $\mathcal{CF}_{Log}$ as a hash-chain to compress the sequence of control flow transfers into a single hash digest. In that way, $\mathcal{P}$rv is only required to store the current hash-chain digest and extend it with the next control flow transfer as it occurs. While this can be obtained in a relatively easy way (assuming hash engines are available), it also requires $\mathcal{V}$rf to enumerate all control flow paths (for the entire execution) that could have led to the received final hash digest. Unfortunately, the complexity of this task grows exponentially with the number of control flow transfers in the path, leading to the well-known path explosion problem [8, 21, 46, 56]. With these trade-offs in mind, we also provide an implementation of *ACFA* using a hardware hash engine and compare its overhead with the original *ACFA* design. This implementation and comparative results are discussed in Section 6.

• **Strict vs. Best-Effort Auditing and Remediation:** as discussed in Section 3, we describe the strict version of *ACFA* protocol, in which auditing software integrity is a first-class priority (e.g., consider an MCU deployed as a part of a nuclear facility). Therefore, whenever a *CFA* report is sent to $\mathcal{V}$rf, *TCB* in $\mathcal{P}$rv waits for a response indefinitely (while re-transmitting the report periodically). In that case, to avoid detection, a Dolev-Yao $\mathcal{A}$dv might jam the network communication rendering $\mathcal{P}$rv unavailable. We note that resuming the execution of the attested application in this case is entirely possible (i.e., by jumping from step 2 to step 9 in Figure 4 upon a timeout). While this may be desired in some application domains, it remains unclear why one would aim to guarantee availability to a compromised application. Alternatively, *TCB* could resume *AER* execution for a fixed finite period, issuing a subsequent timer trigger to check if $\mathcal{V}$rf response was received. In the latter, $\mathcal{P}$rv does not "busy-wait" on $\mathcal{V}$rf response. The security implications of these policies to the application domain should be considered carefully.

• **Adapting *ACFA* to Higher-End Devices:** as noted in Section 2.1, *ACFA* initial design targets bare-metal MCUs. Adapting *ACFA* for higher-end systems is an interesting and promising direction for future work. The main challenge lies in the dependence of higher-end systems on MMU-based virtual memory assignment, whereas *ACFA* performs its checks based on physical addresses. On higher-end devices, the MMU translations are themselves controlled by privileged software that could be itself compromised and tamper with address translations to circumvent *ACFA*. Future work could consider methods to verify the consistency of virtual-to-physical address translations across the runtime of an attested/audited process, perhaps by augmenting MMUs with new (yet backward-compatible) hardware features.

• **Non-Control Data Attestation:** Subtle software integrity attacks are still possible by compromising non-control data, without modifying a program's control flow path. While this class of vulnerabilities (e.g., "write anywhere" vulnerability) is less common, they are still possible. One approach to deal with this problem is to append all data inputs (any memory read from outside the attested program's stack) to $\mathcal{CF}_{Log}$, as proposed in [41]. In possession of both the executed control flow path and all data inputs, $\mathcal{V}$rf can abstractly execute the attested program [56] to observe any such exploit.

6 Implementation & Evaluation

ACFA implements the workflow and architecture shown in Figures 2 and 3, respectively. As discussed in Section 2.1, our prototype is built on the low-end MSP430 MCU, primarily due to its simplicity and open-source availability.

We use Xilinx Vivado tool-set to synthesize *ACFA* hardware. *ACFA* hardware is written in the Verilog hardware description language and implements each of *ACFA* submodules according to the logic defined in Section 4. In total, *ACFA* hardware is implemented in 2042 lines of Verilog code. The *CFA* module accounts for 982 lines, the Active RoT module (including VRASED) accounts for additional 927 lines, and 123 lines tie the two modules together. We then synthesized and deployed *ACFA* on the Basys3 prototyping board that features an Artix-7 FPGA.

The *TCB* Software implements functions, *TCB*-Att, *TCB*-Wait, and *TCB*-Heal, as described in Section 4. All three functions are linked so that the entire *TCB* code is located within a contiguous region of *PMEM*. This ensures that *TCB* can be monitored and protected as intended by the *ACFA* hardware.

TCB-Wait is responsible for communicating with $\mathcal{V}$rf and authenticating $\mathcal{V}$rf messages. We use VRASED authentication module to support $\mathcal{V}$rf authentication in *TCB*-Wait. In our prototype, $\mathcal{P}$rv and $\mathcal{V}$rf are physically connected using a USB-UART interface. As explained in Section 3, *TCB*-Heal is configurable to meet application needs. In our prototype and evaluation, we implement a simple remediation option: shutting down $\mathcal{P}$rv.

To assess the trade-off discussed in Section 5, we also implement an *ACFA* variant using a hash engine. It integrates a SPONGENT hash engine implemented for openMSP430 by the SANCUS project [40]. In this variant, the hash engine module replaces the Logger module and receives the same inputs, i.e., $(src, dest)$ and hw_{en}. When hw_{en} is set, the $(src, dest)$ pair is accumulated into the SPOGENT hash digest. The hash engine operates with default parameters: at 100MHz in a 128-128-8 bit SPONGENT configuration, producing a 128-bit digest. Since each control flow contains 32 bits of data (16-bit source and destination addresses), the hash engine reads 8 bits at a time from a 512-bit FIFO buffer of control flow transfers at each cycle. With the hash engine, the Logger module is no longer required. Similarly, as $\mathcal{V}$rf is not provided with $\mathcal{CF}_{Log}$ verbatim (see Section 5 for a discussion on implications related to path explosion), the $flush$ signal and the tracking of $\mathcal{CF}_{Log}$ size are not required. The hash engine (including its integration with other *ACFA* modules) is implemented in 730 lines of Verilog code.

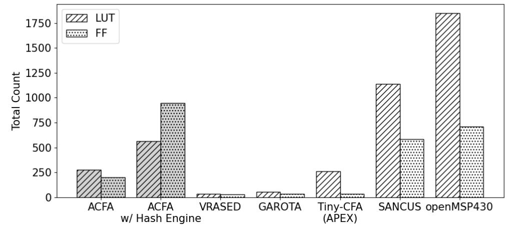
6.1 Results

To the best of our knowledge, no prior work implements *ACFA* features. Nonetheless, to provide a reference point, we report *ACFA* costs in comparison to other security architectures targeting the same class of MCUs (namely VRASED [18], SANCUS [40], GAROTA [4], and Tiny-CFA [21]). We also compare *ACFA* to closely related hardware-based *CFA* architectures (namely LiteHAX [24], LoFAT [23] and Atrium [65]) noting that these architectures were implemented on a different MCU class with a less strict hardware budget than the MSP430.

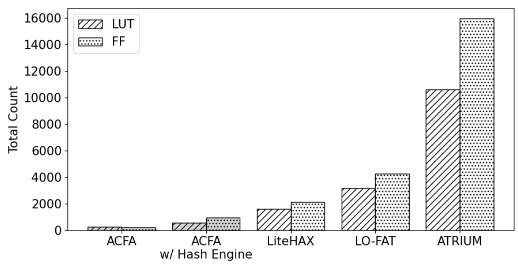
Hardware and Memory Overhead

Similar to the related work, we consider the hardware overhead in terms of additional Look-up Tables (LUTs) and flip-flops/registers (FFs). The increase in LUTs estimates the additional chip cost and size due to combinatorial logic. The number of extra FFs indicates additional state required by sequential logic. Figure 11 compares *ACFA* hardware cost with other architectures and a baseline unmodified openMSP430 core. Overall, *ACFA* requires additional 275 LUTs and 202 FFs. This represents a ≈18.7% increase with respect to the openMSP430 core.

Due to its hybrid design, the standard version of *ACFA* incurs significantly lower hardware overhead than hardware-based approaches such as SANCUS, as shown in Figure 11(a). Compared to other hybrid architectures, i.e., Tiny-CFA/APEX, *ACFA* incurs the similar number of LUTs but requires more FFs for sequential logic. Since *ACFA* is a superset of



(a) MSP430-based implementations



(b) *CFA*-specific architectures

Figure 11: Comparison of hardware cost

VRASED and GAROTA, *ACFA* hardware cost is naturally larger than the two.

Figure 11(b) compares *ACFA* with hardware-based *CFA* architectures, showing that a hybrid design for *CFA* requires less hardware. For instance, *ACFA* requires ≈5.8 times less LUTs and ≈10.5 times less FFs than LiteHAX, which is the cheapest related hardware-based *CFA* architecture.

Finally, the *ACFA* variant equipped with a hash engine adds 510 LUTs and 946 FFs to the openMSP430 baseline, representing an increase of 235 LUTs and 744 FFs over *ACFA* standard design. This difference highlights the hardware savings of a hybrid *CFA* approach. Nonetheless, it is important to note that a hash engine reduces the storage/transmission overhead of $\mathcal{CF}_{Log}$ on $\mathcal{P}$rv. On the other hand, it also increases verification complexity (to be performed by $\mathcal{V}$rf) exponentially due to the path explosion problem [2, 21, 56, 59]. These trade-offs should be considered carefully when deciding for a particular design option.

Runtime Overhead

Since *ACFA* does not require code instrumentation, no runtime overhead is incurred to save entries to $\mathcal{CF}_{Log}$. Similarly, there is no code size increase for *AER*. The exact runtime overhead incurred by the *TCB* execution varies depending on factors such as communication delays, $\mathcal{V}$rf choice of remediation function, and time taken by $\mathcal{V}$rf to verify reports. In practice, however, when testing our end-to-end application use-cases, we have noticed that this overhead is dominated by the time required to compute the HMAC function on $\mathcal{P}$rv and to communicate between $\mathcal{P}$rv and $\mathcal{V}$rf. Because of this, $\mathcal{V}$rf should consider a suitable configuration of $\mathcal{CF}_{Log}$ and *AER* sizes depending on their response time requirements.

Table 2: Runtime statistics for 0.5kB and 1 KB $\mathcal{CF}_{Log}$

Program	Max CF_{size}	# [T1]	# [T2]	# [T3]	$\mathcal{CF}_{Log}$ Data	# Reports
Ultrasonic Sensor	0.5kB	0	0	2	0.01 KB	2
	1.0kB	0	0	2	0.01 KB	2
Temperature Sensor	0.5kB	0	2	2	1.2 KB	4
	1.0kB	0	1	2	1.2 KB	2
Syringe Pump	0.5kB	0	7	2	3.6 KB	9
	1.0kB	5	0	2	3.6 KB	7

Table 3: *ACFA* vs. prior *CFA* for MCUs qualitatively

	C-FLAT [2]	LO-FAT [23],LiteHAX [24]	Tiny-CFA [21],DIALED [41],OAT [56]	*ACFA*
SW Instrumentation	Yes	No	Yes	**No**
HW Hash Engine	No	Yes	No	**No**
$\mathcal{CF}_{Log}$ slicing	No	No	No	**Yes**
Control Flow Auditing	No	No	No	**Yes**
Active Remediation	No	No	No	**Yes**

We discuss more details and timing results for the end-to-end prototype in Section 6.2.

Evaluation with Sample Applications

We evaluate *ACFA* on three exemplary applications (which were ported to run on openMSP430): an Ultrasonic Sensor [51], a Temperature Sensor [50], and a Syringe Pump [63].

During evaluation of the application software, we fix the timeout period (for trigger **[T1]**) to 50ms. We note that in practice we expect this time-out to be much larger. However, we choose a small value to force trigger occurrences so as to evaluate a worst-case. We consider two maximum $\mathcal{CF}_{Log}$ sizes: 0.5kB and 1.0kB (similar to the timer we intentionally choose very small $\mathcal{CF}_{Log}$ sizes to force trigger-s). The boundaries of *AER* are fixed to cover the entire untrusted software: $AER = S$. Table 2 shows the size of $\mathcal{CF}_{Log}$ data (i.e., total $\mathcal{CF}_{Log}$ bytes), the number of *ACFA* reports sent to $\mathcal{V}$rf, and the number of trigger-s issued during execution of each sample application under both maximum CF_{size} settings.

The Ultrasonic Sensor application contains very few control flow transfers, so its execution does not fill up $\mathcal{CF}_{Log}$. Thus, executing this application causes only two **[T3]** trigger-s (one at boot and one at the end of execution). On the other hand, both Temperature Sensor and Syringe Pump applications produce more control flow transfers, filling up the 0.5kB in $\mathcal{CF}_{Log}$ before their execution is completed. Hence, additional trigger-s occur during their execution to send partial $\mathcal{CF}_{Log}$ reports to $\mathcal{V}$rf. For both of these applications, increasing the maximum CF_{size} results in fewer trigger-s, since **[T2]** trigger will happen less often. We observe that, for the Syringe Pump application, fewer reports are generated and also fewer trigger-s occur with a larger $\mathcal{CF}_{Log}$ size. In addition, the source of intermediate reports changes from **[T2]** to **[T1]**. This is because all intermediate reports surpass 0.5kB before reaching the 50ms threshold. However, this does not occur when $\mathcal{CF}_{Log}$ size is 1KB. This small change also causes fewer reports to be generated due to the runtime of the application.

Given these observations, device operators should consider the trade-off between resource allocation (e.g., for storing $\mathcal{CF}_{Log}$), timeout periods, and application requirements.

Energy Consumption

Using the Vivado tool-set, we synthesize the hardware and generate energy consumption reports for *ACFA*. The unmodified openMSP430 requires 0.06W, whereas *ACFA* hardware requires an additional 0.001W. This represents a 1.6% increase in energy consumption.

CFA Verification

Our discussion thus far emphasizes *ACFA* architecture on $\mathcal{P}$rv. Based on the authenticated information produced by this architecture, $\mathcal{V}$rf must determine if a runtime attack has occurred. This verification can be implemented in a number of ways. In any case, $\mathcal{V}$rf must always check the *RA* measurement to confirm that the expected binary has not been modified. Similarly, it must examine the size and contents of $\mathcal{CF}_{Log}$ to determine if any violations occurred during the execution and generation of $\mathcal{CF}_{Log}$. In our implementation, $\mathcal{V}$rf generates the CFG of $\mathcal{P}$rv binary to check if the control flow transfers reported in $\mathcal{CF}_{Log}$ match a valid path in the CFG. $\mathcal{V}$rf also uses a shadow stack to confirm the validity of return addresses reported in $\mathcal{CF}_{Log}$. Aside from this sample implementation, any CFI policy that would otherwise be implemented on the resource-constrained $\mathcal{P}$rv can now be outsourced to the higher-end $\mathcal{V}$rf for faster and less intrusive runtime integrity verification.

6.2 End-to-End Prototype

To demonstrate *ACFA*'s practicality in on-demand sensing settings, we implement a fully-functional prototype including $\mathcal{P}$rv and $\mathcal{V}$rf realizing the *ACFA* end-to-end workflow in real-time. The implementation and a video demonstration of this end-to-end example are also available at [12].

$\mathcal{P}$rv **setup:** In this application, $\mathcal{P}$rv contains a simple program that receives a password input from a remote user and compares it with an expected password. We intentionally introduce a buffer-overflow vulnerability in this program by not performing array bound checks when storing the received password input. As a result, $\mathcal{A}$dv can overflow the buffer and overwrite a return address. If the correct password is entered, $\mathcal{P}$rv then records six ultrasonic-sensor readings and exits the program. In terms of *ACFA* configuration parameters, we set the maximum $\mathcal{CF}_{Log}$ size to 256B and the timeout period to an overwhelmingly large value, essentially deactivating timeout triggers (**[T1]**).

$\mathcal{V}$rf **Offline Phase (performed once):** We implement $\mathcal{V}$rf in Python and execute it on a 64-bit Ubuntu 18.04 machine with an Intel i7 @ 3.6GHz. $\mathcal{V}$rf offline phase consists of two main tasks. First, $\mathcal{V}$rf pre-computes the $\mathcal{P}$rv program's CFG by parsing the object-dump file of the application binary. In addition, $\mathcal{V}$rf prepares to verify the *ACFA* report (Step 4 in Figure 4) by pre-computing the hash of *AER*. Since *AER* is not expected to change between the reports, this optimization is put in place to speed up the online verification process. The entire offline phase takes $\approx$2.5s.

$\mathcal{V}$rf **Online Phase (performed on every protocol instance):** In this phase, $\mathcal{V}$rf receives $\mathcal{CF}_{Log}$ slice from $\mathcal{P}$rv and aims to validate whether $\mathcal{CF}_{Log}$ corresponds to a valid software execution, i.e., following a specific path in CFG generated from the offline phase. Each received $\mathcal{CF}_{Log}$ slice may be acquired from the first, intermediate and last *ACFA* reports. $\mathcal{V}$rf validates $\mathcal{CF}_{Log}$ from each type of report as follows:

- **First *ACFA* report.** Recall that a first *ACFA* report is valid if it is produced on $\mathcal{P}$rv immediately after *AER* is invoked. Thus, in this phase, $\mathcal{V}$rf checks whether its first entry corresponds to a function call to *AER*, i.e., the destination address matches AER_{min}. The rest of $\mathcal{CF}_{Log}$ entries must adhere to a valid control flow path in the CFG generated in the offline phase.
- **Intermediate *ACFA* reports.** In all reports other than the first and the last, the first $\mathcal{CF}_{Log}$ entry must correspond to a jump from TCB_{max} to *AER* after obtaining $\mathcal{V}$rf approval for the previous report. Subsequent entries must continue a valid control flow path. Hence, $\mathcal{V}$rf checks the first entry by comparing its source address to TCB_{max} and its destination to *AER* region. To validate other entries, $\mathcal{V}$rf keeps traversing CFG.
- **Last *ACFA* report.** *AER* execution completion results in a trigger to the *TCB*. The first $\mathcal{CF}_{Log}$ entry in this report is checked in the same way as in an intermediate report. $\mathcal{V}$rf validates the last $\mathcal{CF}_{Log}$ entry by matching it with a control flow transfer caused by a jump from AER_{max} to TCB_{min}. For other $\mathcal{CF}_{Log}$ entries, $\mathcal{V}$rf continues to traverse the CFG.
- **Single *ACFA* report.** In the case that no intermediate reports are created, $\mathcal{V}$rf obtains a single *ACFA* report that covers the entire *AER* execution trace. $\mathcal{V}$rf checks validity of this trace via CFG traversal.

In addition, to detect attacks overwriting return addresses, we implement a shadow stack on $\mathcal{V}$rf during the online phase. While traversing the CFG, if $\mathcal{V}$rf encounters a function call, it pushes the expected return address to the shadow stack. Upon returning from a function, $\mathcal{V}$rf pops the return address from the stack and matches it with the corresponding $\mathcal{CF}_{Log}$ entry obtained from $\mathcal{P}$rv.

Remediation Options: Some remediation options can be implemented with very few lines of code, such as a system shutdown. In MSP430, for instance, this can be implemented by simply setting a bit in the system status register. Similarly, system reset can be achieved by calling the reset vector (the first address of the interrupt vector table). In other cases, $\mathcal{V}$rf may implement a more thorough remediation option, such as erasing data memory or updating the MCU software followed by a system reset. Our end-to-end example supports both shutdown and reset options. In our sample implementation, $\mathcal{V}$rf detects an invalid $\mathcal{CF}_{Log}$ in the second instance of the *ACFA* protocol and thus successfully discovers the buffer-overflow attack on $\mathcal{P}$rv. In this case, $\mathcal{V}$rf chooses to remediate $\mathcal{P}$rv by shutting it down, preventing the malicious program from

Table 4: End-to-end protocol timing (in ms) for varying size of AER. Steps correspond to the protocol steps of Figure 4

AER Size	Steps 1-3	Step 4	Step 5	Step 6	Steps 7-8	Total
1 KB	351.9	2.096	6.70E-02	205.8	96.05	655.9
2 KB	367.9	1.887	7.12E-02	206	96.01	671.9
4 KB	399.8	2.557	8.24E-02	205.2	96.1	703.8
8 KB	463.8	2.054	7.66E-02	205.8	96.12	767.9

running on $\mathcal{P}$rv.

Timing Results: We evaluate the end-to-end sample by timing the protocol for a varied size of AER. When the end-to-end example runs with the benign input, four 256-Byte partial $\mathcal{CF}_{Log}$-s are generated and transmitted to $\mathcal{V}$rf. Table 4 presents the average runtime of each step in the protocol of Figure 4. The overall runtime of the protocol increases as the size of the AER increases since more time is required to attest a larger region of program memory. The remaining steps are completed in constant time as they do not depend on the size of AER – including the time to verify *ACFA* report due to the pre-computation of AER hash in $\mathcal{V}$rf offline phase. Steps 1-3 and Step 6 require the most time due to the communication delay.

Table 4 also shows the runtime of cryptographic computations used to authenticate messages. In Step 5, $\mathcal{V}$rf produces $Auth$ with an average runtime of $\approx$ 0.07ms. In Step 7, $\mathcal{P}$rv produces *out* with an average runtime of $\approx$ 96.1ms. Step 8 (either a call to *TCB*-Heal or return to *S*) requires negligible time. Thus, $\mathcal{P}$rv time to produce a MAC dominates the reported runtime of Steps 7-8.

This end-to-end example aims to illustrate the impact of more complex software on *ACFA*'s workflow. Including the receipt of network inputs, the password-check, and 6 sequential sensing operations, this sample application incurs $\approx 6,000$ control flow transfers. While *ACFA* guarantees are maintained as the complexity of applications increases, the number of communication rounds in *ACFA* pipeline (and associated overhead) also increases accordingly.

7 Related Work

RA: *RA* architectures fall into three categories: software-based, hardware-based, or hybrid. Software-based *RA* [36, 52–54] does not depend on specialized hardware modules and does not require any modifications to the existing hardware on a device. However, these approaches are limited due to their reliance on strong assumptions about adversaries' capabilities and timing requirements for the link connecting $\mathcal{V}$rf and $\mathcal{P}$rv. Hardware-based methods [9, 38, 43, 49] use dedicated hardware support either from external modules such as TPMs [60] or from the instruction set architecture, as in Intel SGX [15, 35]. *ACFA* leverages hybrid *RA* architecture VRASED [18] to implement a part of its active RoT for *CFA*, responsible for measuring the installed binary and authenticating $\mathcal{CF}_{Log}$ before a report can be sent to $\mathcal{V}$rf. In addition to VRASED [18], other hybrid *RA* approaches such as SMART [26] and TyTAN [28] use a combination of software and hardware for attestation. Typically, the hardware cost of hybrid approaches is substantially lower because they implement the *RA* measurement (e.g., MAC or signature) in software, while a hardware monitor is used to validate $\mathcal{P}$rv execution, ensuring the integrity of the *RA* execution and the secrecy of the *RA* cryptographic key(s). RealSWATT [57] is a recent software-based approach to continuously attest real-time and multi-core systems. It dedicates a core to attesting other applications continuously. In contrast, ACFA (and other hybrid architectures) targets single-core bare-metal MCUs, where RealSWATT would not apply. Different from the above-mentioned static *RA* methods, *ACFA* aims to support secure control flow *auditing*, in addition to attestation.

CFI Methods: CFI [1, 16] is a class of approaches (we include shadow stacks [10] and Address Space Layout Randomization (ASLR) [55] in this class) intimately related to *CFA*. While CFI has the similar goal of ensuring that a valid program path has been executed, *CFA* is more suitable for resource constrained devices. Compared to *CFA*, *CFI* does not provide $\mathcal{V}$rf with $\mathcal{CF}_{Log}$ and instead checks the control flow locally – on $\mathcal{P}$rv. In addition, many *CFI* methods rely on security capabilities that are usually expensive to low-end MCUs (e.g., MMUs). *CFA*, on the other hand, outsources the control flow verification to $\mathcal{V}$rf: a more resourceful trusted device. GRIFFIN [30] uses a shadow stack to restrict return targets. It also restricts indirect call sites and does not log static transfers, reducing storage requirements. Similar optimizations could be applied to *ACFA* to reduce $\mathcal{CF}_{Log}$ size. In general, CFI is considered challenging due to the hardness of associated sub-problems. For a discussion on CFI, see [58].

Runtime Attestation & CFA Methods: C-FLAT [2] was the earliest work on *CFA*. It relies on binary instrumentation along with hardware support from ARM TrustZone [7] to securely log control flow transfers in TrustZone's protected memory. At each instruction that alters the control flow (e.g., jump, branch, return), execution is trapped into the secure world and the control flow path taken is logged into protected memory. LO-FAT [23] and LiteHAX [24] are custom hardware-based approaches that improve upon C-FLAT by removing the need for binary instrumentation and by moving away from TrustZone. They introduce custom hardware support to hash branching instructions at runtime. As a result, instrumentation is no longer required and the runtime overhead (both execution time and code size) is reduced. However, an expensive hardware overhead is incurred due to the introduction of a hardware hash engine. Similar to C-FLAT, Tiny-CFA [21] also relies on instrumentation, but leverages cheaper hardware support from the Proof-of-Execution architecture APEX [19]. Therefore, it provides *CFA* at a relatively lower cost, making *CFA* amenable to low-end MCUs. Unlike prior architectures, Tiny-CFA constructs a verbatim log of control flow transitions, rather than computing a hash-chain. Therefore, Tiny-CFA is limited by the growth of $\mathcal{CF}_{Log}$ in relation to the amount of memory available to store it on $\mathcal{P}$rv. Nonetheless, this approach benefits from not requiring $\mathcal{V}$rf to enumerate all possible valid control flow paths. DIALED [41] builds upon Tiny-CFA to also provide Data Flow Attestation (DFA). Similarly, OAT [56] augments a variant of C-FLAT with DFA and provides optimizations to reduce the size of $\mathcal{CF}_{Log}$, when sent to $\mathcal{V}$rf verbatim. Compared to *CFA*, DFA [24,56] also detects "non-control data-only attacks" that corrupt intermediate data memory values during execution without affecting the program's control flow. While vulnerabilities that enable this type of attack are less common, they are still possible in specific cases (see [41] for examples).

Comparison of *ACFA* with Related Work: *ACFA* addresses key limitations of prior *CFA* methods. To the best of our knowledge, *ACFA* is the first *CFA* technique to support secure control flow auditing and remote remediation guarantees when control flow attacks are detected by $\mathcal{V}$rf. It also supports streamed reports that slice $\mathcal{CF}_{Log}$, making continuous *CFA* possible to large or infinite executions. In addition, *ACFA* implements the first hybrid approach to simultaneously obviate the need for instrumentation and minimize *CFA* hardware overhead. Table 3 presents a qualitative comparison between *ACFA* and prior *CFA* architectures. *ACFA* does not incur overhead due to instrumentation or hardware hash engines. It also constructs fixed size reports that are continuously streamed to $\mathcal{V}$rf. Finally, unlike prior *CFA*, *ACFA* supports active remediation when $\mathcal{V}$rf determines that $\mathcal{P}$rv has been compromised, as well as control flow auditing capabilities.

8 Conclusion

We designed, implemented, and evaluated *ACFA*: an inexpensive hybrid active *CFA* architecture that supports control flow auditing and guaranteed remediation of detected compromises. *ACFA* implementation is systematically de-constructed into sub-modules that jointly enforce *ACFA* required properties. Based on this set of properties, we argue *ACFA*'s security. *ACFA* public prototype (available at [12]) was implemented and synthesized on top of the low-end openMSP430 MCU.

Acknowledgments

We sincerely thank the paper's anonymous shepherd and the anonymous reviewers for their constructive comments and feedback. The first and third authors were 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). The second author was supported by the ASEAN IVO (www.nict.go.jp/en/asean_ivo/) project, Artificial Intelligence Powered Comprehensive Cyber-Security for Smart Healthcare Systems (AIPOSH), funded by NICT (www.nict.go.jp/en/).

References

[1] Martín Abadi et al. Control-flow integrity principles, implementations, and applications. *ACM TISSEC*, 13(1):1–40, 2009.

[2] Tigist Abera et al. C-flat: Control-flow attestation for embedded systems software. In *ACM CCS*, 2016.

[3] Fritz Alder et al. Aion: Enabling open systems through strong availability guarantees for enclaves. In *ACM CCS*, 2021.

[4] Esmerald Aliaj et al. Garota: Generalized active root-of-trust architecture (for tiny embedded devices). *USENIX Security Symposium*, 2022.

[5] Mahmoud Ammar et al. Simple: A remote attestation approach for resource-constrained iot devices. In *ACM/IEEE ICCPS*, pages 247–258, 2020.

[6] Orlando Arias et al. Hafix: Hardware-assisted flow integrity extension. In *DAC*, 2015.

[7] Arm Ltd. Arm TrustZone, 2018.

[8] Roberto Baldoni et al. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[9] Ernie Brickell et al. Direct anonymous attestation. In *ACM CCS*, 2004.

[10] Nathan Burow et al. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*.

[11] Claude Castelluccia et al. On the difficulty of software-based attestation of embedded devices. In *ACM CCS*, 2009.

[12] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. ACFA Github Repository. https://github.com/RIT-CHAOS-SEC/ACFA/, 2023.

[13] Adam Caulfield et al. Asap: reconciling asynchronous real-time operations and proofs of execution in simple embedded systems. In *DAC*, 2022.

[14] Abraham Clements et al. ACES: Automatic compartments for embedded systems. In *USENIX Security Symposium*, 2018.

[15] Victor Costan and Srinivas Devadas. Intel SGX explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[16] Crispan Cowan et al. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[17] Lucas Davi et al. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security '14*.

[18] Ivan De Oliveira Nunes et al. VRASED: A verified hardware/-software co-design for remote attestation. *USENIX Security Symposium*, 2019.

[19] Ivan De Oliveira Nunes et al. APEX: A verified architecture for proofs of execution on remote devices under full software compromise. In *USENIX Security Symposium*, 2020.

[20] Ivan De Oliveira Nunes et al. On the toctou problem in remote attestation. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2921–2936, 2021.

[21] Ivan De Oliveira Nunes et al. Tiny-cfa: A minimalistic approach for control flow attestation using verified proofs of execution. In *DATE*, 2021.

[22] Jyoti Deogirikar and Amarsinh Vidhate. Security attacks in iot: A survey. In *IEEE I-SMAC*, 2017.

[23] Ghada Dessouky et al. Lo-fat: Low-overhead control flow attestation in hardware. In *DAC*, 2017.

[24] Ghada Dessouky et al. Litehax: lightweight hardware-assisted attestation of program execution. In *IEEE/ACM ICCAD*, 2018.

[25] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[26] Karim Eldefrawy et al. SMART: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*. Internet Society, 2012.

[27] Karim Eldefrawy et al. HYDRA: hybrid design for remote attestation (using a formally verified microkernel). In *Wisec*. ACM, 2017.

[28] Ferdinand Brasser et al. Tytan: Tiny trust anchor for tiny devices. In *DAC*. ACM, 2015.

[29] Aurélien Francillon et al. Code injection attacks on harvard-architecture devices. In *CCS '08*, 2008.

[30] Xinyang Ge et al. Griffin: Guarding control flows using intel processor trace. *ACM SIGPLAN Notices*, 52(4):585–598, 2017.

[31] Jairo Giraldo et al. Integrity attacks on real-time pricing in smart grids: Impact and countermeasures. *IEEE Transactions on Smart Grid*, 8(5):2249–2257, 2016.

[32] Olivier Girard. openMSP430, 2009.

[33] Michele Grisafi et al. Pistis: Trusted computing architecture for low-end embedded systems. 2022.

[34] Manuel Huber et al. The lazarus effect: Healing compromised devices in the internet of small things. *arXiv preprint arXiv:2005.09714*, 2020.

[35] Intel. Intel Software Guard Extensions (Intel SGX).

[36] Rick Kennell et al. Establishing the genuinity of remote computer systems. In *USENIX Security*, 2003.

[37] Patrick Koeberl et al. TrustLite: A security architecture for tiny embedded devices. In *EuroSys*. ACM, 2014.

[38] X. Kovah et al. New results for timing-based attestation. In *IEEE S&P '12*, 2012.

[39] Ramya Jayaram Masti et al. Enabling trusted scheduling in embedded systems. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 61–70, 2012.

[40] Jo Noorman et al. Sancus 2.0: A low-cost security architecture for iot devices. *ACM TOPS*, 2017.

[41] Ivan De Oliveira Nunes et al. Dialed: Data integrity attestation for low-end embedded devices. In *DAC*. IEEE, 2021.

[42] Johannes Obermaier and Vincent Immler. The past, present, and future of physical security enclosures: from battery-backed monitoring to puf-based inherent security and beyond. *Journal of Hardware and Systems Security*, 2(4):289–296, 2018.

[43] Jr. Petroni et al. Copilot — A coprocessor-based kernel runtime integrity monitor. In *USENIX Security*, 2004.

[44] Lukas Petzi et al. Scraps: Scalable collective remote attestation for pub-sub iot networks with untrusted proxy verifier. *USENIX Security Symposium*, 2022.

[45] Md Masoom Rabbani et al. Reserve: Remote attestation of intermittent iot devices. In *SenSys*, 2021.

[46] Ganesan Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[47] Srivaths Ravi et al. Tamper resistance mechanisms for secure embedded systems. In *VLSID*. IEEE, 2004.

[48] Ryan Roemer et al. Return-oriented programming: Systems, languages, and applications. *ACM TISSEC*, 2012.

[49] Dries Schellekens et al. Remote attestation on legacy operating systems with trusted platform modules. *Science of Comp. Programming*, 2008.

[50] Seeed-Studio. Temperature Sensor Github Repository. `https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/temp_humi_sensor`, 2022.

[51] Seeed-Studio. Ultrasonic Ranger Github Repository. `https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/ultrasonic_ranger`, 2022.

[52] A. Seshadri et al. SWATT: Software-based attestation for embedded devices. In *IEEE S&P '04*, 2004.

[53] A. Seshadri et al. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SOSP*, 2005.

[54] Arvind Seshadri et al. SAKE: Software attestation for key establishment in sensor networks. In *DCOSS*. 2008.

[55] Hovav Shacham et al. On the effectiveness of address-space randomization. In *CCS*, 2004.

[56] Zhichuang Sun et al. Oat: Attesting operation integrity of embedded devices. In *IEEE S&P*, 2020.

[57] Sebastian Surminski et al. Realswatt: Remote software-based attestation for embedded devices under realtime constraints. In *CCS*, 2021.

[58] Laszlo Szekeres et al. Sok: Eternal war in memory. In *IEEE S&P*, 2013.

[59] Flavio Toffalini et al. Scarr: Scalable runtime remote attestation for complex systems. In *RAID 2019*, 2019.

[60] Trusted Computing Group. Trusted platform module (tpm), 2017.

[61] Jaikumar Vijayan. Stuxnet renews power grid security concerns. http://www.computerworld.com/article/2519574/security0/stuxnet-renews-power-grid-security-concerns.html, june 2010.

[62] Jo Vliegen et al. Sacha: Self-attestation of configurable hardware. In *DATE*. IEEE, 2019.

[63] Theo Walker. OpenSyringePump Github Repository. https://github.com/manimino/OpenSyringePump, 2022.

[64] Meng Xu et al. Dominance as a new trusted computing primitive for the internet of things. In *IEEE S&P*, 2019.

[65] Shaza Zeitouni et al. Atrium: Runtime attestation resilient under memory attacks. In *IEEE/ACM ICCAD*, 2017.

[66] Yumei Zhang et al. Recfa: Resilient control-flow attestation. In *ACSAC*, 2021.

APPENDIX

A Detecting Interrupts Accurately

It is possible that several instructions execute in the time between an interrupt being triggered and the CPU actually jumping to the associated ISR. Therefore, Branch Monitor tracks the *irq* signal to determine when an interrupt is triggered and the *gie* signal to determine the moment it is accepted. The signal $call_{irq}$ is set when this pattern is detected.

In *ACFA*, the signal $call_{irq}$ is an internal signal to Branch Monitor that is set by monitoring *irq* and *gie*. The $call_{irq}$ signal is controlled by the FSM shown in Figure 12 within Branch Monitor. When an interrupt is triggered, several cycles take place in order for the MCU to retrieve the address of the interrupt service routine and accept the interrupt. During the time between the interrupt being triggered and actually accepted, it is possible that multiple instructions are executed by the CPU. Therefore this FSM within Branch Monitor is crucial in order to determine the exact instruction that is the source of the transition.

Branch detection due to an interrupt is modeled as a three-state FSM with states *Wait*, *Pend*, and *Acc* with *Wait* being the initial state. A transition from *Wait* to *Pend* represents the moment a maskable interrupt (*irq*) or *ACFA*-specific non-maskable interrupt (*nmi*) due to **[T1-T3]** has been triggered. Then, a transition from *Pend* to *Acc* occurs when the interrupt is accepted, which is indicated by an internal signal irq_{acc}. After transitioning to *Acc*, $call_{irq}$ is set since a call due to an interrupt has occurred. Once this has been set, the third transition occurs from *Acc* to *Wait* and the flag is cleared. Since $call_{irq}$ causes $branch_{detect}$ to be set at this moment (per Figure 8), this allows the log entry for this interrupt to represent the exact instruction that the jump due to the interrupt occurred.

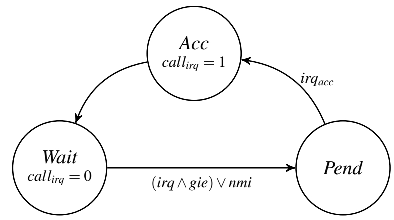


Figure 12: Branch Monitor FSM to detect branching due to an interrupt and support **[P3]** in *ACFA*.

HW Specification: Loop detection and counting

$$hw_{en} \wedge (ctr == 1) \rightarrow (src_{loop} = PC_{prev}) \wedge (dest_{loop} = PC) \tag{12}$$

$$hw_{en} \wedge (src_{loop} = PC_{prev}) \wedge (dest_{loop} = PC) \rightarrow (ctr++) \tag{13}$$

$$hw_{en} \wedge (src_{loop} \neq PC_{prev} \vee dest_{loop} \neq PC) \rightarrow (ctr = 1) \tag{14}$$

$$loop_{detect} = \begin{cases} 1, & \text{if } ctr > 1 \\ 0, & \text{otherwise} \end{cases} \tag{15}$$

Figure 13: Hardware Spec.: Loop Monitor Sub-module

B Loop Detection & Optimization Module

Figure 13 shows the hardware specification for accurately detecting a loop without internal branches (e.g., delay loops) and counting its iterations. Loop Monitor detects a loop based on the current *PC* and the previous PC (PC_{prev}). It also takes the output signal from Branch Monitor (hw_{en}) as input, which determines if a branch has been detected.

Whenever detecting a branch or $hw_{en} = 1$, Loop Monitor saves its source address to src_{loop} signal and its destination address to $dest_{loop}$. Loop Monitor then uses these signals to detect repeated jumps due to executing a loop. When repeated jumps happen $(PC_{prev}, PC) = (src_{loop}, dest_{loop})$, it increments an internal counter ctr to indicate the number of loop iterations that have occurred. When $ctr > 1$, the Loop Monitor sets $loop_{detect}$ to 1. When the loop execution is over or $(PC_{prev}, PC) \neq (src_{loop}, dest_{loop})$, Loop Monitor resets ctr.

The Loop Monitor ensures that all instances of loops are detected and their iterations are counted accurately. Thus, loops are logged to $\mathcal{CF}_{Log}$ efficiently and correctly.