TRACES: TEE-based Runtime Auditing for Commodity Embedded Systems

Adam Caulfield*, Antonio Joia Neto*, Norrathep Rattanavipanon† and Ivan De Oliveira Nunes*
*Rochester Institute of Technology, USA; †Prince of Songkla University, Thailand

***Abstract*—Control Flow Attestation (CFA) offers a means to detect control flow hijacking attacks on remote devices, enabling verification of their runtime trustworthiness. CFA generates a trace (CF_{Log}) containing the destination of all branching instructions executed. This allows a remote Verifier (Vrf) to inspect the execution control flow on a potentially compromised Prover (Prv) before trusting that a value/action was correctly produced/performed by Prv. However, while CFA can be used to detect runtime compromises, it cannot guarantee the eventual delivery of the execution evidence (CF_{Log}) to Vrf. In turn, a compromised Prv may refuse to send CF_{Log} to Vrf, preventing its analysis to determine the exploit's root cause and appropriate remediation actions.**

In this work, we propose *TRACES*: TEE-based Runtime Auditing for Commodity Embedded Systems. *TRACES* guarantees reliable delivery of periodic runtime reports even when Prv is compromised. This enables secure runtime auditing in addition to best-effort delivery of evidence in CFA. *TRACES* also supports a guaranteed remediation phase, triggered upon compromise detection to ensure that identified runtime vulnerabilities can be reliably patched. To the best of our knowledge, *TRACES* is the first system to provide this functionality on commodity devices (i.e., without requiring custom hardware modifications). To that end, *TRACES* leverages support from the ARM TrustZone-M Trusted Execution Environment (TEE). To assess practicality, we implement and evaluate a fully functional (open-source) prototype of *TRACES* atop the commodity ARM Cortex-M33 micro-controller unit.

***Index Terms*—Control Flow Attestation, Software Security, Embedded System Security.**

1. Introduction

Embedded devices have become ubiquitous and play critical roles within larger systems. These devices are typically implemented using resource-constrained micro-controller units (MCUs) that prioritize energy and space efficiency, as well as low cost. Due to these budgetary limitations, they lack security mechanisms commonly found in higher-end application computers, including Memory Management Units (MMUs), strong privilege separation, and inter-process isolation. Consequently, embedded devices tend to be more vulnerable to a wide range of attacks [1], [2], [3], [4], [5].

Remote Attestation (RA) [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22] has been proposed as an inexpensive means to remotely verify the software integrity of MCUs. RA is a challenge-response protocol wherein a trusted Verifier (Vrf) issues a cryptographic challenge and requests a timely response from a potentially compromised remote Prover device (Prv). In RA, a root of trust within Prv is responsible for producing evidence of Prv's software state by computing an authenticated integrity check (e.g. a MAC or signature) over the current snapshot of Prv's program memory and the received challenge. By examining the produced response message, Vrf can determine if Prv's software has been illegally modified.

Although classic RA can detect illegal program memory modifications, it cannot detect runtime attacks that do not modify code [23]. For instance, an adversary (Adv) could exploit a vulnerability (e.g. a buffer overflow) to hijack a program's control flow without modifying its code [24]. Consequently, Adv could execute a malicious code sequence (e.g., Jump-/Return-Oriented Programming – JOP/ROP – attacks [25]) and remain oblivious to RA.

Control Flow Attestation (CFA) [26] was introduced to augment classic RA evidence to include an authenticated trace (denoted CF_{Log}) of the attested software's most recent execution. CF_{Log} contains all control flow transfers executed (due to branching instructions such as `jumps`, `returns`, `calls`, etc.), allowing Vrf to learn the exact path followed (see Sec. 2.3 for more details on CFA). CF_{Log} is usually generated by instrumenting each branching instruction in the attested binary [23], [27], [28], [29] or by using custom hardware to detect and store the source/destination of branching instructions [30], [31], [32], [33]. As custom hardware is not yet present on commodity devices, currently deployable CFA leverages Trusted Execution Environments (TEEs) along with binary instrumentation.

Unfortunately, existing TEE-based CFA techniques are not able to guarantee that CF_{Log} is received by Vrf. Although an absence of responses containing CF_{Log} leads Vrf to distrust Prv (since an honest Prv would respond), it does not support remotely auditing the compromising behavior on Prv. *ACFA* [33] has recently acknowledged this problem and proposed hardware modifications to existing MCU architectures to ensure the reliable delivery of runtime evidence to Vrf and to facilitate Vrf-triggered remediation. However, because ACFA relies on custom hardware extensions, its guarantees cannot be realized until new MCU chips are fabricated. Consequently, no existing technique is directly deployable in today's commercial "off-the-shelf" MCUs. Our work aims to resolve this conflict by making

the following contributions:

- We propose *TRACES*, the first design realizing *secure runtime auditing* on off-the-shelf MCUs. *TRACES*'s TEE-based approach combines a CFA Engine and a Supervisor, both of which are implemented within TrustZone's Secure World. The former records control flow transfers to $\mathsf{CF}_{\mathsf{Log}}$, and the latter actively takes over Prv's execution to enforce reliable delivery of $\mathsf{CF}_{\mathsf{Log}}$ to Vrf. Additionally, *TRACES* supports Vrf-configured remediation if/when compromises are detected. As a consequence, our work demonstrates that runtime auditing and guaranteed remediation are achievable on commodity MCUs, featuring the TrustZone-M TEE. As *TRACES* proposes a software framework leveraging TrustZone-M, it requires a clean-slate design without overlapping architectural features with the prior work [33], while achieving equivalent security guarantees.
- We implement and evaluate a fully functional and open-source prototype of *TRACES* (available at [34]) using the well-known ARM Cortex-M33 MCU. *TRACES* prototype realizes runtime auditing/guaranteed remediation and is accompanied by sample use cases targeting on-demand sensing/actuation settings. We also conduct a systematic security analysis, performance evaluation based on several embedded programs, and an empirical evaluation of *TRACES* under exemplary exploits.

2. Background

2.1. TrustZone for ARM Cortex-M MCUs

TrustZone for ARM Cortex-M (TrustZone-M) [35] is an architectural security extension available on ARM V8 MCUs. It implements a TEE by isolating hardware and software resources on the MCU between two worlds, namely the "Secure" and "Non-Secure" Worlds [36]. TrustZone-M hardware controllers isolate the two worlds so that both code and data in the Secure World are immutable and inaccessible to the Non-Secure World. The Secure World code can only be called by the Non-Secure World from well-defined entry points, called Non-Secure-Callables (NSC). This mechanism enables controlled invocation and trustworthy execution of security-critical code, as well as the safe storage of private data within the Secure World, even when the Non-Secure World code (in this case, untrusted MCU applications) is fully compromised.

TrustZone-M defines the security state of memory segments by configuring hardware controllers called the Secure Attribution Unit (SAU) and the Implementation-Defined Attribution Unit (IDAU) to enforce memory isolation between worlds [37]. IDAU is a fixed memory map defined by the manufacturer and it assigns a default minimal security level to a given set of addresses. SAU, on the other hand, can be configured by the Secure World code to further reserve additional parts of the address space to the Secure World.

Memory accesses are first checked according to the security attribution defined by SAU, then are checked by the Memory Protection Unit (MPU). With TrustZone security extension, the MPU is segmented into Non-Secure and Secure states, effectively establishing one MPU for each world. The Secure MPU registers are only accessible to the Secure World, whereas the Non-Secure MPU (NS-MPU) registers are accessible to both worlds. MPU configuration registers are by default only accessible to software executing in privileged mode [38] and write access to MPU configuration registers can be revoked via the System Configuration Registers [39].

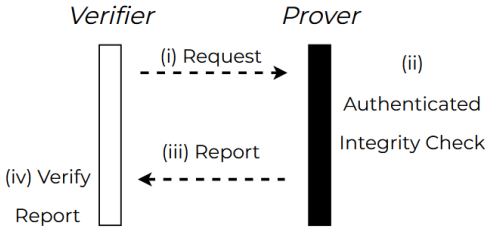

Figure 1: Typical RA interaction

TrustZone-M also supports the assignment of interrupts using separate Interrupt Vector Tables (IVTs) for the Secure and Non-Secure Worlds. IVTs are managed by the Nested Vector Interrupt Controller (NVIC). Each interrupt can be assigned as Secure or Non-Secure by setting the Interrupt Target Non-Secure (NVIC_ITNS) register [40], which is only configurable by the Secure World code. All secure interrupts have higher or equal priority than non-secure interrupts. When a secure interrupt is triggered while the CPU is executing in the Non-Secure World, the CPU pauses its execution, fetches the address stored in the Secure IVT, and transfers execution to the Secure World Interrupt Service Routine (ISR) pointed by this address. The context of the interrupted task is pushed onto the Non-Secure stack and popped upon return from the interrupt to the Non-Secure World.

2.2. Remote Attestation (RA)

RA is a challenge-response protocol in which Vrf aims to determine whether a remote Prv is installed with the expected software image. As depicted in Fig. 1, RA usually is composed of the following steps:

1) Vrf sends an attestation request to Prv containing a cryptographic challenge $Chal$.
2) A root of trust on Prv produces a report H by computing an authenticated integrity-ensuring function over Prv's own program memory and $Chal$.
3) Prv transmits H to Vrf.
4) Vrf compares H against the expected value to determine if Prv is in a trustworthy state.

The authenticated integrity-ensuring function in step 2 above can be implemented as a Message Authentication Code (MAC) or a digital signature. The secret key used in this computation must be securely stored by Prv's root of trust to ensure that it is immutable and inaccessible to any untrusted (potentially compromised) software on Prv.

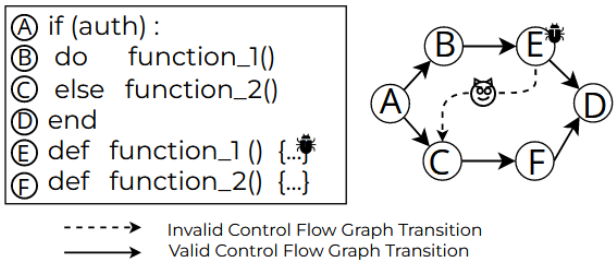

Figure 2: Illustration of a control flow attack

Therefore, secure storage for the RA secret key implies some level of hardware support (e.g., from TEEs, as in this work).

2.3. Control Flow Attestation (CFA)

Control flow attacks [41], [42], [43] aim to alter a program's intended control flow by executing unintended sequences of instructions. To illustrate these attacks, we refer to Fig. 2 which shows a simple exemplary program and its control flow graph (CFG). In benign executions, there are two valid paths in the program's CFG: $\{A, B, E, D\}$ or $\{A, C, F, D\}$ depending on the variable `auth`. However, suppose a memory safety vulnerability (e.g., a buffer overflow[24]) exists in `function_1()` implementation (i.e., node E). In that case, it can be exploited to illegally overwrite `function_1()`'s return address in the stack modifying the expected return site (node D) to an arbitrary address chosen by Adv – the call to `function_2()` (node C) in this example. As a consequence, the illegal sequence of nodes $\{A, B, E, C, F, D\}$ would be executed instead, even though the direct transition from the E to C does not exist in the program's CFG. More sophisticated control flow attacks – such as return-oriented programming (ROP) [25] and jump oriented programming (JOP) [44]) – can chain multiple such illegal control flow transfers to trigger arbitrary (often Turing-complete) behavior without modifying the program's code. In turn, these attacks cannot be detected by classic RA protocols.

CFA [6], [28], [27], [31], [32], [45], [46], [23], [47] augments RA reports to enable detection of control flow attacks. In addition to proving whether the correct software image is installed on Prv, CFA also produces a trace informing Vrf of the order in which the program's instructions have executed. This trace consists of an authenticated log ($\mathsf{CF}_{\mathsf{Log}}$) of all control flow transfers that occurred during a program's execution. $\mathsf{CF}_{\mathsf{Log}}$ is produced at runtime and stored in protected memory. Existing CFA techniques use either (1) binary instrumentation along with TEE support; or (2) custom hardware modifications to generate $\mathsf{CF}_{\mathsf{Log}}$ by detecting and saving each branch destination to hardware-protected memory. Upon receiving $\mathsf{CF}_{\mathsf{Log}}$, Vrf can inspect it alongside the attested software image to detect control flow attacks. For instance, in the attack example of Fig. 2, the illegal sequence $\{A, B, E, C, F, D\}$ would appear on $\mathsf{CF}_{\mathsf{Log}}$ and therefore Vrf would distrust this malicious execution.

In the case of TEE-based CFA (i.e., the class of CFA approaches applicable to existing devices without requiring custom hardware modifications) TrustZone's Secure World is used as a root of trust to build and store $\mathsf{CF}_{\mathsf{Log}}$. The binary to be attested is instrumented so that all branch instructions (e.g., `jumps`, `returns`, `calls`, etc.) are prepended with additional TrustZone calls that trap execution onto the Secure World. Once in the Secure World, $\mathsf{CF}_{\mathsf{Log}}$ is updated to reflect the correspondent control flow transfer.

Once execution completes, Prv authenticates $\mathsf{CF}_{\mathsf{Log}}$ and the installed software image (as in typical RA) to produce the attestation response (e.g., by computing a MAC or signature using the attestation secret key). Finally, Prv transmits $\mathsf{CF}_{\mathsf{Log}}$ to Vrf along with the produced authentication token (i.e., the MAC/signature result). In possession of the attested binary, $\mathsf{CF}_{\mathsf{Log}}$, and their authenticator, Vrf can determine if Prv execution occurred as intended and detect control flow attacks (in addition to binary modifications).

3. TRACES Overview

As noted earlier, CFA cannot guarantee that $\mathsf{CF}_{\mathsf{Log}}$ is received by Vrf when Prv is compromised. While this suffices to detect compromises (in general, the absence of a response indicates that something is wrong), it does not enable *auditing* of $\mathsf{CF}_{\mathsf{Log}}$ to pinpoint the source of compromises (i.e., to determine what is wrong). The latter property is non-trivial to obtain since a compromised Prv might ignore the protocol and refuse to send back attestation responses indicating a compromise (and its root cause).

TRACES is designed to guarantee that Vrf eventually receives runtime reports containing the information about Prv execution. *TRACES* also supports guaranteed healing of Prv when a compromise is detected. In contrast with prior related work [33], *TRACES* does not require custom hardware modifications to the MCU, enabling control flow auditing in existing (*"off-the-shelf"*) devices.

TRACES targets auditing on-demand sensing/actuation operations denoted App-s, where Vrf requests the execution of one particular App on Prv, at a given time. It extends the traditional "Vrf `sends request` → Prv `executes requested operation` → Vrf `receives result`" paradigm to enable control flow auditing (and potential remediation) of each requested operation. *TRACES* implements a Secure World-resident software monitor. Any attested App is untrusted and thus executes in the Non-Secure World.

3.1. Goals

Runtime Auditing: *TRACES* guarantees that a compromised App on Prv cannot interfere with the generation or transmission of a runtime report. It also ensures that each report is reliably received by Vrf, periodically retransmitting a report until a subsequent confirmation message is received from Vrf. In line with prior TEE-based CFA, *TRACES* instruments App binary to construct $\mathsf{CF}_{\mathsf{Log}}$ by logging all non-deterministic control flow transfers during

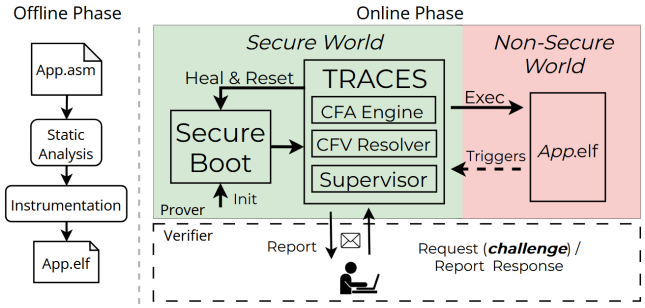

Figure 3: High level illustration of *TRACES* design.

App execution. Each runtime report sent to Vrf contains an authenticated CF_{Log} of App's execution (or a partial slice of CF_{Log}, if the memory region reserved to store CF_{Log} fills up during App's execution). By inspecting the received CF_{Log}, Vrf can audit Prv's runtime execution and pinpoint the exploit source when an attack is detected.

Guaranteed Remediation: *TRACES* Secure World implementation retains control over Prv execution after sending a report and until a response is received by Vrf. If the response indicates that no exploit was found, the Non-Secure World execution simply resumes. If Vrf responds with a remediation request, *TRACES* executes a Secure World-resident remediation function immediately. Examples include: wiping all data memory, shutting down Prv, or updating its software (i.e., its Non-Secure World program memory section containing the vulnerable App). Remediation actions are configurable by Vrf according to a desired security policy.

3.2. Architecture at a High-Level

As illustrated in Fig. 3, *TRACES* consists of three Secure World-resident software modules: **(i)** the CFA engine, **(ii)** the Control Flow Violation (CFV) Resolver, and **(iii)** the Supervisor. CFA engine is responsible for maintaining CF_{Log}. CFV Resolver implements Vrf desired remediation action. Lastly, the Supervisor acts as a controller within *TRACES*. It handles the transitions between different *TRACES* actions and enforces the required security properties while executing each of these actions.

Before deployment, App's binary is instrumented with a call to CFA Engine at each non-deterministic branching instruction (note that the presence of expected instrumentation is conveyed to Vrf as part of *TRACES* responses – see below). At boot, *TRACES* workflow enforces that the Supervisor is always the first software to run on Prv. It performs boot-time configurations to restrict the Non-Secure World's access to security-critical resources and waits for an initial Vrf authorization to initialize the Non-Secure World execution.

At runtime, upon receiving a Vrf-issued request for the attested execution of some App, the Supervisor measures App's binary (by hashing the Non-Secure World's program memory), configures the measured region as read-only (to prevent code modifications after the initial measurement), disables Non-Secure World interrupts, and initiates App's execution in the Non-Secure World. During App execution, CF_{Log} is continually appended with control flow transfers due to the instrumented CFA Engine calls. Aside from CFA Engine calls, after App execution is initiated, *TRACES* Secure World implementation acts upon three events, referred to as triggers [T1], [T2], and [T3] defined as:

[T1]: A predefined time limit for periodic communication with Vrf has been reached. This guarantee is obtained by leveraging the NVIC controller (recall Sec. 2) to assign a timer-based interrupt to the Secure World.

[T2]: The execution of App has concluded. [T2] is obtained with a TrustZone-protected return instruction to the Secure World.

[T3]: The memory region reserved to store CF_{Log} is full and the current CF_{Log} values need to be sent to Vrf before new control flow transfers can be added. To obtain [T3], *TRACES* checks if CF_{Log} designated memory is full after appending each new control flow transfer.

[T1] is implemented as a Secure World-protected interrupt. [T2] is a secure (i.e., TrustZone-protected) return to Secure World that is measured by App's hash and immutable after that stage. [T3] is implemented within the Secure World code. Therefore, given TrustZone guarantees, these triggers cannot be disabled by the Non-Secure World. Furthermore, the Secure World execution, once triggered, cannot be interrupted by the Non-Secure World. Any of the three triggers results in a call to *TRACES* to generate a signed runtime report containing the current snapshot of CF_{Log} as well as the hash of App's binary. The Supervisor sends the report to Vrf for analysis and retains control over Prv in the Secure World until an authenticated acknowledgment response is received from Vrf.

Remark: *note that the hash of* App*'s binary is sent along with every report, allowing* Vrf *to also ascertain* App*'s binary integrity. This implies the presence of the expected code instrumentation responsible for generating* CF_{Log}*. See protocol details in Sec. 4.*

While waiting for Vrf's response, the produced report is re-transmitted periodically to cope with eventual network losses. If Vrf response indicates that a compromise was detected, Supervisor invokes CFV Resolver for remediation. If no compromise is indicated, Supervisor simply resumes App execution from where it left off when the trigger occurred. This process continues while App executes (i.e., until Vrf receives a report issued due to [T2]).

3.3. TRACES Security Intuition.

The intuition for *TRACES*'s security (in achieving auditing and guaranteed remediation capabilities) follows from the facts that: **(1)** triggers [T1], [T2], and [T3] cannot be disabled by the Non-Secure World and **(2)** the Supervisor-enforced workflow cannot be disrupted by the Non-Secure World. The workflow assures that Vrf always receives reports and can act upon them before execution of untrusted software in Prv is resumed. In Sec. 4, we delve into the details of how these high-level ideas are obtained concretely.

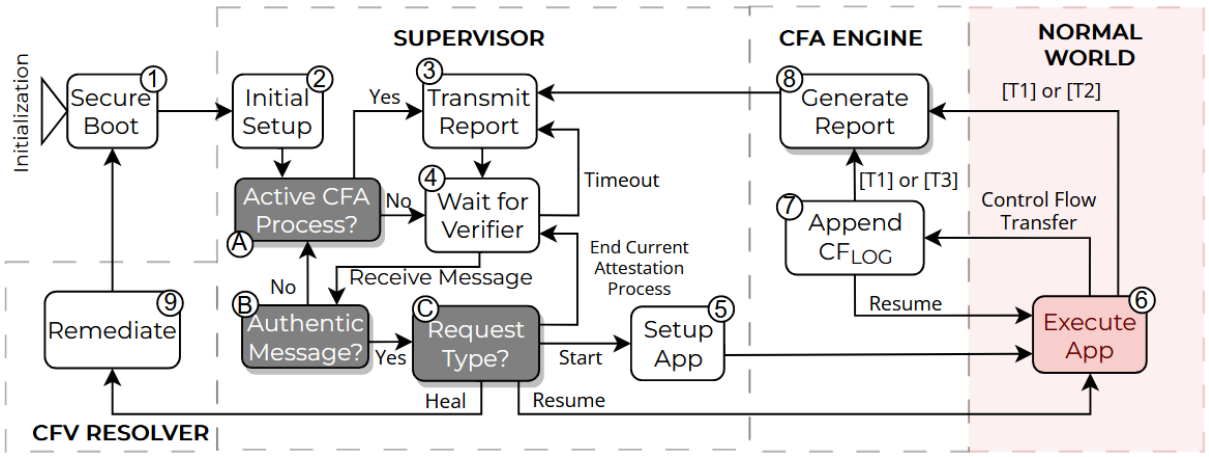

Figure 4: *TRACES* workflow.

In Sec. 5, we perform a systematic security analysis of *TRACES*.

4. TRACES in Detail

4.1. Scope and System Model

TRACES is focused on single-core, bare-metal MCUs equipped with TEEs, (TrustZone-M, in our case). Attested App-s execute in the Non-Secure World, while the Secure World contains *TRACES* trusted computing base (TCB). We rely on the following standard requirements from Prv's TEE (which are attainable through standard ARM TrustZone-M v8 architectural support [35]):

- Cryptographic keys are securely provisioned to Prv and Vrf prior to deployment. Vrf is trusted and Prv keys are stored within the Secure World (thus inaccessible to the Non-Secure World).
- The Secure World (code and data) is trusted and isolated from Non-Secure World (including App);
- Prv has separate IVTs for the Non-Secure and Secure Worlds. Interrupt sources can be assigned to either world and the Secure-IVT has priority over the Non-Secure-IVT.
- Prv has a Non-Secure Memory Protection Unit (NS-MPU) that controls access to Non-Secure World memory. (see Sec. 2.1).

Following the on-demand sensing/actuation regime discussed in Sec. 3, Vrf aims to audit the runtime behavior of one Non-Secure World App at a time. Nonetheless, Prv may have any number of App-s installed in its Non-Secure World program memory. Similar to any TEE-based CFA, the branching instructions of App must be correctly instrumented with additional instructions to log the control flow transfers. This instrumentation is performed at compile-time before App's deployment on Prv (which can also happen remotely). In addition, Vrf keeps a copy of the instrumented App binary (or hash thereof) to verify the received hash of App's binary included in Prv's response, as described in Sec. 2.2.

4.2. Adversary Model

Our Adv model is consistent with that of secure systems built atop TEEs. We consider Adv capable of fully compromising the Non-Secure World on Prv. Adv can exploit vulnerabilities to launch code-injection attacks, hijack App's control flow, or perform code-reuse attacks. In addition, Adv can manipulate Non-Secure World interrupts and their ISRs. Adv cannot tamper with code or data in the Secure World or circumvent access controls enforced by the TEE hardware.

Adv's ability to modify Non-Secure World code must be accounted for in *TRACES*'s design because Adv could use it to remove CFA-related instrumentation. *TRACES* leverages TEE controls along with temporally consistent code integrity measurements to ensure that any such attempt is detected by Vrf (see Section 4.3). This is in contrast to prior work [23], [27], [48] that rules out code modifications from their threat model by assumption (the latter also implies an inability to perform benign software updates at runtime [13]). Our design shows the latter requirement/limitation as unnecessary.

Invasive physical attacks that modify hardware are out-of-scope of this work, as they require an orthogonal set of physical security measures. For details see [49], [50].

4.3. *TRACES* Workflow

TRACES workflow is presented in Fig. 4 and this section details each step, including processes (① - ⑨) and decisions (Ⓐ - Ⓒ), along with details of how the workflow rules are enforced by *TRACES* design at each stage.

Initialization Routine: The first step in the Initialization Routine is a standard ① *Secure Boot* verification that ensures the integrity of the trusted software loaded onto the Secure World, i.e., *TRACES*'s implementation.

TRACES Supervisor module is the first to execute after secure boot. It performs the ② *Initial Setup* by reserving a memory region $SW_{MEM'}$, located within the Secure World, to store the runtime report. To complete initialization, Supervisor retrieves the runtime auditing context. This includes Secure World-resident memory regions that store

$\mathsf{CF_{Log}}$, $\mathsf{CF_{Log}}$ size, and other metadata related to the runtime auditing context. We note that in most cases, the context is empty at this stage. However, in some instances, it may contain values from a previous malicious execution that led the MCU to a reset (see below).

Waiting for App's Attested Execution Request: After completing all initialization tasks, the Supervisor determines if an active attestation process of some App was ongoing before boot (A). For instance, if a software fault occurred and led to a system reset during App execution, a report transmission (3) must occur next to notify Vrf of the runtime state that has led to the fault. If there is no previous operational context (Prv was inactive), Supervisor continues to (4) *Waiting for Vrf Message*. Upon receiving a message from Vrf, the Supervisor checks whether this message is timely and generated by Vrf (B) (via standard cryptographic authentication – see Sec. 4.4 for protocol details). Upon successful authentication, the Supervisor processes Vrf request (C). A request to start the attested execution of some App on Prv contains a unique challenge Chal used to ensure the freshness of the report to be generated by the App's attested execution. The Supervisor saves Chal and associated metadata within the Secure World before executing App.

Attested Execution: A new attested execution requires the Supervisor to (5) *Setup App* and perform the following actions before beginning execution.

- Setting the Non-Secure World program memory as Read/Execute-Only and its data memory as Non-Executable using the NS-MPU. This prevents unauthorized code modification and data execution at runtime.
- Revoking the Non-Secure World's ability to reconfigure NS-MPU. This is achieved in two steps: (1) by setting the System Configuration Registers to prevent accesses to the NS-MPU configuration registers; and (2) by using SAU to assign the System Configuration registers themselves to be accessible only by the Secure World. After this stage, both the NS-MPU and System Control registers are inaccessible to any Non-Secure World code.
- Configure [T1] as a secure timer interrupt to activate after δ clock cycles, with a higher priority than all other interrupts. As a Secure World interrupt, [T1] cannot be disabled or misconfigured by untrusted code in the Non-Secure World. This timer resets every time a report is generated and is deactivated at the end of App's attested execution. The value of δ can be either fixed to a default or chosen by Vrf within the attested execution request.
- Measure App's code by hashing the entire Non-Secure World's program memory to capture its state immediately before execution. The hash result (H_{PMEM}) is stored in Secure World memory.
- Initialize CFA engine metadata. This includes setting the CFA status flag as "active".

Once all configurations are set, the Supervisor (6) *Executes App* in the Non-Secure World. During App's execution, CFA Engine is invoked by App instrumented instructions whenever a new control flow transfer occurs to (7) *Append* $\mathsf{CF_{Log}}$. Each instrumented instruction appends the current control flow transfer to $\mathsf{CF_{Log}}$ and then resumes App execution (6) as long as $\mathsf{CF_{Log}}$ is not full. If $\mathsf{CF_{Log}}$ is full, CFA Engine triggers [T3] to transmit a new report to Vrf (8) before additional transfers can be added.

Reliable Runtime Report & Vrf Response: When either trigger [T1], [T2], or [T3] is activated, CFA Engine must (8) *Generate a Report* to be sent to Vrf. To ensure atomic execution, all interrupts are disabled as the first step of (8). Moreover, the timer for [T1] is cleared and paused until App execution is resumed. The report is cryptographically authenticated (see protocol details in Sec. 4.4) and includes Chal, H_{PMEM}, $\mathsf{CF_{Log}}$, its size, and outputs (e.g., sensed values) produced by App execution (if any).

After computing the report, Supervisor (3) *Transmits the Report* and waits for a response (4). While Prv waits, Vrf is expected to: receive the report and verify its authenticity, validate $\mathsf{CF_{Log}}$ and H_{PMEM}, and respond to Prv based on this analysis. During this waiting period in (4), the Supervisor enters sleep mode and periodically wakes up to re-transmit the report in (3). This periodic re-transmission is necessary to ensure that the report eventually reaches Vrf, despite occasional network failures or network denial of service attempts. Upon receiving a response from Vrf, Prv proceeds to perform checks (B) and (C).

If a violation is detected in the report, Vrf message includes a *Heal* request (in (C)), causing Supervisor to invoke the remediation routine managed by CFV Resolver in (9). In the absence of control flow violations, there are two cases. If the previous report was generated by [T2], indicating that App has completed its execution, Vrf response instructs the Supervisor to stop the current attestation process and return to the idle waiting stage (4). To do so, Supervisor changes the CFA status flag to "inactive" and waits for a new attested execution request from Vrf. Otherwise, App's attested execution is resumed (6).

CFV Resolver: When Vrf message contains a request to *Heal* Prv, the CFV Resolver is invoked following (C) to execute the (9) *Remediation Procedure*. This module contains the Vrf-defined remediation action that can accommodate a variety of policies. For instance, erasing all data memory, updating App's binary to patch the vulnerability, or shutting Prv down. By default, this action is followed by a system reset to ensure that the system reboots in the newly configured state properly. All non-maskable interrupts and hardware fault exceptions that could otherwise preempt remediation actions are configured to reset Prv, thereby triggering the Supervisor execution at reboot and resuming the remediation action in (9).

As mentioned earlier, if Prv resets (1) during App's attested execution (e.g., due to faults or software bugs) the remnants of $\mathsf{CF_{Log}}$ must be sent to Vrf for auditing (e.g., to pinpoint the exploit causing the reboot). In this case, the CFA process remains active when Supervisor reaches (A). At this point, Supervisor continues to (3) instead of (4) and re-transmits the report containing the remnants of $\mathsf{CF_{Log}}$.

4.4. TRACES Protocol

Based on the workflow defined in Sec. 4.3, we specify the interaction between Vrf and Prv in Protocol 1. This protocol starts when Prv receives a request from Vrf.

In Step 1, the Supervisor hashes and locks (prevents writes to) the Non-Secure World program memory ($PMEM$) to produce H_{PMEM}. In Step 2, *TRACES* starts executing App. $\mathsf{CF_{Log}}$ is appended whenever a control flow transfer happens within App. Upon any of the triggers [T1], [T2], or [T3], Prv proceeds to Step 3.

In Step 3, *TRACES* attests Prv state by computing a MAC (σ_{Prv}) on H_{PMEM} and $\mathsf{CF_{Log}}$ using a pre-shared symmetric key (an asymmetric version of the protocol can be obtained in the standard way, by replacing the MAC operations by signatures). The runtime report (R_P), which includes σ_{Prv}, $\mathsf{CF_{Log}}$ and Log_{Size}, is then created and sent to Vrf in Step 4. Upon receiving R_P in Step 5, Vrf proceeds to:

1) validate σ_{Prv} by checking whether it was computed over the latest challenge Chal, the hash of the expected binary $PMEM'$, and the received $\mathsf{CF_{Log}}$.
2) analyze App execution using the received $\mathsf{CF_{Log}}$. Vrf can complete this step using several techniques, e.g., determining whether $\mathsf{CF_{Log}}$ matches a valid path in App's CFG, emulating a shadow stack of App's execution, and more. Sec. 7.3 elaborates on verification possibilities.

Based on this analysis, in Step 6, Vrf produces an output (vrf_{result}) to indicate the verification result and next action. In Step 7, Vrf issues a fresh challenge Chal' and creates an authentication token (σ_{Vrf}) by MAC-ing Chal' and vrf_{result} and transmits their response R_V to Prv in Step 8.

In Step 9, Prv receives and parses R_V. Next, in Step 10, Prv verifies R_V by checking whether the MAC σ_{Vrf} is valid and if σ_{Vrf} was computed on a fresh challenge. This step produces a verification output out; when $out = \texttt{False}$, Prv disregards the response, continues to wait, and repeats Step 4 until it receives an authenticated message from a valid Vrf.

Otherwise, Prv updates its own persistent copy of the latest challenge to the newly received Chal' and examines vrf_{result} to decide on the next course of action. If Vrf disapproves the report ($vrf_{result} = \texttt{Heal}$), Prv invokes the CFV Resolver to execute the remediation (Step 11) and subsequently restarts the system. Conversely, when Vrf approves the report ($vrf_{result} = \texttt{Exec}$ or $\texttt{End}$), Prv transfers control back to the Non-Secure World to resume or end App's execution.

Protocol 1 - *TRACES* Protocol

NOTATION:
- $PMEM$: Prv's Non-Secure World program memory.
- $PMEM'$: Expected Prv's Non-Secure World program memory.
- Log_{size}: Size of $\mathsf{CF_{Log}}$.
- $\mathtt{h}$: A secure cryptographic hash function.
- $\mathtt{MAC}_K$: Compute MAC using key K.
- $\mathtt{Verify}_K$: MAC verification using public key K.
- k: key pre-shared between Prv and Vrf
- Chal: Challenge based on a (persistent) increasing counter.

PROTOCOL:

Prover (Prv) Secure World

1. Generate hash of and lock $PMEM$ (executed before App's execution.)
$$H_{PMEM} := \mathtt{h}(PMEM)$$
2. Execute App in the Non-Secure World. During App's execution, the Secure World is invoked to append $\mathsf{CF_{Log}}$ whenever a control flow transfer happens.
3. Upon a trigger, compute MAC:
$$\sigma_{\mathsf{Prv}} := \mathtt{MAC}_k(H_{PMEM}, Log_{size}, \mathsf{CF_{Log}}, \mathsf{Chal})$$
4. Send report R_P to Vrf with the following format:
$$R_P := (\sigma_{\mathsf{Prv}} || Log_{size} || \mathsf{CF_{Log}})$$
Wait for Vrf's response and re-transmit R_P periodically until the response is received.

Verifier (Vrf)

5. Receive R_P and extract $\sigma_{\mathsf{Prv}}, Log_{size}, \mathsf{CF_{Log}}$
6. Verify report :
$$vrf_{result} := \mathtt{Verify}_k(\sigma_{\mathsf{Prv}}, PMEM', Log_{size}, \mathsf{CF_{Log}}, \mathsf{Chal})$$
7. Increment $\mathsf{Chal}' := \mathsf{Chal} + 1$ and creates an authorization token based on this new challenge:
$$\sigma_{\mathsf{Vrf}} := \mathtt{MAC}_k(\mathsf{Chal}', vrf_{result})$$
8. Construct and send response R_V to Prv
$$R_V := (vrf_{result} || \mathsf{Chal}' || \sigma_{\mathsf{Vrf}}) \rightarrow \mathsf{Prv}$$

Prover (Prv) Secure World

9. Receive R_V and extract $vrf_{result}, \mathsf{Chal}', \sigma_{\mathsf{Vrf}} \leftarrow R_V$
10. Authenticate the response, producing a one-bit output:
$$out := \mathtt{Verify}_k(vrf_{result}, \mathsf{Chal}', \sigma_{\mathsf{Vrf}}) \texttt{ and } (\mathsf{Chal}' > \mathsf{Chal})$$
Based on out and vrf_{result}, it decides the next transition:
- If $out = \texttt{False}$: Re-enter Wait (Jump to Step 4)
- Else If $vrf_{result} = \texttt{Heal}$: Update local value of Chal to Chal' and enter Remediate state (Jump to Step 11)
- Else If $vrf_{result} = \texttt{Exec}$, update local value of Chal to Chal' and resume App (jump to Step 2)
- Else If $vrf_{result} = \texttt{End}$, end CFA process unlocking PMEM and concluding the protocol instance.

11. Execute remediation software and restart the system.

5. Security Analysis

Our Adv model (see Sec. 4.2) considers that Adv has full control over Prv's Non-Secure World. To circumvent *TRACES* guarantees Adv must (1) forge a report that is accepted by Vrf and does not correspond to the actual execution of App; (2) prevent Vrf from receiving a legitimate response (and $\mathsf{CF_{Log}}$ therein); or (3) prevent a Vrf-initiated remediation.

5.1. Report Forgery

A runtime report is considered trustworthy if it faithfully reflects the control flow of App's timely execution as well as its binary. Adv may attempt to manipulate the response message to deceive Vrf in the following ways:

App **Binary Modifications**. Adv may modify the Non-Secure World binary to remove or add instructions that generate $\mathsf{CF_{Log}}$ entries in an attempt to produce a valid $\mathsf{CF_{Log}}$ that differs from the true control flow of App's execution. However, Vrf can detect any modifications to App binary by checking H_{PMEM}, which is computed immediately before App execution. In between H_{PMEM} generation and the end

of App's execution, App's binary cannot be modified, as enforced using NS-MPU and SAU protections.

Control Flow Attacks. Adv may attempt to corrupt App's execution by exploiting memory safety vulnerabilities to cause a malicious sequence of control flow transfers without modifying App's binary. However, any such attempt must be reflected on $\mathsf{CF}_{\mathsf{Log}}$ (due to the instrumentation of all branching instructions) and thus visible to Vrf.

Interrupt Manipulation. An attacker could also leverage non-secure interrupts to stealthily modify App's control flow. By default, *TRACES* disables interrupts during App's execution. For real-time App-s that must process interrupts, this requirement can also be alleviated by leveraging interrupt-safety mechanisms for CFA, such as ISC-FLAT [29].

$\mathsf{CF}_{\mathsf{Log}}$ **Forgery.** Adv may attempt to directly forge $\mathsf{CF}_{\mathsf{Log}}$ by modifying the response message or the memory region storing $\mathsf{CF}_{\mathsf{Log}}$ on Prv. Since $\mathsf{CF}_{\mathsf{Log}}$ is append-only and stored in the Secure World, Adv cannot modify $\mathsf{CF}_{\mathsf{Log}}$ (or other *TRACES* data) in Prv's memory. In addition, attempts to modify or replay a response message are ineffective due to the use of an unforgeable cryptographic function (see below) computed on a fresh challenge (Chal) unique for every response message in the protocol.

Forgery of Attestation Result. Adv may attempt to forge the attestation result σ_{Prv}. However, this forgery is computationally infeasible without knowledge of the key given the security of the underlying cryptographic function. Finally, the key is stored in the Secure World, thus inaccessible to Adv.

5.2. Preventing Evidence Delivery

Since triggers [T1], [T2], and [T3] cause *TRACES* to generate and send a report, Adv must prevent them from occurring to block the delivery of evidence to Vrf. Adv can only avoid filling $\mathsf{CF}_{\mathsf{Log}}$ to its maximum size (hence triggering [T3]) by modifying the binary of App or by launching a control flow attack that jumps to an uninstrumented section of the Non-Secure program memory outside of App's binary. Although these measures prevent [T3], these attacks will always be reflected in the next report caused by triggers [T1] or [T2]. If $\mathsf{CF}_{\mathsf{Log}}$ does not fill up to its maximum size during App execution, App will eventually end and cause a trigger [T2] or a timeout [T1] (whichever comes first). Since the timer is configured as a secure interrupt, it is impossible for Adv to prevent [T1] because it is handled by the Secure World. The NVIC interrupt configuration is controlled by the Secure World and cannot be modified by the Non-Secure World.

TRACES implementation re-transmits evidence until an authenticated confirmation (and remediation request, if applicable) is received from Vrf. This guarantees that Vrf does not lose evidence due to network faults/attacks. However, it also prevents execution on Prv from resuming before verification is completed successfully, adversely affecting systems that rely on real-time response and time-critical actions. To cope with this, TRACES can be modified to impose a less-strict policy. This could work by allowing App to continue execution until the next [T1] trigger, when a new report must be transmitted to Vrf. If no receipt confirmation for the older report is received from Vrf, the new report would now contain both the old $\mathsf{CF}_{\mathsf{Log}}$ and new control flow transfers added since App's resumption. The less-strict verification policy, however, introduces a security trade-off. A compromised Prv could, in this case, execute malicious actions for a longer period, i.e., until the second [T1] trigger.

5.3. Preventing Remediation Actions

It follows from the atomic execution of the remediation action after communication with Vrf, that Adv cannot prevent it. *TRACES* ensures that Prv execution remains in the Secure World until it has received approval from Vrf to resume the Non-Secure World execution. In addition, Prv stays in the Secure World and attests to the result of any remediation action after its completion. The latter serves as confirmation to Vrf that the remediation action was executed properly. Although Prv may reset, perhaps in an attempt from Adv to prevent the remediation from taking place, the report transmission phase is always re-initiated after any reset, and it is eventually followed by `remediation` (recall Fig. 4). As a result, if a reset indeed occurs, malware in the Non-Secure World is prevented from executing until the remediation phase is completed successfully.

6. Implementation Details

6.1. Instrumentation.

Prior to deployment, App's assembly is instrumented to redirect branches to a trampoline function in the NSC that calls the CFA Engine. We implement trampoline functions for, indirect calls, conditional branches, and returns.

Conditional Branch Instructions. Since there are two possible destinations of a conditional branch ("taken" and "not-taken"), a call to the trampoline function (via `bl`) is inserted in both places. Then after calling the trampoline, the link register `lr` holds the branch's destination address. The trampoline passes `lr` to the CFA Engine for logging. Afterwards, the CFA Engine executes `bxns lr` instruction to return to the branch destination. When a conditional branch occurs due to a static loop (i.e., a loop with no internal branching), it can be instrumented differently to optimize the logging. A loop is detected in App's assembly by locating a "backward" non-linking branch instruction (i.e., a conditional branch instruction whose "branch-taken" destination precedes the instruction itself) [32]. Once located, the register that is incremented (`ri`) and the register (or value) used as the limit for comparison (`rL`) are identified. Then, the instruction that initializes `ri` is located to determine the loop entry. At the entry, three instructions are then added. First, the loop's "branch-taken" destination is loaded into a reserved register $\mathtt{rr}_0$. Next, the value from `rL` is loaded into a second reserved register $\mathtt{rr}_1$. Lastly, a `bl` to a loop

TABLE 1: App before and after instrumentation

Application Information	Sensor Applications					BEEBS Programs [51]		
	Ultrasonic [52]	Geiger [53]	Syringe [54]	Temperature [55]	GPS [56]	prime	crc32	sglib-arraybinsearch
Total Instructions	82	223	152	157	1200	133	57	97
Instructions post-instrumentation	91	238	172	174	1389	146	61	116
Task runtime (ms)	0.2	0.2	0.8	0.3	0.4	0.9	1.2	0.6
Instrumented task runtime (ms)	0.4	1.0	1.8	1.6	3.0	5.8	9.4	13.4
Control Flow Transfers (at runtime)	1015	744	654	607	649	1304	2051	3225
Generated CFLog size (Bytes)	56	186	1400	1212	2596	5216	8204	12900

trampoline function is inserted. The loop trampoline reads from $\texttt{rr}_0$ and $\texttt{rr}_1$ to append $\mathsf{CF}_{\mathsf{Log}}$ with the destination address and the limit of the loop, respectively. This optimization significantly reduces the number of calls to CFA Engine (to just 1) to log all static loop iterations. The loop exit is instrumented like any conditional branch "not-taken" destination. If the loop is not static or `rL` is modified in the loop, the conditional branch is instrumented as described earlier, and $\mathsf{CF}_{\mathsf{Log}}$ optimization occurs in the CFA Engine. In this case, CFA Engine treats repeated backward edges as loops and increments an internal loop counter instead of logging the repeated address. When the loop exits, the counter is logged.

Indirect Call Instructions. Indirect calls are of the form `blx rx`, which calls the address stored in a register `rx`. To save the value of `rx`, the trampoline function for indirect calls uses the reserved register $\texttt{rr}_0$. Assume an instruction `blx rx` indirectly calls the function `func`. This instruction in App is replaced with two instructions: first, an instruction to load `rx` into $\texttt{rr}_0$; then, a `bl` to the trampoline is inserted. When reaching the trampoline, $\texttt{rr}_0$ holds `func`'s address, and `lr` holds `func`'s return address. The trampoline then passes the value of $\texttt{rr}_0$ to the CFA engine to update $\mathsf{CF}_{\mathsf{Log}}$. After appending $\mathsf{CF}_{\mathsf{Log}}$, the trampoline returns to App via `bxns` $\texttt{rr}_0$ and resumes App at the first instruction of `func` while preserving `func`'s return address in `lr`. Our implementation reserves `r10` as $\texttt{rr}_0$ and `r11` as $\texttt{rr}_1$. All other instances of `r10` and `r11` in App assembly are replaced with different general-purpose register before recompilation.

Return Instructions. There are two scenarios for returns from a function in App's assembly. If the function performs no other calls, the return is implemented as `bx lr`. If it contains a call, `lr` must be pushed onto the stack, and thus the return is implemented as `pop pc`. In the first case, we replace `bx lr` with a direct branch (via `b`) to the trampoline. A direct branch ensures the return address in `lr` is not modified. After logging `lr`, the trampoline returns via `bxns lr` to the proper destination. In the second case, `pop pc` is replaced with two instructions: a `pop lr` followed by a direct branch (via `b`) to the trampoline. The logging and return are then performed in the same manner as the first scenario.

6.2. Module Configurations:

In our prototype, the Supervisor communicates with Vrf using a UART-to-USB connection. The available LP-UART interface is configured at `baud` rate of 921600 bps. The Supervisor also reserves the MCU's Timer #3 to the Secure World to implement [T1] with a deadline of 5 seconds and sets the maximum $\mathsf{CF}_{\mathsf{Log}}$ size to 50 KBytes. CFA Engine implementation uses SHA256 and HMAC-SHA256 from HACL* [57] formally verified library for hash and MAC computations. We use default parameters with 256-bit keys, and 512-bit Chal. In our prototype, CFV Resolver implements three simple remediation actions: freeze Prv execution (i.e., run an infinite loop); disable the compromised App; and wipe the compromised App from the Non-Secure World.

The memory region $SW_{MEM'}$ that stores the runtime report must be recoverable through software resets, e.g., by assigning $SW_{MEM'}$ to a persistent memory location. Alternatively, some ARM Cortex-M devices with two SRAM segments allow one segment to be retained when the internal voltage regulator powers off as a feature of the lowest-power mode [39]. In this work, we configure SRAM2 to retain the runtime report through software resets.

7. Prototype Evaluation

We implement and test *TRACES* using a NUCLEO-L552ZE-Q development board shown equipped with an STM32L552ZE MCU based on the ARM Cortex-M33 (v8) operating at 110 MHz. The MCU supports ARM TrustZone-M. We evaluate *TRACES* usage on a set of open-source sensor applications[1] and on programs from the BEEBS benchmark suite [51], detailed in Table 1. Additionally, we implement all Vrf's operations in Python. *TRACES* TCB is implemented with 2383 lines of C code. The Supervisor (including the NSC) accounts for 1250 lines, the CFA Engine – 861 lines (including SHA256 and HMAC formally verified implementations), and the CFV Resolver – 272 lines. In total, *TRACES* TCB requires 30.8 KBytes of program memory.

Table 1 details the tested applications. For the evaluated applications, instrumentation alone adds 0.2-12.8ms of runtime overhead. Due to the lack of custom hardware to detect branches, the same instrumentation is required in any TEE-based CFA, irrespective of *TRACES* added guarantees. Repeated loops with internal branching cause more significant increases in programs like *prime*, *crc32*, and *sglib-arraybinsearch* (abbreviated *search*). Table 1 also shows the number of control flow transfers in each application execution. We note, however, that this number does not always reflect the size of $\mathsf{CF}_{\mathsf{Log}}$ due to *TRACES* optimization to replace static loops with counters (recall Sec. 6).

1. Some applications required small modifications (ports) to run on ARM Cortex-M.

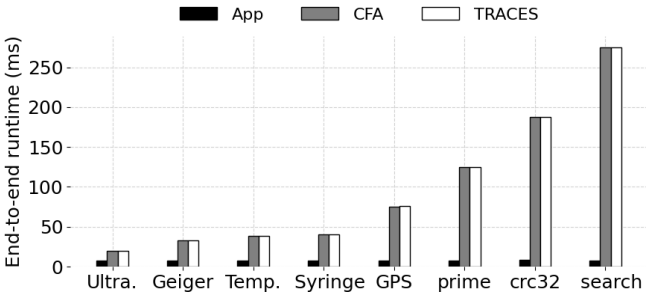

Figure 5: End-to-end runtime comparison: baseline App, best-effort CFA, and runtime auditing with *TRACES*.

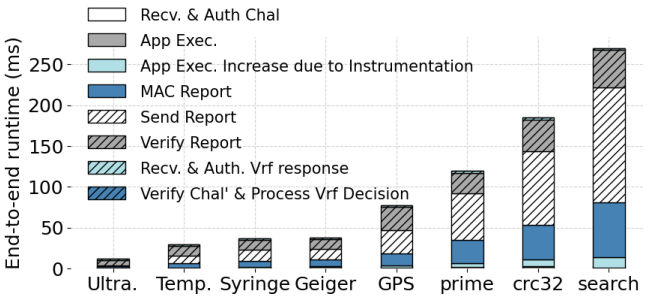

Figure 6: End-to-end runtime breakdown according to steps in Protocol 1.

7.1. End-to-End Runtime

We measure the end-to-end runtime to execute these applications, i.e., the time between Vrf's initial request and the receipt of *TRACES*'s last report at the end of App's execution. The end-to-end runtime depends on several factors, such as the characteristics of the application (shown in Table 1) and the configuration of *TRACES* triggers. Among those, the size of $\mathsf{CF}_{\mathsf{Log}}$ dedicated memory has a strong influence on the overhead because it determines the rate of [T3] trigger, affecting the time spent generating and transmitting each partial execution report.

We compare the end-to-end App runtime in three settings: original App without generation of any runtime evidence (baseline), App with best-effort CFA (ISC-FLAT [29]), and App with *TRACES* runtime auditing/guaranteed remediation guarantees. For a fair comparison to standard CFA, we set *TRACES* [T1] and [T3] triggers to large values to ensure that only one report is transmitted by *TRACES*. This is because most CFA approaches do not support the delivery of partial $\mathsf{CF}_{\mathsf{Log}}$-s, simply aborting execution when $\mathsf{CF}_{\mathsf{Log}}$-dedicated memory is full.

Fig. 5 presents this comparison. Measured times are the average of 20 executions of each step in each application. Standard deviations in all cases are less than 1% and omitted from the figure. The end-to-end time varies linearly with the size of $\mathsf{CF}_{\mathsf{Log}}$, resulting in $\approx$12.5-268ms of additional runtime across different applications.

Fig. 6 further breaks down *TRACES* end-to-end runtime overheads (for each tested App) according to individual steps in *TRACES*'s protocol. The time associated with receiving and authenticating messages from Vrf is application-independent. On the other hand, the overhead associated with the execution of instrumented instructions, as well as MAC-ing/transmitting/verifying the report, increases with the complexity of App-s. The only steps unique to *TRACES*, compared to CFA, are the steps for receiving and authenticating Vrf's response (containing Chal' and their next-action decision) and steps to process this response.

Fig. 6 shows that the total time taken is dominated by the transmission of generated reports, especially for applications with large $\mathsf{CF}_{\mathsf{Log}}$-s. Hence, the size of $\mathsf{CF}_{\mathsf{Log}}$-s influences the total end-to-end runtime the most. Similar to Fig. 5, times reported in Fig. 6 are the average of 20 measurements. Standard deviations in all cases are less than 1%.

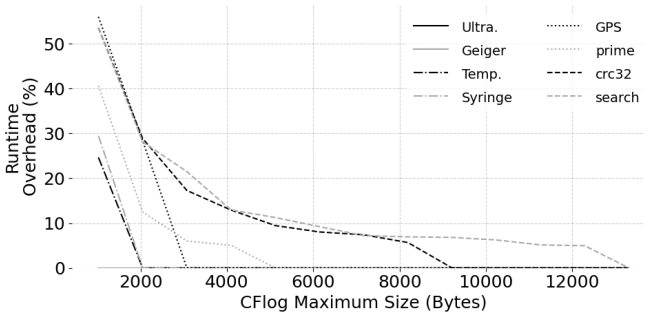

Figure 7: End-to-end overhead as $\mathsf{CF}_{\mathsf{Log}}$ increases

Next, we experiment by varying the size of $\mathsf{CF}_{\mathsf{Log}}$ from 1 to 13 KBytes. In this experiment, we disable the remaining triggers ([T1], [T2]) in order to guarantee that only [T3] triggers occur. Fig. 7 shows *TRACES*'s end-to-end runtime increase compared to the baseline, where Prv contains an unlimited $\mathsf{CF}_{\mathsf{Log}}$ storage and generates one report (i.e., the *TRACES* runtime reported in Fig. 5).

7.2. Attack Detection Delay

While *TRACES* ensures that $\mathsf{CF}_{\mathsf{Log}}$ is received by Vrf, there is a brief period (in between triggers) in which detection by Vrf is delayed. We refer to this period as the maximum "attack window". Two parameters can impact the attack window: (1) the period of [T1], which determines the maximum frequency of report generation, and (2) the maximum size of $\mathsf{CF}_{\mathsf{Log}}$. The impact of the maximum $\mathsf{CF}_{\mathsf{Log}}$ size on the attack window further depends on the "branch density" of App, i.e., App's rate of branch instructions executed. Fig. 8 illustrates the relationship between the maximum $\mathsf{CF}_{\mathsf{Log}}$ size and the attack window (in CPU cycles) for various branch density values (each line in Fig. 8 representing a different density). We note that the time to communicate and verify the report does not affect the attack window. This is because during that time, Prv remains in the Secure World.

If the branch density remains constant, decreasing the maximum $\mathsf{CF}_{\mathsf{Log}}$ size will lead to a shorter attack window; increasing $\mathsf{CF}_{\mathsf{Log}}$ size will result in a longer attack window.

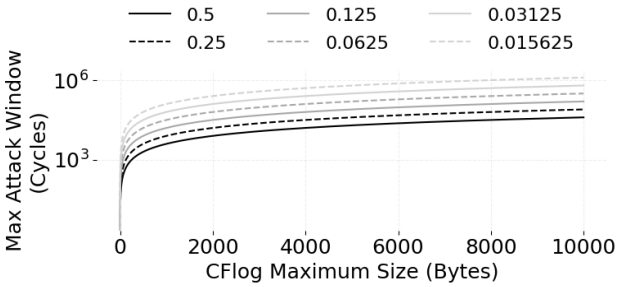

Figure 8: Attack window as $\mathsf{CF}_{\mathsf{Log}}$ increases

Furthermore, a shorter attack window reduces the time for Adv to interfere with App's execution. However, it also causes an increase in the end-to-end runtime due to the generation of more reports. This results in an interesting trade-off for *TRACES* parameter choices. For optimal performance in the on-demand sensing setting, [T1] and [T3] should be configured to meet the time and $\mathsf{CF}_{\mathsf{Log}}$ storage required for App's benign execution. Alternatively, [T1] and [T3] can be set to lower values to reduce the attack window by generating more frequent reports (at the price of more frequent transmission overhead).

Finally, we note that the less-strict verification policy discussed in Sec. 5.2 further impacts the attack window since the time for Vrf to detect the malicious behavior in the less-strict setting could increase by up to one [T1] trigger period.

7.3. Exploit Detection and CFA Verification

To exemplify *TRACES* functionality, we evaluate a vulnerable program containing a crafted exploit. We implement a sensor system intentionally designed to contain a buffer overflow vulnerability. In this App, Prv reads from an input buffer to determine which sensor software to execute (Ultrasonic, Temperature, or both) and how many readings to perform. This application reads from the input until a stop character is received. After reading the input, the command is parsed, and the sensor readings are performed based on the parsed input. Since there is no array bound check while reading from the input buffer, the return address stored on the stack can be overwritten to cause arbitrary behavior; in our example, malicious input causes the program to return to the function that reads the input, effectively causing an infinite loop and preventing any sensor readings from occurring thereafter. Since this attack overwrites a return address, the malicious return address value is written to $\mathsf{CF}_{\mathsf{Log}}$. Based on Prv's runtime report, Vrf must determine if Prv has been compromised. Vrf first always checks the authenticity of the report and H_{PMEM} to ensure that App's instrumented binary was indeed executed. If these checks succeed, as they do with the example attack, Vrf examines $\mathsf{CF}_{\mathsf{Log}}$. To verify $\mathsf{CF}_{\mathsf{Log}}$, Vrf emulates a shadow stack to validate return addresses shown in $\mathsf{CF}_{\mathsf{Log}}$ and utilizes App's CFG to validate the execution trace. As the (significantly more expensive) verification process is outsourced to Vrf, it does not incur additional overhead on the Prv. After identifying the illegal address caused by the buffer overflow, Vrf initiates remediation. This example, including the vulnerable software and resulting $\mathsf{CF}_{\mathsf{Log}}$, is available on *TRACES* prototype public repository [34].

8. Related Work

Remote Attestation (RA): RA architectures are generally classified in three types: software-based (or keyless), hardware-based, and hybrid. Software-based architectures [58], [59], [60], [61], [18], [11], [62], [16] require no hardware support. However, they require strong assumptions about Adv capabilities and implementation optimality. Hardware-based architectures [63], [64], [65], [20], [15] rely on standalone cryptographic coprocessors (e.g., TPMs [66]) or complex support from the CPU instruction set architecture (e.g., Intel SGX [67]). For contexts in which standalone hardware is too costly, hybrid architectures [8], [7], [9], [10], [12], [22] have been proposed. Hybrid architectures aim to combine the low hardware cost of software-based approaches with the security guarantees offered by hardware-based approaches through hardware/software co-designs for RA. Hybrid RA has also been extended to provide Vrf with a Proof-of-Execution (PoX) [68], [69] as unforgeable proof that an attested binary executed its outputs were generated by the execution.

CFA and Runtime Attestation: C-FLAT [23] proposed the concept of CFA by leveraging ARM TrustZone to detect and log control flow transfers. In C-FLAT, the binary is instrumented at branching instructions to trap execution to the Secure World. Once in the Secure World, the branching information is added to a hash chain. At the end of execution, C-FLAT's hash chain produces a unique value representing the attested program's control flow path. Similarly to C-FLAT, *TRACES* and many later techniques use App instrumentation and TEE support for CFA [45], [46], [27], [48]. To remove the requirement of instrumentation, several hardware-based CFA approaches [30], [31], [32] propose using customized hardware to handle detecting control flow transfers and attesting $\mathsf{CF}_{\mathsf{Log}}$, thus removing the need for instrumentation and requiring a TrustZone-equipped device. In hardware-based CFA, the CPU is extended with dedicated hardware for detecting branch instructions and extracting information about the control flow event (such as the source/destination addresses and type of branch instruction). In addition, they use a hardware-based hash engine to produce a CFA report for Vrf. Although these techniques remove the requirement for instrumentation, the hardware cost of a fully hardware-based solution is too costly for MCUs. Tiny-CFA [28] reduces the hardware cost significantly by executing an instrumented binary atop APEX [68], a low hardware-cost PoX architecture. Since only minimal hardware changes are required, Tiny-CFA demonstrates CFA that is realistic for low-end MCUs. ACFA [33] reduces the hardware cost without requiring instrumentation by implementing hardware for branch detection/logging and using software for generating the report.

TABLE 2: Qualitative comparison to related work

Related Work	'Off-the-shelf' Support	Verbatim CF_{Log}	CF_{Log} Slicing	Runtime Auditing	Remote Healing	Runtime Overhead	Hardware Overhead
C-FLAT [23]	Yes	No	No	No	No	Yes	No
OAT [27]	Yes	No	No	No	No	Yes	No
ARI [48]	Yes	No	No	No	No	Yes	No
LO-FAT [30]	No	No	No	No	No	No	Yes
ATRIUM [32]	No	No	No	No	No	No	Yes
LiteHAX [31]	No	Yes	Yes	No	No	No	Yes
Tiny-CFA [28]	No	Yes	No	No	No	Yes	Yes
TRACES	***Yes***	***Yes***	***Yes***	***Yes***	***Yes***	***Yes***	***No***

C-FLAT and other early approaches return a single hash of CF_{Log}, putting Vrf at risk of path explosion during the verification process. To avoid this problem, several recent approaches record a verbatim log of all control flow transfers [33], [28], [70]. A limitation to logging all control flow transfers verbatim is that CF_{Log} quickly fills all memory available to the low-end MCU. Because of this, several approaches aim to reduce the size of the verbatim CF_{Log} by only logging control flow transfers that cannot be determined statically in their entirety [27], [31], [46], [48] by reducing the storage size and sending send a series of intermediate log slices [45], [33], recording encodings of full addresses [27], [31], [46], or recording forward edges verbatim alongside a hash-chain of the return addresses [27], [48]. *TRACES* logs only non-deterministic branches and supports CF_{Log} slicing to provide Vrf fine-grained reports while incurring minimal and fixed memory overheads.

Comparison to Related Work. Table 2 compares *TRACES* to most closely related runtime attestation techniques. Similar to our work, C-FLAT [23], OAT [27], and ARI [48] propose CFA for off-the-shelf ARM MCUs equipped with TrustZone-M. Thus, they incur a similar runtime overhead due to the required instrumentation for recording control flow events. Unlike these techniques, *TRACES* actively triggers the transmission of whole or partial CF_{Log}-s, guarantees delivery of this evidence, and enables Vrf-triggered device healing in commercial MCUs. Tiny-CFA [28] combines instrumentation atop a hardware-assisted PoX architecture [68] to achieve CFA. Hence, it also incurs runtime overhead. Additionally, Tiny-CFA requires custom hardware support That is not available off-the-shelf. LO-FAT [30], ATRIUM [32], and LiteHAX [31] are fully-hardware based approaches. In contrast to *TRACES*, they avoid the runtime overhead cost in exchange for additional hardware features. Because of this, they require new MCUs to be fabricated before they can be used. Similarly, these are techniques for attestation rather than *auditing*, so they do not offer the same guarantees as *TRACES*.

Active CFA (ACFA) and Device Healing: As discussed in Sec. 1, ACFA is a hybrid (software/hardware) based approach to augment the capabilities of CFA by leveraging the concept of "active roots of trust" [71] to provide control flow auditing. As CF_{Log} is built, a (custom hardware-based) active root of trust interrupts execution to attest the binary and CF_{Log}. ACFA also supports a remediation phase after a compromise is detected. However, contrary to *TRACES*, ACFA requires custom hardware modifications and is therefore not applicable to current devices. Other approaches also discuss remote device healing [72], [73], [74], [75]. Unlike *TRACES*, they either rely on custom hardware, do not consider runtime auditing, and/or use attestation (without reliable delivery) to verify the healing action without guaranteeing its occurrence when Prv is compromised.

ARM TrustZone: Prior work has used TrustZone-M to enhance various aspects of embedded system security: availability in real-time systems [76], low latency secure interrupts [40], Address-Space Layout Randomization (ALSR) without memory management units [77], and virtualization [78]. For a more comprehensive discussion of TrustZone and related applications, we refer the reader to [37].

9. Conclusion

We proposed *TRACES*: an approach for runtime auditing and guaranteed remediation aimed at commodity MCUs equipped with TEEs. *TRACES* guarantees that control flow logs that contain compromise evidence are always received by a remote verifier. This enables analysis of these logs to pinpoint unknown vulnerabilities and support their remediation. To our knowledge, *TRACES* is the first design to support these security services without requiring custom hardware modifications. We implement a fully functional and open-source *TRACES* prototype [34] based on the ARM TrustZone-M TEE atop the commodity ARM Cortex-M33 MCU.

Acknowledgements

We sincerely thank the paper's anonymous shepherd and ACSAC'24 reviewers for their constructive comments and feedback. RIT authors were partly funded by the National Science Foundation (SaTC award #2245531). PSU author was partly 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] J. Deogirikar and A. Vidhate, "Security attacks in iot: A survey," in *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*. IEEE, 2017, pp. 32–37.

[2] J. Vijayan, "Stuxnet renews power grid security concerns," https://www.computerworld.com/article/1511749/stuxnet-renews-power-grid-security-concerns-2.html, june 2010.

[3] J. Giraldo, A. Cárdenas, and N. Quijano, "Integrity attacks on real-time pricing in smart grids: Impact and countermeasures," *IEEE Transactions on Smart Grid*, vol. 8, no. 5, pp. 2249–2257, 2016.

[4] M. N. Nafees, N. Saxena, A. Cardenas, S. Grijalva, and P. Burnap, "Smart grid cyber-physical situational awareness of complex operational technology attacks: A review," *ACM Computing Surveys*, vol. 55, no. 10, pp. 1–36, 2023.

[5] H. Kayan, M. Nunes, O. Rana, P. Burnap, and C. Perera, "Cybersecurity of industrial cyber-physical systems: a review," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–35, 2022.

[6] I. D. O. Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "Towards remotely verifiable software integrity in resource-constrained iot devices," *IEEE Communications Magazine*, 2024.

[7] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing dynamic) root of trust," in *NDSS*, vol. 12, 2012, pp. 1–15.

[8] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified Hardware/Software Co-Design for remote attestation," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1429–1446.

[9] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Proceedings of the 52nd annual design automation conference*, 2015, pp. 1–6.

[10] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–14.

[11] M. Ammar, B. Crispo, and G. Tsudik, "Simple: A remote attestation approach for resource-constrained iot devices," in *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2020, pp. 247–258.

[12] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "HYDRA: hybrid design for remote attestation (using a formally verified microkernel)," in *Proceedings of the 10th ACM Conference on Security and Privacy in wireless and Mobile Networks*, 2017, pp. 99–110.

[13] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the toctou problem in remote attestation," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2921–2936.

[14] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *Proceedings of the 11th ACM conference on Computer and communications security*, 2004, pp. 132–145.

[15] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A low-cost security architecture for iot devices," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 3, pp. 1–33, 2017.

[16] L. Petzi, A. E. B. Yahya, A. Dmitrienko, G. Tsudik, T. Prantl, and S. Kounev, "SCRAPS: Scalable collective remote attestation for Pub-Sub IoT networks with untrusted proxy verifier," pp. 3485–3501, 2022.

[17] I. D. O. Nunes, G. Dessouky, A. Ibrahim, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "Towards systematic design of collective remote attestation protocols," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1188–1198.

[18] M. Grisafi, M. Ammar, M. Roveri, and B. Crispo, "PISTIS: Trusted computing architecture for low-end embedded systems," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3843–3860.

[19] M. M. Rabbani, E. Dushku, J. Vliegen, A. Braeken, N. Dragoni, and N. Mentens, "Reserve: Remote attestation of intermittent iot devices," in *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, 2021, pp. 578–580.

[20] J. Vliegen, M. M. Rabbani, M. Conti, and N. Mentens, "SACHa: Self-attestation of configurable hardware," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 746–751.

[21] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to remote attestation," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.

[22] F. Brasser, K. B. Rasmussen, A.-R. Sadeghi, and G. Tsudik, "Remote attestation for low-end embedded devices: the prover's perspective," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[23] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 743–754.

[24] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[25] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, pp. 1–34, 2012.

[26] M. Ammar, A. Caulfield, and I. D. O. Nunes, "Sok: Runtime integrity," *arXiv preprint arXiv:2408.10200*, 2024.

[27] Z. Sun, B. Feng, L. Lu, and S. Jha, "Oat: Attesting operation integrity of embedded devices," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1433–1449.

[28] I. D. O. Nunes, S. Jakkamsetti, and G. Tsudik, "Tiny-CFA: Minimalistic control-flow attestation using verified proofs of execution," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 641–646.

[29] A. J. Neto and I. D. O. Nunes, "Isc-flat: On the conflict between control flow attestation and real-time operations," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2023, pp. 133–146.

[30] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "Lo-fat: Low-overhead control flow attestation in hardware," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[31] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: lightweight hardware-assisted attestation of program execution," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[32] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi, "Atrium: Runtime attestation resilient under memory attacks," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 384–391.

[33] A. Caulfield, N. Rattanavipanon, and I. D. O. Nunes, "ACFA: Secure runtime auditing & guaranteed device healing via active control flow attestation," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5827–5844. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/caulfield

[34] A. Caulfield, A. J. Neto, N. Rattanavipanon, and I. De Oliveira Nunes, "TRACES Prototype Repository," 2024. [Online]. Available: https://github.com/RIT-CHAOS-SEC/TRACES/

[35] A. Ltd, "Trustzone technology for armv8-m architecture version 2.1," https://developer.arm.com/documentation/100690/0201/, 2019.

[36] *ARM Security Technology - Building a Secure System using TrustZone Technology*, ARM Limited, 2009.

[37] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM computing surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[38] *Memory Protection Unit (MPU) Version 1.0*, ARM, 2016.

[39] STMicroelectronics, *RM0438 Reference manual: STM32L552xx and STM32L562xx advanced Arm-based 32-bit MCUs*, December 2020.

[40] R. Pan and G. Parmer, "Sbis: Application access to safe, baremetal interrupt latencies*," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 82–94.

[41] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[42] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 552–561.

[43] Z. Ma, X. Tan, L. Ziarek, N. Zhang, H. Hu, and Z. Zhao, "Return-to-non-secure vulnerabilities on arm cortex-m trustzone: Attack and defense," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[44] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM symposium on information, computer and communications security*, 2011, pp. 30–40.

[45] F. Toffalini, E. Losiouk, A. Biondo, J. Zhou, and M. Conti, "ScaRR: Scalable runtime remote attestation for complex systems," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 121–134.

[46] Y. Zhang, X. Liu, C. Sun, D. Zeng, G. Tan, X. Kan, and S. Ma, "ReCFA: resilient control-flow attestation," in *Annual Computer Security Applications Conference*, 2021, pp. 311–322.

[47] M. Geden and K. Rasmussen, "Hardware-assisted remote runtime attestation for critical embedded systems," in *2019 17th International Conference on Privacy, Security and Trust (PST)*. IEEE, 2019, pp. 1–10.

[48] J. Wang, Y. Wang, A. Li, Y. Xiao, R. Zhang, W. Lou, Y. T. Hou, and N. Zhang, "ARI: Attestation of real-time mission execution integrity," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2761–2778.

[49] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *17th International Conference on VLSI Design. Proceedings.* IEEE, 2004, pp. 605–611.

[50] J. Obermaier and V. 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*, vol. 2, no. 4, pp. 289–296, 2018.

[51] J. Pallister, S. Hollis, and J. Bennett, "Beebs: Open benchmarks for energy measurements on embedded platforms," *arXiv preprint arXiv:1308.5174*, 2013.

[52] Seeed-Studio, "Ultrasonic Ranger," Jun. 2015. [Online]. Available: https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/ultrasonic_ranger

[53] Y. Tournade, "ArduinoPocketGeiger Github Repository," https://github.com/MonsieurV/ArduinoPocketGeiger, 2020.

[54] T. Walker, "OpenSyringePump," Apr. 2022. [Online]. Available: https://github.com/manimino/OpenSyringePump

[55] Seeed-Studio, "Temperature Sensor," Jun. 2015. [Online]. Available: https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/temp_humi_sensor

[56] M. Hart, "Tinygps++," http://arduiniana.org/libraries/tinygpsplus/, 2014.

[57] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1789–1806.

[58] R. Kennell and L. H. Jamieson, "Establishing the genuinity of remote computer systems," in *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

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

[60] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 1–16.

[61] A. Seshadri, M. Luk, and A. Perrig, "Sake: Software attestation for key establishment in sensor networks," in *Distributed Computing in Sensor Systems: 4th IEEE International Conference, DCOSS 2008 Santorini Island, Greece, June 11-14, 2008 Proceedings 4*, 2008, pp. 372–385.

[62] S. Surminski, C. Niesler, F. Brasser, L. Davi, and A.-R. Sadeghi, "Realswatt: remote software-based attestation for embedded devices under realtime constraints," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2890–2905.

[63] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor." in *USENIX security symposium*. San Diego, USA, 2004, pp. 179–194.

[64] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New results for timing-based attestation," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 239–253.

[65] D. Schellekens, B. Wyseur, and B. Preneel, "Remote attestation on legacy operating systems with trusted platform modules," *Science of Computer Programming*, vol. 74, no. 1-2, pp. 13–22, 2008.

[66] Trusted Computing Group., "Trusted platform module (tpm)," 2017. [Online]. Available: http://www.trustedcomputinggroup.org/work-groups/trusted-platform-module/

[67] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086, Tech. Rep.

[68] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "APEX: A verified architecture for proofs of execution on remote devices under full software compromise," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 771–788.

[69] A. Caulfield, N. Rattanavipanon, and I. De Oliveira Nunes, "ASAP: reconciling asynchronous real-time operations and proofs of execution in simple embedded systems," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 721–726.

[70] I. D. O. Nunes, S. Jakkamsetti, and G. Tsudik, "Dialed: Data integrity attestation for low-end embedded devices," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 313–318.

[71] E. Aliaj, I. D. O. Nunes, and G. Tsudik, "GAROTA: Generalized active root-of-trust architecture (for tiny embedded devices)," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2243–2260.

[72] A. Ibrahim, A.-R. Sadeghi, and G. Tsudik, "Healed: Healing & attestation for low-end embedded devices," in *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*. Springer, 2019, pp. 627–645.

[73] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "Pure: Using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[74] M. Ammar and B. Crispo, "Verify&revive: Secure detection and recovery of compromised low-end embedded devices," in *Annual Computer Security Applications Conference*, 2020, pp. 717–732.

[75] M. Huber, S. Hristozov, S. Ott, V. Sarafov, and M. Peinado, "The lazarus effect: Healing compromised devices in the internet of small things," pp. 6–19, 2020.

[76] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang, "Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 352–369.

[77] L. Luo, X. Shao, Z. Ling, H. Yan, Y. Wei, and X. Fu, "faslr: Function-based aslr via trustzone-m and mpu for resource-constrained iot systems," *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 17 120–17 135, 2022.

[78] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on trustzone-enabled microcontrollers? voilà!" in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 293–304.