RESPEC-CFA: Representation-Aware Speculative Control Flow Attestation

Liam Tyler[1][0009−0003−0341−5631], Adam Caulfield[2][0000−0002−5631−7328], and Ivan De Oliveira Nunes[1][0000−0003−3486−6550]

[1] University of Zurich, Zurich, Switzerland
[2] University of Waterloo, Waterloo, Canada

Abstract. Microcontroller Units (MCUs), often remotely deployed to perform safety-critical sensing and actuation within larger cyber-physical systems, require low-cost security services due to their resource constraints. Among them, Control Flow Attestation (*CFA*) is a challenge-response protocol wherein a remote Verifier ($\mathcal{V}$rf) issues a cryptographic challenge (*Chal*) to a potentially compromised Prover MCU ($\mathcal{P}$rv) to demonstrate $\mathcal{P}$rv has executed intended software without the presence of control flow attacks. A root of trust within $\mathcal{P}$rv is responsible for producing an authenticated log of the control flow path (CF_{Log}) taken during the execution of an attested software operation and computing an authenticated integrity token (e.g., a MAC or signature) over the current snapshot of $\mathcal{P}$rv's program memory, CF_{Log}, and *Chal*. By examining the produced response, $\mathcal{V}$rf can determine if $\mathcal{P}$rv's code has been illegally modified or fell victim to a control flow attack.
An important bottleneck in *CFA* is the storage and transmission of CF_{Log}-s. To address this, state-of-the-art optimization methods focus on application-specific optimizations that speculate on likely control flow sub-paths by replacing likely paths with reserved symbols of reduced size. We argue that prior approaches overlook the data representation of control flow paths in their speculation strategy. Based on this observation, we propose *RESPEC-CFA*, an architectural extension for *CFA* allowing control flow path speculation based on (1) the locality of control flow paths and (2) their Huffman encoding. *RESPEC-CFA* alone reduces CF_{Log} sizes by up to 90.1%. We also strive to design *RESPEC-CFA* such that it can compose synergistically with state-of-the-art methods. As a result, when combined with prior methods, *RESPEC-CFA* achieves up to 99.7% in log size reductions (without loss of information), significantly outperforming previous approaches and advancing practical *CFA*.

Keywords: Attestation · Embedded Systems · Software Security.

1 Introduction

Modern cyber-physical systems depend on Microcontroller Units (MCUs) for sensing and actuation. However, given their low cost and low energy requirements, MCUs often lack security features comparable to general-purpose computers. For example, they typically lack Memory Management Units (MMUs),

inter-process isolation, or strong privilege level separation (see Section 2.1 for more details on MCU architectures). Yet, MCUs often perform system-critical tasks as a part of larger systems in which they are embedded, making them attractive targets of attacks [25]. Therefore, reliable methods to assess the integrity of remote MCUs are crucial.

Remote Attestation (*RA*) [42,41,57] is a two-party protocol that allows a Verifier ($\mathcal{V}$rf) to measure the software state of a remote Prover MCU ($\mathcal{P}$rv) in a cost-effective manner. In *RA*, $\mathcal{V}$rf requests an authenticated report from $\mathcal{P}$rv to determine if the correct software is installed on $\mathcal{P}$rv. While effective in detecting malicious code modifications, *RA* is oblivious to attacks such as control flow hijacking [54] that alter the program's behavior without changing instructions. Control Flow Integrity (CFI) [48,19,2] can be used to locally detect some of these attacks on $\mathcal{P}$rv. However, it provides no evidence of the attack behavior to $\mathcal{V}$rf.

Control Flow Attestation (*CFA*) [3,78,15,63,12,10,79,66,73,77] provides $\mathcal{V}$rf the ability to ascertain both the runtime behavior and integrity of $\mathcal{P}$rv. *CFA* extends *RA* to record a trace of the control flow path followed during the attested program's execution. This trace is created by logging the destinations of all control flow instructions (e.g., `call`, `jump`, or `ret`) executed. The resulting control flow log (CF_{Log}) is authenticated alongside $\mathcal{P}$rv's installed code (per standard *RA*) and sent to $\mathcal{V}$rf. With CF_{Log}, $\mathcal{V}$rf can determine whether the attested execution had valid runtime behavior. For more details on CFI, *CFA*, as well as their differences, we refer the reader to the systematization in [6].

As CF_{Log} contains all branches taken, its storage and eventual transmission are bottlenecks for *CFA*. Early *CFA* techniques [3,17,78] avoided this by compressing CF_{Log} into a single hash digest by computing a hash-chain of all control flow destinations in CF_{Log}. However, as attested programs become more complex, this approach leads to the well-known path explosion problem [52], making verification by $\mathcal{V}$rf infeasible. Similarly, hash-based approaches do not offer insight into malicious control flows taken. As a consequence, more recent *CFA* methods [15,63,12,10,79,66,73,77] tend to log paths *verbatim* aside from simple program-agnostic log optimizations (e.g., replacing simple loops with counters).

Program-agnostic optimizations do not capture application-specific characteristics that can offer further CF_{Log} reductions. Therefore, recent work proposed application-specific optimizations. SpecCFA [13] allows $\mathcal{V}$rf to speculate on and configure $\mathcal{P}$rv with a set of expected/likely control flow paths for the application being attested. Then, at runtime, matching sub-paths in CF_{Log} are replaced with reserved symbols of reduced size. As a result, SpecCFA significantly reduces CF_{Log} storage and transmission costs. SpecCFA's optimization strategy depends on the predictability and frequency of sub-paths in $\mathcal{P}$rv's execution. However, SpecCFA does not account for application characteristics such as redundancy in the data representation of CF_{Log} or the memory locality of instructions.

Based on the observation above, our premise in the present work is that speculating on these other predictable characteristics could further reduce CF_{Log}. Therefore, we propose REpresentation-aware SPECulative *CFA* (*RESPEC-CFA*), a method (accompanied by corresponding architectural design and implementa-

tion) to enable secure *CFA* speculation based on two new application-specific characteristics:

- First, *RESPEC-CFA* allows $\mathcal{V}\mathsf{rf}$ to speculate on the locality of instructions in an attested program. MCU applications are typically statically linked to fixed program memory address ranges. Hence, code addresses often share common prefixes. *RESPEC-CFA* allows $\mathcal{V}\mathsf{rf}$ to speculate on the length of this prefix, grouping CF_{Log} entries by shared prefix. Each prefix is added to CF_{Log} with a special symbol to distinguish it from regular addresses. For subsequent entries sharing the same prefix, only the suffix is logged. When a new address has a different prefix, the new prefix is logged, and the process repeats: only suffixes are logged until the next prefix mismatch. This reduces the size of most CF_{Log} entries, removing redundant data without loss of information.
- Second, *RESPEC-CFA* allows $\mathcal{V}\mathsf{rf}$ to speculate on CF_{Log}'s data representation itself. For this, $\mathcal{V}\mathsf{rf}$ speculates on a Huffman encoding [22] (e.g., based on previously received CF_{Log}-s for the same code) and sends it to $\mathcal{P}\mathsf{rv}$ along with a *CFA* request. Upon receipt, *RESPEC-CFA* uses the Huffman encoding to compress CF_{Log} at runtime. This allows *CFA* to benefit from Huffman-based compression without placing the burden of computing compression algorithms on the resource-limited $\mathcal{P}\mathsf{rv}$.

CFA schemes either rely on custom hardware support [12,15,78,16] or Trusted Execution Environments (TEEs) [3,63,73,10,79,66,39]. While *RESPEC-CFA*'s concept applies to both categories, we implement *RESPEC-CFA* by modifying SpecCFA's TEE-based implementation [13]. This choice is motivated by (1) SpecCFA's open-source availability and (2) our goal of jointly implementing SpecCFA and RESPEC-CFA to maximize the combined benefits of both methods (their combined benefits are later confirmed by our experiments in Section 4.2). Therefore, *RESPEC-CFA*'s prototype inherits SpecCFA's characteristic of targeting "off-the-shelf" MCUs with TEE support (specifically, ARM Cortex-M with TrustZone). We evaluate *RESPEC-CFA*'s performance using real-world MCU applications and find that it achieves large CF_{Log} reduction with little runtime overhead. When combined with SpecCFA, *RESPEC-CFA* achieves up to 99.7% reduction of CF_{Log} size for the evaluated applications. We also make *RESPEC-CFA*'s prototype publicly available at [68].

2 Background & Related Work

2.1 MCU Architectures

MCUs are compact processors with CPU, memory, and I/O peripherals built into one low-cost chip. They are typically embedded within larger systems and used for sensing/actuation and real-time responses to stimuli. Additionally, they offer low-power execution modes/idle states until asynchronous interrupt-based processing is triggered. These characteristics make them useful for a variety of settings, including those that require lengthy deployments or minimal energy consumption.

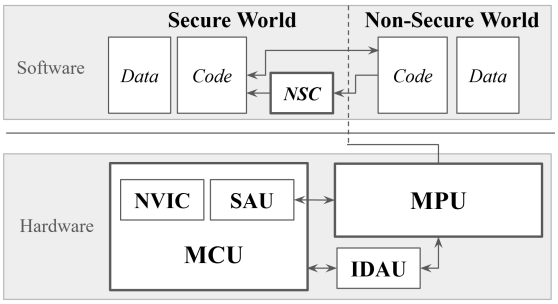


Fig. 1: ARM Cortex-M TrustZone

The CPU within an MCU is typically single-core and executes software from physical memory (at "bare-metal"), i.e., without an MMU to enable virtualization and inter-process isolation. On the lower end of the scale (e.g., 8- or 16-bit CPUs from Microchip AVR [37] or TI MSP430 [24]), they typically run 1-16 MHz clock frequencies with 4-256 KB FLASH or FRAM memory for instructions and 1-64KB SRAM memory for data. As it relates to security resources, many devices are not equipped with extensive modules. In some cases, they might be equipped with general-purpose Memory Protection Units (MPU), but are limited (e.g., support for three configurable regions only in program memory [20]), or other security modules (e.g., Intellectual Property Encapsulation in TI MSP430 [23]).

Slightly more advanced MCUs include ARM Cortex-M MCUs (e.g., ARM Cortex-M33 used for prototyping in this work [35]). The ARM Cortex-M class of MCUs has 32-bit CPUs that typically range from 48-800 MHz clock frequencies, between 16-2048 KB of FLASH memory, and 4-1400 KB SRAM memory [62,61]. They are also equipped with a Wake-up Interrupt Controller (WIC) that enables entering idle states and low-power modes. The ARM Cortex-M class of MCUs also has more security features, such as stronger MPUs (e.g., supporting up to 8-16 configurable regions over all addressable memory) and the TrustZone security extension (discussed further in Section 2.2). Yet, it lacks MMUs/virtual memory.

2.2 TrustZone for MCUs

ARMv8 Cortex-M MCUs are equipped with the TrustZone (i.e., TrustZone-M) TEE [31]. As illustrated in Figure 1, TrustZone provides strong software isolation by dividing hardware and software into two worlds: the "Non-Secure" and "Secure" worlds.

These worlds are defined by two hardware controllers: the Secure Attribution Unit (SAU) and the Implementation-Defined Attribution Unit (IDAU) [34]. The region definitions enforced by the IDAU are fixed by the manufacturer, and developers can configure the SAU via Secure World code to assign additional memory to the Secure World as needed for a particular program. These configurations set by IDAU and SAU are enforced alongside any specific inner-world access controls (e.g., setting Non-Secure World code as read and execute only).

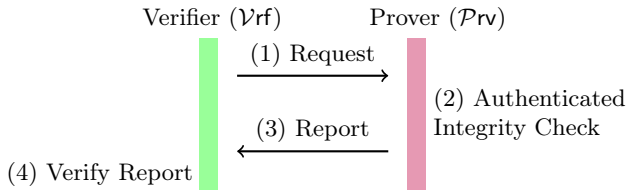


Fig. 2: A typical *RA* interaction

Additionally, ARM Cortex-M MCUs are typically equipped with a Nested Vector Interrupt Controller (NVIC) [30,29] that manages interrupts. The NVIC can be controlled by Secure World code to assign interrupts to a particular world. It can also be configured to ensure Non-Secure World interrupts do not interrupt the Secure World and to set Secure World interrupts as higher priority.

TrustZone's hardware-based isolation ensures that the Non-Secure World cannot tamper with code and data belonging to the Secure World [28]. As such, the Secure World can safely store security-critical functionality. TrustZone also forces controlled invocation of the Secure World through dedicated entry points called Non-Secure-Callables (NSCs), while enabling the Secure World to call Non-Secure World code directly, as depicted in Figure 1.

Prior work has used TrustZone-M to enhance various aspects of embedded system security, including but not limited to availability/performance [72,49] and enabling security mechanisms of high-end CPUs (e.g., ALSR without MMUs [36], and virtualization [51]). Similarly, several works have utilized TrustZone for detecting control flow attacks, whether done locally through CFI [65,74,48] or remotely through CFA [3,4,63,73,39,10]. For a more comprehensive discussion of TrustZone see [50].

2.3 Remote Attestation

RA occurs between a $\mathcal{V}$rf and a potentially compromised $\mathcal{P}$rv, allowing $\mathcal{V}$rf to remotely assess $\mathcal{P}$rv's state. An *RA* instance is comprised of the following core steps (depicted in Figure 2):

1. $\mathcal{V}$rf sends a cryptographic challenge *Chal*, requesting $\mathcal{P}$rv attest to its current state.
2. Upon receiving *Chal*, $\mathcal{P}$rv produces a token H by computing an authenticated integrity check on its memory and *Chal*.
3. $\mathcal{P}$rv responds to $\mathcal{V}$rf by sending H.
4. $\mathcal{V}$rf compares H against its expected value to determine if $\mathcal{P}$rv has been compromised.

The authenticated integrity check in step 2 is implemented using a message authentication code (MAC) or a digital signature. Hence, the secret key used to produce H must be securely stored and used by a root of trust (RoT) on $\mathcal{P}$rv in full isolation from any compromised software on $\mathcal{P}$rv. Optionally, the RoT in $\mathcal{P}$rv may also authenticate $\mathcal{V}$rf requests (in step 1). This mitigates denial-of-service

attempts via bogus *RA* requests [8] and ensures that any other data within the request (e.g., $\mathcal{V}$rf-issued commands in security services that build upon *RA*, such as [12,10,45,44,14]) are genuine. In the context of this work, it also ensures that $\mathcal{V}$rf-defined speculations are authentic.

RA is generally classified by its RoT implementation. Early schemes relied solely on software to attest the $\mathcal{P}$rv's state. While applicable to commodity MCUs, these software-only approaches require deterministic timing characteristics such as a wired interface between $\mathcal{V}$rf and $\mathcal{P}$rv for predictable network latency [59,58,57]. These assumptions often make software-based approaches inapplicable to remote settings [9].

Other models [26,53,41] use dedicated hardware and hardware-protected secrets to attest the $\mathcal{P}$rv. Hardware-based approaches provide stronger security guarantees, but the additional hardware cost can be prohibitive for resource-constrained MCUs.

Hybrid *RA* [42,18,7] schemes balance hardware-assisted checks with the lower cost of software. They typically implement the MAC/signature generation in software, while using minimal hardware to securely store the secret key and protect the execution of the RoT software.

Runtime Attestation expands upon classic *RA* to also convey evidence about $\mathcal{P}$rv execution properties. For example, *proofs of execution* [43,11,40] also prove whether the attested function executed completely and whether claimed outputs were indeed produced by such execution. Another type of runtime attestation, and the focus of this work, is *CFA* [3,63,73,10,39,66,79,77,78,17,15,12,5]. *CFA* extends *RA* to detect control flow attacks [64] that alter the execution of an attested program without modifying its code.

2.4 Control Flow Attestation

CFA extends *RA* to generate a CF_{Log} of the attested program's execution by recording the destination of branching instructions (e.g., `jump`, `call`, and `ret` instructions) at runtime. To detect these branching instructions and securely store CF_{Log}, existing *CFA* techniques rely on either (1) binary instrumentation with TEE-support [3,63,73,10,39,66,79,77,5,38] or (2) custom hardware elements [78,17,15,12]. When the attested execution completes, the resulting CF_{Log} is signed/MAC-ed alongside $\mathcal{P}$rv's program memory content (as per *RA*) to produce H. Both H and CF_{Log} are sent to $\mathcal{V}$rf. With H, $\mathcal{V}$rf can validate $\mathcal{P}$rv's code integrity and authenticate CF_{Log}. CF_{Log} tells $\mathcal{V}$rf the executed path.

Early *CFA* schemes used a single hash to represent CF_{Log} [3,78,17], compressing the execution trace into a small fixed-size value. This approach minimized the storage and transmission overhead associated with *CFA*. Similarly, to verify a given execution, $\mathcal{V}$rf simply needs to check if the received hash exists in the set of all valid execution hashes. However, as binaries get more complex, trying to enumerate all possible paths through the program becomes exponentially complex, leading to the path explosion problem [52]. Further, hash-based approaches can only detect if a given run is invalid. While malicious control flows change

the final hash result, the malicious path itself is not visible to $\mathcal{V}\mathsf{rf}$. As a result, $\mathcal{V}\mathsf{rf}$ cannot learn what triggered the attack nor how to correct it.

To address these limitations, recent *CFA* techniques log all control flow transfers *verbatim* [15,63,10,12,66,79,77]. This eases verification; however, *verbatim* logs can quickly outgrow the memory available on MCUs. Hence, prior work introduced several simple CF_{Log} optimizations. Some approaches reduce the size of CF_{Log} by limiting their scope to a subset of operations, such as indirect branches [47,63,46] or a subset of the code [73]. Others reduce the size of log entries themselves rather than the number of entries logged. LiteHAX [15] records conditional branches with a single bit ('1' if the branch was taken, '0' otherwise) while indirect branches are logged in full. OAT [63] uses a similar bitstream representation to LiteHAX; however, OAT creates a hash-chain of return addresses rather than logging them directly. Despite using hash-chains, the added context of the rest of CF_{Log} allows OAT to avoid the issues associated with the early hash-based *CFA* approaches. Many *CFA* techniques also replace repeated loop entries with a count denoting how many times the loop executed [3,12,10,47,46,78,79].

Regardless of these optimizations, it is still possible for CF_{Log} to outgrow the available memory. In response, some *CFA* controls fix the size of CF_{Log} in memory and transmit the log in slices throughout the attested execution when available memory is full [66,12,10]. On its own, this approach trades storage overhead for increased transmission/runtime costs due to the additional intermediate log transmissions. As such, *CFA* techniques often combine this approach with other optimizations to reduce the number of CF_{Log} slices that must be transmitted.

Unsurprisingly, methods applied generically across different applications inherently miss application-specific characteristics that can be leveraged to further reduce CF_{Log}. SpecCFA [13] shows the benefits of application-aware optimization by enabling $\mathcal{V}\mathsf{rf}$ to speculate on high-likelihood control flow sub-paths. From the binary or previously received CF_{Log}-s, $\mathcal{V}\mathsf{rf}$ can configure $\mathcal{P}\mathsf{rv}$ with a set of frequent/expected execution paths (e.g., frequent control loops, sensing operations, etc.). At runtime, SpecCFA replaces these sub-paths in CF_{Log} with fixed-size IDs, reducing CF_{Log}'s size. As $\mathcal{V}\mathsf{rf}$ knows the unique path-to-ID correspondence, SpecCFA does not result in any loss of information in CF_{Log}.

3 *RESPEC-CFA*

MCU applications are typically statically linked within a fixed address range, and branch instructions often use relative offsets, making destination addresses predictable. This results in program locality, i.e., common address prefixes can be anticipated. Additionally, CF_{Log} data may exhibit skewed distributions due to frequent patterns such as loops, sub-paths, or common address ranges.

Building on these observations, we present a method (and supporting design) that enables $\mathcal{V}\mathsf{rf}$ to speculate on address prefix sizes and Huffman encodings tailored to the expected CF_{Log} data. This improves CF_{Log} compression at

its construction time. We realize *RESPEC-CFA* as a TrustZone Secure World module that extends *CFA* to support these two key optimizations. We also show how *RESPEC-CFA* can be composed with the state-of-the-art and the benefits of this composition.

Remark. *Key to RESPEC-CFA's practicality is not burdening resource-constrained* $\mathcal{P}$rv *with Huffman encoding computation. Instead,* $\mathcal{V}$rf *takes on this burden by speculating on the ideal encoding based on previously received* CF_{Log}*-s. This enables both: reduced* CF_{Log} *size and minimal runtime overhead on* $\mathcal{P}$rv*.*

3.1 System and Adversary Models

RESPEC-CFA targets single-core, bare-metal MCUs (recall Section 2.1) equipped with TEEs (TrustZone-M, in our prototype). Attested applications (App-s) execute in the Non-Secure World. The Secure World is used to house trusted software modules, including *RESPEC-CFA*. TEE support is used for:

- Secure storage of attestation keys, which must be securely provisioned prior to deployment;
- Isolation of the Secure World's code and data from any App in the Non-Secure World;

These characteristics can be achieved through standard ARM TrustZone-M v8 architectural support [31].

We consider an adversary ($\mathcal{A}$dv) capable of fully compromising $\mathcal{P}$rv's Non-Secure World. $\mathcal{A}$dv can exploit memory vulnerabilities in $\mathcal{P}$rv to perform control flow hijacking or code-reuse attacks. In addition, $\mathcal{A}$dv can manipulate Non-Secure World interrupts and their interrupt service routines (ISRs). $\mathcal{A}$dv cannot modify any Secure World code and data due to the underlying TEE hardware protections. $\mathcal{A}$dv cannot bypass $\mathcal{P}$rv's hardware-enforced controls. TEE-based *CFA* relies on binary instrumentation to log control flow transfers. Thus, the code of the application being attested must be immutable during its execution. This is a standard requirement enforced by the underlying *CFA* schemes [6] and is achieved by using TrustZone memory access controls. We consider physical attacks (e.g., fault injection [69] or bus interposition [60]) and availability attacks (e.g., denial of service [76]) orthogonal and out-of-scope for this work.

3.2 *RESPEC-CFA* High-Level Workflow

RESPEC-CFA's workflow is shown in Figure 3. To configure *RESPEC-CFA*, $\mathcal{V}$rf extends the *CFA* request to include a speculated Huffman encoding table and speculated prefix length generated for the attested application App. Recall that the *CFA* request (and the speculation strategy within) is authenticated. If no speculation is specified in the request, *RESPEC-CFA* uses previously configured speculations by default.

Upon receiving and authenticating the request, $\mathcal{P}$rv saves the speculated Huffman table and prefix length to Secure World memory and begins App attested execution in the Non-Secure World. Before deployment, App is instrumented

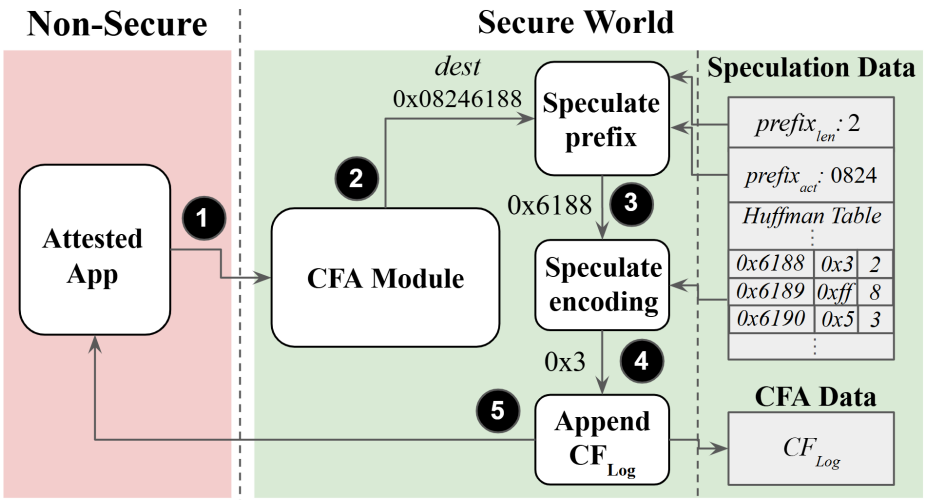


Fig. 3: *RESPEC-CFA* architecture. Addresses are represented in hexadecimal.

(as in prior work [3,4,39,10,66,63]) with NSC calls to the Secure World at each branching instruction. When each of these instrumented calls executes (step ❶), execution switches to the trusted *CFA* module in the Secure World to log the branch destination. The destination address (*dest*) is passed to *RESPEC-CFA*'s first submodule (step ❷). In this example, *dest* is the address `0x08246188`.

RESPEC-CFA's first submodule – **Speculate prefix** – uses the $\mathcal{V}$rf-configured prefix byte length. In the example shown in Figure 3, $prefix_{len}$ is 2 bytes. This submodule compares the prefix of *dest* to the currently active prefix ($prefix_{act}$ in Figure 3). In this example, $prefix_{act}$ is `0x0824` (in hexadecimal representation). As *dest*'s prefix matches $prefix_{act}$, it is removed from *dest* and the remaining bytes are passed to *RESPEC-CFA*'s next submodule. In this example, the suffix `0x6188` is given as output in step ❸.

RESPEC-CFA's second submodule – **Speculated encoding** – uses the $\mathcal{V}$rf-configured Huffman encoding to compress the suffix. This submodule converts the received data to its corresponding encoding(s). In this example, `0x6188` maps to the 2-bit Huffman encoding `0x3`. Therefore, in step ❹, the submodule outputs `0x3` as the final value to be appended to CF_{Log}. After appending CF_{Log}, *RESPEC-CFA* resumes the execution of `App` in step ❺.

The following sections explain the stages of this workflow in more detail.

3.3 Prefix Size Speculation Details

RESPEC-CFA leverages the locality of MCU software to reduce CF_{Log}'s size. Recall from Section 2.1 that low-end MCUs are typically equipped with limited-sized program memory (e.g., 4 to 2048KB). Within that memory, the attested application generally only makes up a small dedicated portion of it. Further, as attested software is normally statically linked (using custom linker scripts) it has

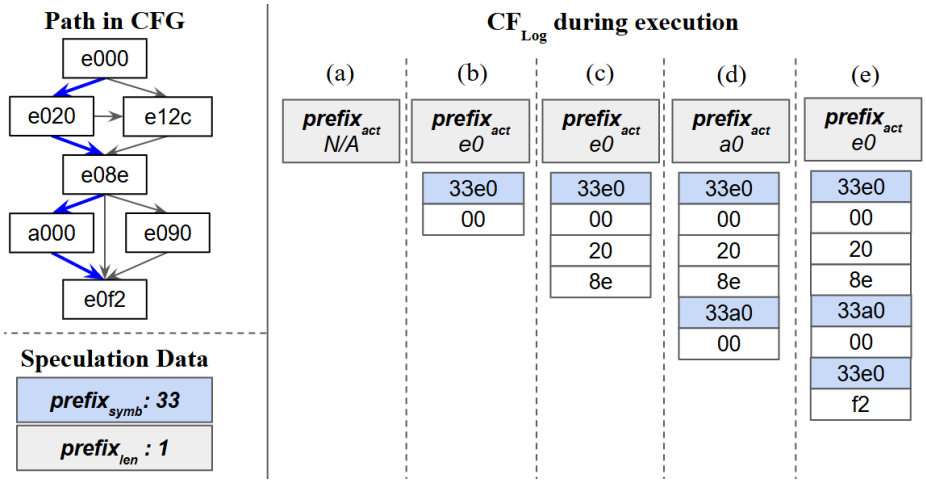


Fig. 4: Example CF_{Log} reduction due to prefix size speculation

a fixed memory location [33]. Therefore, $\mathcal{V}$rf has some prior knowledge of the attested application's memory bounds. Similarly, while some branch instructions can target arbitrary addresses (e.g., indirect jumps), most branch instructions either target a fixed memory address (e.g., direct jumps) or an offset (e.g., conditional branches) [32]. Considering these characteristics, it is likely that branch instructions within an attested application visit destination addresses that share some locality (See Section 4 for more details). Thus, it is likely that subsequent CF_{Log} entries share a common memory address prefix.

To leverage this, *RESPEC-CFA* enables $\mathcal{V}$rf to speculate on the best prefix size to use based on knowledge of the attested application's placement in program memory or analysis of a prior CF_{Log}. Upon receiving the *CFA* request, *RESPEC-CFA* saves the received prefix length ($prefix_{len}$) to Secure World memory. For the first CF_{Log} entry, *RESPEC-CFA* saves the entry's prefix as the current active prefix ($prefix_{act}$). *RESPEC-CFA* then logs the prefix alongside a reserved symbol to indicate to $\mathcal{V}$rf that this entry denotes a new prefix. After that, *RESPEC-CFA* adds the entry's suffix to the log. For each subsequent CF_{Log} entry, *RESPEC-CFA* compares the new entry's prefix to $prefix_{act}$. If the prefixes match, only the entry's suffix is added to CF_{Log}. Otherwise, the entry's prefix becomes the new $prefix_{act}$, the new prefix is added to CF_{Log} alongside the reserved prefix symbol, and the entry's suffix is added to CF_{Log}.

A demonstration of the resulting CF_{Log} due to prefix speculation is shown in Figure 4. For the sake of simplicity, this example demonstrates a Control Flow Graph (CFG) with seven nodes, each having a 16-bit start address. In this example, $\mathcal{V}$rf has selected $prefix_{len}$ of 1 and configured the reserved prefix symbol as `33`. When execution starts in (a), no $prefix_{act}$ has been determined yet. The first address is used to select the current $prefix_{act}$, which is `e0`. As such, the reserved prefix symbol is logged with $prefix_{act}$ and then the suffix is subsequently

logged. Since addresses of the same prefix are visited in (c), only their suffixes are logged. The prefix changes in consecutive control flow transitions in (d) and (e), and thus in both cases, $prefix_{act}$ is updated, the prefix symbol is logged with the new $prefix_{act}$, and the suffix is logged.

Note: If used jointly with other *CFA* optimizations that take place before *RESPEC-CFA* (e.g., loop counters [3] or SpecCFA [13]), *RESPEC-CFA*'s prefix sub-module might receive non-address inputs (i.e., already optimized entries that do not directly correspond to destination addresses). Non-address inputs are usually encoded with special symbols [12,13]. Therefore, *RESPEC-CFA* first determines if the input is an address that needs prefix speculation or a special symbol. In the latter, *RESPEC-CFA* logs the non-address without changing $prefix_{act}$.

3.4 Huffman Encoding Speculation

RESPEC-CFA also enables the optimization of CF_{Log} using speculated Huffman encodings [22]. Huffman encoding replaces fixed-length symbols with variable-length codes. The length of these codes is determined by the frequency of symbols in the data, where the more frequently a symbol occurs, the smaller its resulting code. We chose the Huffman algorithm given its optimal encoding properties [27,22]. Nonetheless, we note that any other lossless data encoding scheme of $\mathcal{V}$rf's choice can also be used. $\mathcal{V}$rf generates Huffman codes from prior CF_{Log}-s and sends the resulting encoding table to $\mathcal{P}$rv as part of the *CFA* request. Further, as new CF_{Log}-s become available, $\mathcal{V}$rf can use *CFA* requests to update the encoding table as desired. *RESPEC-CFA* uses the received encoding table to convert CF_{Log} entries to their corresponding Huffman code at runtime. The Huffman encoding table is stored in the Secure World on $\mathcal{P}$rv and protected from tampering by $\mathcal{A}$dv. Note that $\mathcal{V}$rf does not need to send an encoding table with every *CFA* request. If no new encoding table is received, *RESPEC-CFA* continues to use the existing table to encode log entries.

An example demonstrating the effect of Huffman encoding speculation is shown in Figure 5. The CFG of App is the same as the prior example in Figure 4, but now $\mathcal{V}$rf is configured with a Huffman table denoting the mapping from word to encoding, including the bit length of the encoding. In (a), the first address `e000` is encoded using the table into the bits `1011`. This behavior repeats for each control flow transition in (b)-(d). The final CF_{Log} is represented with the hex values at the end of (d), showing a compressed 3-byte CF_{Log}.

3.5 Security Analysis

We analyze *RESPEC-CFA*'s security against $\mathcal{A}$dv with capabilities outlined in Section 3.1. We argue that *RESPEC-CFA*'s additional optimization strategies do not impact the security guarantees of the underlying *CFA* architecture.

To break *CFA* guarantees, $\mathcal{A}$dv must remove evidence of malicious activity from CF_{Log}. As such, $\mathcal{A}$dv may try to modify CF_{Log} directly to remove malicious control flow transfers. However, *RESPEC-CFA* stores CF_{Log} in the Secure

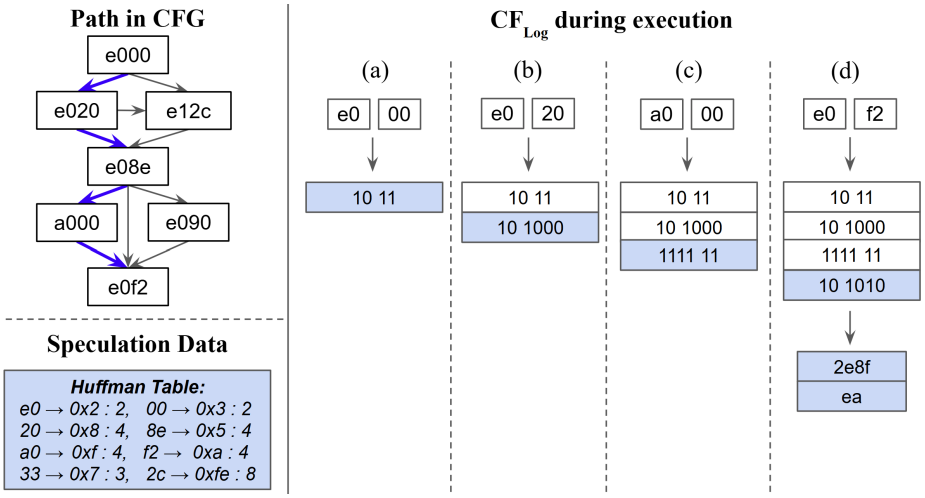


Fig. 5: Example CF_{Log} reduction due to Huffman encoding speculation

World, and thus, it is inaccessible to $\mathcal{A}$dv. $\mathcal{A}$dv could also attempt to abuse *RESPEC-CFA*'s optimizations to hide malicious activity. $\mathcal{A}$dv could try to corrupt $prefix_{act}$ to hide malicious memory prefixes or corrupt the Huffman encoding table to encode malicious values as benign entries. However, both $prefix_{act}$ and the Huffman encoding table are stored in the Secure World and are again inaccessible to $\mathcal{A}$dv. $\mathcal{A}$dv could also attempt to tamper with *RESPEC-CFA*'s implementation, but both *RESPEC-CFA*'s and the underlying *CFA* architecture's code are stored in the Secure World and protected from $\mathcal{A}$dv. Further, only the instrumented NSC calls added to the attested application can modify CF_{Log}. These instructions are protected by TrustZone's hardware and cannot be abused to log incorrect or overwrite existing CF_{Log} entries.

Finally, $\mathcal{A}$dv could attempt to impersonate $\mathcal{V}$rf and send $\mathcal{P}$rv a malicious $prefix_{len}$ or Huffman encoding table to shorten/encode malicious entries to benign values. However, this is prevented by ensuring that $\mathcal{P}$rv's RoT authenticates all $\mathcal{V}$rf requests, as described in Section 3.2. Additionally, $\mathcal{A}$dv could attempt to replay messages from $\mathcal{V}$rf to maintain outdated/incorrect encodings or prefix values. However, $\mathcal{V}$rf is authenticated based on monotonically increasing *Chal*, making replay attacks infeasible.

Feature	Geiger	GPS	Mouse	Syringe	Temp.	Ultra.
Base Address	0x080401F8	0x080401F8	0x080401F8	0x080401F8	0x080401F8	0x080401F8
End Address	0x08040E4C	0x08041CEC	0x0804106C	0x08040D2C	0x08040D1C	0x08040C5C
Size (Bytes)	3160	6904	3704	2872	2856	2664

Table 1: Sizes of test applications in program memory

4 Implementation & Evaluation

We implement *RESPEC-CFA* on a NUCLEO-L552ZE-Q development board featuring an STM32L552ZE MCU with ARM TrustZone-M support. This development board is based on ARM-Cortex-M33, operating at 110 MHz. A UART-to-USB interface with a baud rate of 38400 is used for communication with $\mathcal{V}\mathsf{rf}$. We develop *RESPEC-CFA*'s prototype by extending SpecCFA's open-source design with support for the new optimization strategies. For evaluation, we use several open-source MCU applications: an Ultrasonic Ranger [56], a Temperature Sensor [55], a Syringe Pump [71], a GPS implementation [21], a Geiger Counter [67], and a Mouse [70]. When installed in MCU memory, the selected programs spanned between 2664 and 6904 bytes of program memory. As such, all instructions shared the same 2-byte address prefix. For this reason, we configure *RESPEC-CFA* with a 2-byte prefix length optimization in our tests. Table 1 summarizes each application's location and code size. The speculated Huffman encoding is determined by generating a Huffman encoding from prior CF_{Log}-s of the evaluated applications.

We implement $\mathcal{V}\mathsf{rf}$ in Python and run it on an Ubuntu 20.04 machine. $\mathcal{V}\mathsf{rf}$ functionality is divided into two scripts. The first script generates a Huffman encoding table from prior CF_{Log}-s for a specified alphabet. Our evaluation is based on a 1-byte encoding Huffman alphabet. The second script decodes received CF_{Log}-s into their full form.

4.1 CF_{Log} Reductions of *RESPEC-CFA* in Isolation

We evaluate *RESPEC-CFA*'s impact on CF_{Log} size by comparing CF_{Log}-s generated by a baseline *CFA* architecture TRACES [10] to CF_{Log}-s generated by the same *CFA* architecture equipped with *RESPEC-CFA*. We evaluate *RESPEC-CFA* when $\mathcal{V}\mathsf{rf}$ has selected to speculate on prefixes alone, Huffman encoding alone, and both. The resulting CF_{Log} sizes for each case are shown in Figure 6.

RESPEC-CFA's prefix speculation has a theoretical upper bound based on the size of the prefix compared to the address. Since *RESPEC-CFA* prototype is built atop ARM Cortex M33 (a 32-bit – 4 byte – architecture), configuring $prefix_{len}$ as 2 bytes results in a theoretical CF_{Log} reduction upper bound of 50%. In Figure 6, this is observed, as CF_{Log}-s generated by *RESPEC-CFA*'s prefix speculation submodule alone reduce the baseline CF_{Log}-s by 48.5-49.2%.

RESPEC-CFA's Huffman encoding speculation reduces CF_{Log} by 50.8-71.5%. Speculating on Huffman encoding is beneficial for programs that change prefixes more frequently/have more varied addresses, as apparent with the Syringe Pump application in Figure 6.

RESPEC-CFA with both strategies achieves the best optimizations, reducing CF_{Log} by 68.7-90.1%. Since prefixes are optimized away before being processed by the Huffman encoding submodule, the Huffman table can be more fine-tuned to speculate on the encoding of suffixes. Thus, the two submodules complement each other and achieve higher CF_{Log} reductions together.

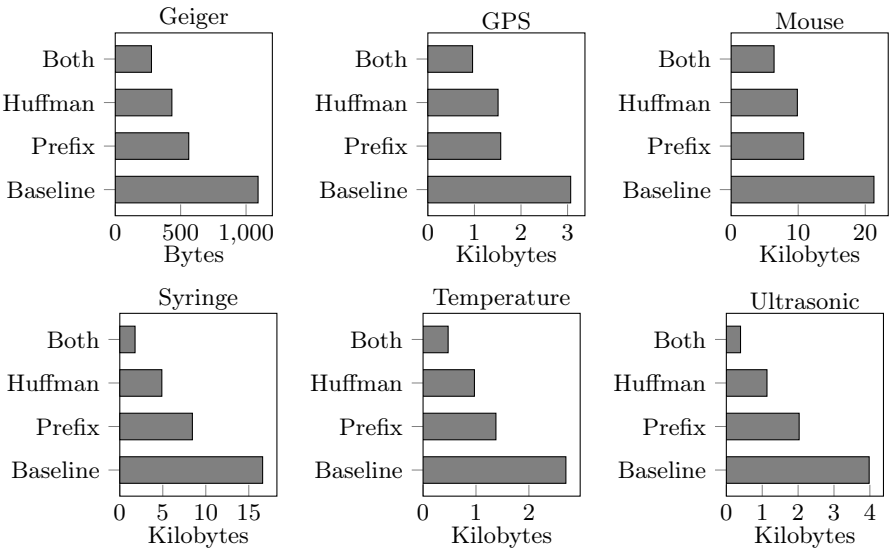


Fig. 6: CF_{Log} size: *RESPEC-CFA* vs. baseline *CFA* [39]

4.2 Combined CF_{Log} reductions of *RESPEC-CFA* + SpecCFA [13]

To demonstrate *RESPEC-CFA*'s effectiveness alongside existing *CFA* speculation strategies, we combine it with SpecCFA and measure the resulting CF_{Log} sizes. To our knowledge, SpecCFA path replacement strategy subsumes the optimizations from prior work and outperforms all other *CFA* techniques, making it an ideal candidate for integration and comparison. In this case, *RESPEC-CFA* workflow (recall Section 3.2) takes place after SpecCFA replacement of sub-paths with symbols of reduced size. We evaluate CF_{Log} sizes in the following speculation strategy scenarios:

1. Program sub-path speculation (i.e., SpecCFA) alone;
2. Program sub-path and *RESPEC-CFA*'s prefix speculation;
3. Program sub-path and *RESPEC-CFA*'s Huffman encoding speculation; and
4. All speculation strategies combined (program sub-path speculation from SpecCFA and both prefix and Huffman encoding speculation from *RESPEC-CFA*)

By default, SpecCFA supports up to 8 sub-path speculations simultaneously. Therefore, our experiments are also performed varying the number of path speculations from 1 to 8. The results are presented in Figure 7.

Regardless of whether *RESPEC-CFA* is used in its entirety or partially, it enhances SpecCFA in each of the evaluated cases. *RESPEC-CFA*'s prefix submodule enhances SpecCFA by reducing entries that are not a part of program sub-paths. This is observed in Figure 7 by achieving an additional 27.1-55.6% CF_{Log} reduction from SpecCFA to SpecCFA + prefix. Similarly, *RESPEC-CFA*'s Huffman encoding speculation alone alongside SpecCFA further reduces CF_{Log} sizes by 41.8-79.5% from SpecCFA-generated CF_{Log}-s.

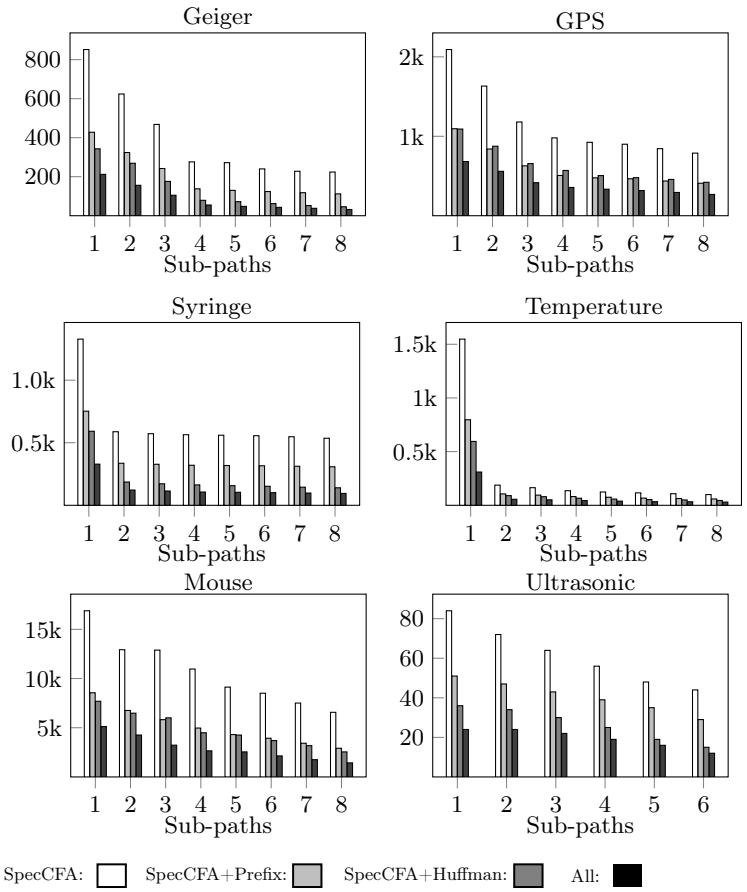
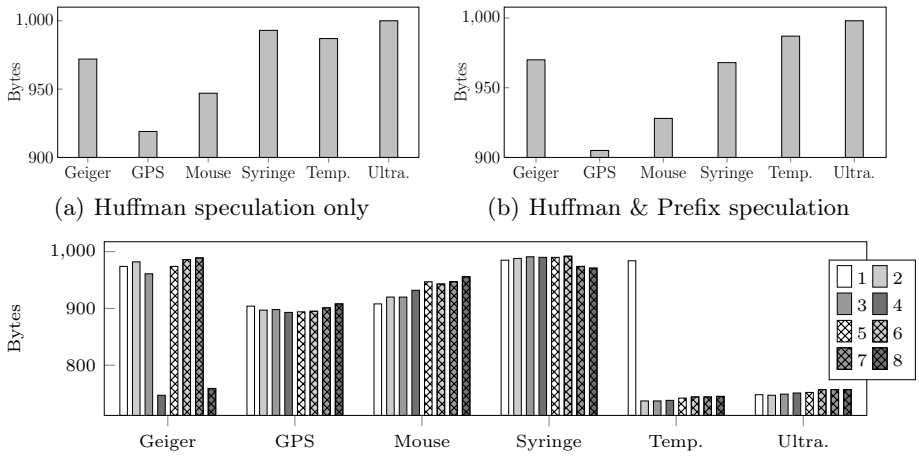


Fig. 7: Total CF_{Log} bytes after executing each application when $\mathcal{P}$rv is equipped with each speculation strategy.

Finally, the best CF_{Log} reductions are seen when *RESPEC-CFA* is fully equipped alongside SpecCFA. For the evaluated applications, *RESPEC-CFA* further reduced SpecCFA CF_{Log}-s by 63.7-85.7%. This represents a 91.5-99.7% reduction in CF_{Log} sizes for different applications, if compared to the baseline *CFA* (without any speculation-based strategy), demonstrating synergy in speculating on both CF_{Log} representation and likely sub-paths.

4.3 Trusted Computing Base (TCB) Size

RESPEC-CFA's prefix speculation submodule was implemented in 38 lines of C code, and the Huffman encoding speculation submodule was written in 70 lines of code. Additionally, *RESPEC-CFA* required 26 lines of C code to integrate into SpecCFA. Therefore, *RESPEC-CFA* in its entirety contributes to a TCB



(a) Huffman speculation only

(b) Huffman & Prefix speculation

(c) Huffman, Prefix, and SpecCFA speculation

Fig. 8: Total Huffman table size on $\mathcal{P}\mathsf{rv}$ for different applications and *RESPEC-CFA* configurations

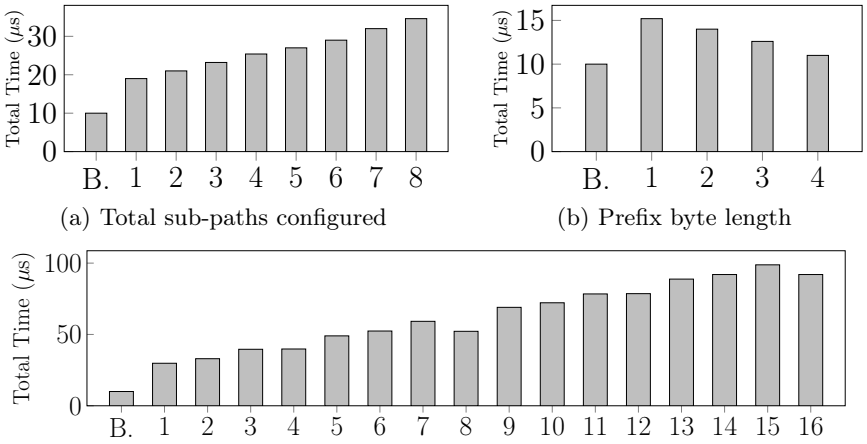
size increase of 134 lines of C code. This correlates to an additional 1140 bytes of Secure World program memory.

4.4 Memory Overhead

RESPEC-CFA also requires some Secure World data memory to store the speculation metadata. When speculating on instruction locality, *RESPEC-CFA* must store the active prefix and its length (1 byte). As a prefix is always shorter than 4 bytes (given ARM Cortex-M 32-bit architecture), the prefix metadata can be stored in at most 5 bytes.

Speculating on Huffman codes has a larger memory impact due to storing the Huffman encoding table. Figure 8 depicts the total size of the Huffman table for the tested *RESPEC-CFA* configurations. In our experiments, we used a 1-byte symbol alphabet to generate Huffman codes, resulting in 256 table entries. Each entry is composed of the encoding and its length. The size of Huffman codes varies depending on the attested application and other optimizations enabled (e.g., SpecCFA or *RESPEC-CFA*'s prefix speculation). Due to this, the total size of Huffman codes ranged from 481 to 744 bytes across all tests. The length of each code is represented as a single byte, resulting in an additional 256 bytes of overhead. Therefore, when combined, the Huffman table overhead spanned from 737 to 1000 bytes of additional memory overhead in our experiments.

While the size of the Huffman table does vary, the overhead generally is fairly consistent for the evaluated applications, best shown in Figure 8(c). However, in some cases, the size of the Huffman table can change drastically. This sudden change in size is due to the relative frequency of data in the CF_{Log}-s used to generate the table. As mentioned in Section 3.4, the more often a symbol ap-



(a) Total sub-paths configured

(b) Prefix byte length

(c) Huffman Encoding bit length

Fig. 9: Average worst-case added NSC time per log entry for varying (a) numbers of sub-paths, (b) prefix lengths, and (c) Huffman alphabet lengths (B. = Baseline)

pears in the dataset (i.e., a given address in CF_{Log}), the smaller its resulting Huffman code. Specifically, Huffman codes are generated using a binary Huffman tree where more frequent symbols are stored higher in the tree [1]. As a consequence, the higher up the tree a symbol appears, the smaller its encoding, but also the less balanced the tree becomes. Therefore, as the input data becomes more disproportional, so does the length of encodings in the resulting table. Thus, the Huffman table's size greatly depends on the distribution of entries in CF_{Log}. Changes in input CF_{Log}-s due to other optimizations (e.g., SpecCFA) can greatly alter this distribution, leading to the jumps in Huffman table size seen in Figure 8(c).

4.5 Runtime Overhead

The best CF_{Log} reductions are achieved with *RESPEC-CFA* and SpecCFA combined. However, the additional submodules added to the Secure World execute upon each NSC. As a result, the execution time increases due to NSC handling. We evaluate this in two ways. First, we measure the additional time per NSC call. Second, we evaluate the cumulative overhead caused by all NSC calls during the execution of each evaluated application.

Additional NSC Time.

Figure 9 shows the average NSC time to process one entry on applications crafted to target the worst-case timing for each Secure World submodule: SpecCFA, prefix speculation, and Huffman encoding speculation.

Figure 9(a) shows the worst-case time to speculate on sub-paths by SpecCFA, the baseline when *RESPEC-CFA* extends it. To create the worst-case scenario, *RESPEC-CFA* varies the total number of sub-path speculations and configures

them so all sub-paths mismatch except for the last configured sub-path (i.e., when configured with 8 sub-path speculations, all mismatch except for sub-path 8). In this case, there is an initial $\approx 9\mu$s increase from baseline to 1 sub-path. After that, there is a linear increase of $\approx 2.22\mu$s per additional sub-path.

Figure 9(b) shows the worst-case time to speculate on memory address prefixes. For the worst-case application, we craft a program that constantly crosses the configured prefix range. As described in Section 3.3, a special ID is logged to denote a change of prefix. Since this ID is the same length as the remaining suffix, *RESPEC-CFA* suffers more runtime overhead when the prefix is shorter. This is because the ID is longer and is logged more often in this worst-case scenario. However, this scenario is unlikely since $\mathcal{V}\mathsf{rf}$ would configure $prefix_{len}$ based on the anticipated behavior.

Finally, Figure 9(c) shows the worst-case time to speculate on a Huffman encoding, which occurs when each byte in the address uses the longest bit-length code from the Huffman table. To examine the impact of encoding length, we measure the time for encoding with encoding lengths from 1 to 16 bits. As shown in Figure 9(c), the total added time generally increases with the bit length. However, at bit lengths that are multiples of 4, the time improves due to architectural characteristics that enhance the performance on even bytes/half-bytes rather than on uneven bit lengths that do not align in this way.

Additional Time for End-to-End *CFA*.

To determine *RESPEC-CFA*'s impact on end-to-end attestations times, we compare the total runtime of *RESPEC-CFA* equipped attestations with those performed by the underlying baseline (i.e., unoptimized) *CFA* architecture. Specifically, we evaluate *RESPEC-CFA* with prefix speculation, Huffman encoding, and both enabled. We timed each configuration 20 times and report the average runtime across experiments. These average runtimes are displayed in Figure 10.

Across all applications, *RESPEC-CFA* achieved significant end-to-end time reductions. When using prefix speculation alone, *RESPEC-CFA* achieved reductions of 40.7%-49.2%. Similarly, when using Huffman encoding alone, *RESPEC-CFA* saw larger total runtime reductions of 46.1%-68.7%. Finally, when using both speculation strategies, *RESPEC-CFA* reduced the total runtime by 61%-87.5% depending on the application. These runtime savings stem directly from the cost of transmitting CF_{Log}. Recall from Section 2.4 that some *CFA* architectures (including the baseline used in this work) transmit CF_{Log} as multiple CF_{Log} slices throughout the attestation process. As such, the more slices $\mathcal{P}\mathsf{rv}$ must send, the more time is spent performing slower transmission operations. Thus, by reducing the amount of data that must be stored, *RESPEC-CFA* also reduces the number of CF_{Log} slices that must be sent to $\mathcal{V}\mathsf{rf}$, leading to the observed runtime reductions.

We also evaluated *RESPEC-CFA*'s impact on the end-to-end attestation time of each application when used in conjunction with SpecCFA. In these experiments, we attested each application using SpecCFA, SpecCFA with *RESPEC-CFA*'s prefix speculation, SpecCFA with *RESPEC-CFA*'s Huffman encoding,

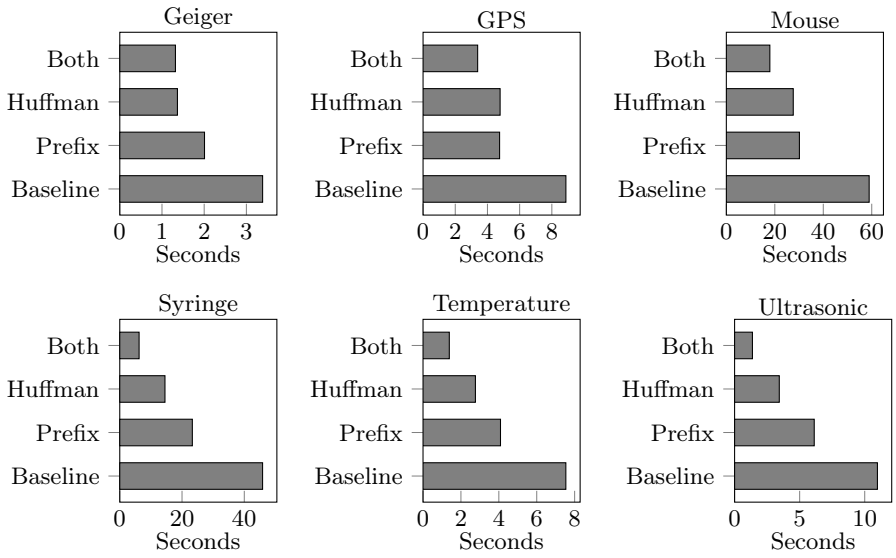


Fig. 10: End-to-End attested execution time: *RESPEC-CFA* vs. baseline *CFA*

and SpecCFA with both *RESPEC-CFA* optimization strategies. For these experiments, SpecCFA was configured with support for 8 sub-path speculations, and tests were again repeated 20 times, with the average attestation time being reported. A summary of these results is depicted in Figure 11.

When combined with SpecCFA, *RESPEC-CFA*'s impact on $\mathcal{P}\mathsf{rv}$'s overall runtime varied widely across applications. For the Geiger Counter, Temperature Sensor, and Ultrasonic Sensor, *RESPEC-CFA* had a negligible effect on the application's runtime. In these experiments, *RESPEC-CFA* decreased the runtime by -5.8% to 6.8%. While some configurations did increase the runtime of the system, this is with respect to SpecCFA alone. Even at its worst, -5.8% with respect to SpecCFA, this represents a decrease of 93.6% from the baseline *CFA* architecture. For the remaining applications, *RESPEC-CFA* achieved larger reductions of 10.9%-75.6%. For these applications, *RESPEC-CFA* could further reduce the number of CF_{Log} slices sent to $\mathcal{V}\mathsf{rf}$, resulting in the larger impact seen. Regardless, when used in conjunction with existing *CFA* optimization strategies, *RESPEC-CFA* achieves comparable or better end-to-end time reductions.

5 Discussion

5.1 Worst Case Scenarios.

The speculation strategies presented rely on prior CF_{Log}-s to generate the appropriate encodings. SpecCFA [13] explored static analysis as an alternative to

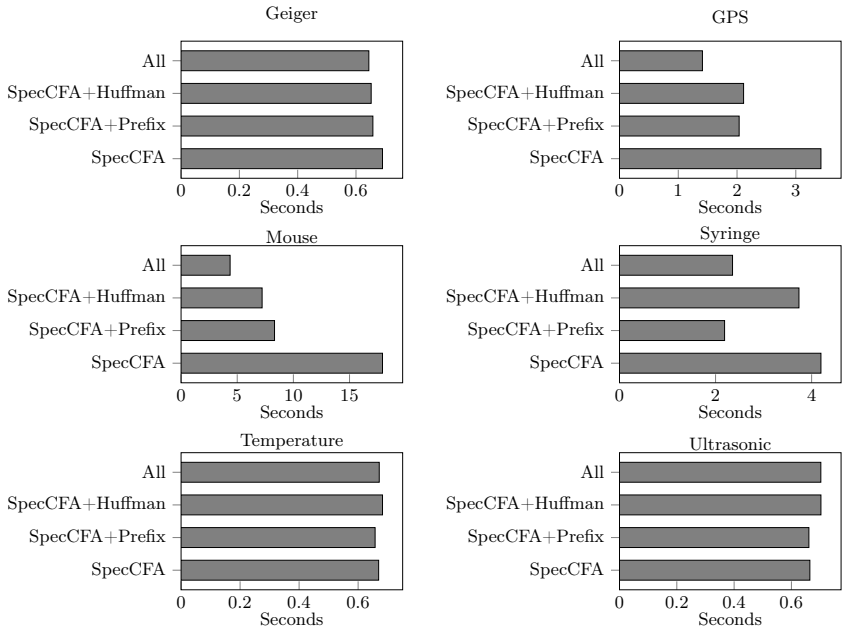


Fig. 11: End-to-End attestation time: *RESPEC-CFA* plus SpecCFA

enable speculations without prior CF_{Log}-s. However, they found static analysis-based speculations performed worse than speculations generated from CF_{Log} analysis. Thus, a worst-case scenario occurs when no prior CF_{Log} exists yet/is available. Without prior context, neither strategy can accurately predict the application's behavior, resulting in no/minimal savings. If no prior CF_{Log}-s exist, *RESPEC-CFA* defaults to the underlying *CFA* behavior without *RESPEC-CFA* optimization. After obtaining a first CF_{Log} slice, speculations can be generated normally from it and subsequent CF_{Log} slices.

The prefix speculation strategy uses a $\mathcal{V}$rf-defined prefix length to optimize CF_{Log} based on the common locality of branch destinations. If a suboptimal length is chosen (e.g., too long), it is more likely that subsequent CF_{Log} entries will not share a common prefix resulting in more CF_{Log} prefix entries and lower savings. While in theory possible, this scenario is in practice very unlikely due to the simplicity of finding common prefixes in CF_{Log}.

Savings due to Huffman encoding depend on high-frequency symbols in the alphabet. Thus, if symbols are uniformly distributed in CF_{Log}, no savings would occur. Similarly, if a particular CF_{Log} has a large number of uncommon symbols, savings gained from the Huffman encoding may be counteracted by the larger encoding of rarer symbols. Fortunately, both scenarios are unlikely due to the type of data in CF_{Log}, i.e., branch destinations that have small cardinality (a subset of program memory's addresses) and occur repetitively.

RESPEC-CFA with Huffman and prefix strategies in combination (or alongside other speculation strategies, such as SpecCFA [13]) can further reduce the likelihood of the above worst-case scenarios as they cover each other's worst cases. In the case of poor prefixing, each additional prefix entry adds repeated symbols to CF_{Log}. Thus, Huffman encoding would replace these entries with smaller symbols minimizing their impact. Similarly, since prefixing removes repeated portions of memory addresses in CF_{Log}, Huffman encoding can better optimize the remaining symbols.

Lastly, in some cases, a Huffman table may become larger than the savings it yields in a single *CFA* instance. However, since the same table can be reused across multiple *CFA* responses, the protocol bandwidth savings grow linearly with the number of protocol instances while the storage cost remains constant. Thus, Huffman encoding is still likely to be cost-effective over multiple instances (i.e., over time).

5.2 *RESPEC-CFA* with Interrupts

Embedded applications often rely on interrupts for real-time event handling. When an interrupt occurs, the application is paused and execution jumps to an associated ISR to handle the event. Once the ISR is finished, execution returns to the program and the application resumes. Therefore, interrupts affect an application's control flow paths. *RESPEC-CFA* inherits support for interrupts from the underlying *CFA* architecture it builds upon. Some *CFA* schemes allow interrupts but do not log them to CF_{Log} [39]. In this case, interrupts do not affect *RESPEC-CFA*'s speculation strategies as they do not appear in CF_{Log}. For architectures that record interrupts [12,66,63], *RESPEC-CFA* can speculate on interrupts similar to regular branch addresses in CF_{Log}.

5.3 *RESPEC-CFA* in High-End Systems

As discussed in Section 3.1, *RESPEC-CFA* is envisioned for MCUs with limited memory and resources to transmit large CF_{Log}-s. Albeit not designed for high-end devices (or complex Systems-on-Chip), *RESPEC-CFA* concepts should also apply in that setting. Larger systems have larger applications and thus more varied CF_{Log} entries. Yet, certain instructions/addresses will still occur more often than others. Therefore, Huffman encoding and prefix speculations would still result in savings. That said, regardless of conceptual applicability, in a high-end system, the cost to compute Huffman encodings or determine common prefixes on the fly (or in parallel) might be relatively small or negligible. This could obviate the demand for $\mathcal{V}\mathsf{rf}$-based path speculation observed in MCUs.

6 Conclusion & Future Work

We propose *RESPEC-CFA* to enable speculation and CF_{Log} optimization based on two new properties. First, *RESPEC-CFA* allows $\mathcal{V}\mathsf{rf}$ to speculate on the locality of branch destinations, reducing CF_{Log} size based on shared prefixes across

sequences of destinations. Second, *RESPEC-CFA* enables speculation on the Huffman encoding of CF_{Log}, replacing entries with their corresponding Huffman code at CF_{Log} construction time. We implement an open-source *RESPEC-CFA* design and evaluate it [68]. Our experiments show that *RESPEC-CFA* results in significant CF_{Log} reductions with little runtime cost. When coupled with prior work in SpecCFA [13], further savings are obtained.

Future work on *RESPEC-CFA* spans several directions. One avenue is developing static analysis techniques for speculation, which would allow $\mathcal{V}\mathsf{rf}$ to generate initial speculations directly from source code or binaries rather than relying on prior execution logs. This poses the challenge of tuning these speculations without runtime context, requiring methods that can reason about data and control flow representations statically. Another promising direction involves hardware integration, where custom hardware extensions could reduce runtime and memory overheads while enabling efficient support for *RESPEC-CFA* on lower-end MCUs. A central challenge here is representing Huffman encoding tables in a hardware-efficient manner.

Further exploration possibilities include alternative encodings and alphabets, such as arithmetic coding [75], which may better support large alphabets without imposing excessive storage costs. Leveraging application-specific knowledge (e.g., valid control flow targets) could also help reduce encoding complexity, though careful handling of unexpected addresses remains necessary for attack detection. Finally, extending speculation to data flow attestation (DFA) [15,46,4], which extends *CFA* to also include data flow events in hopes of detecting non-control data-only attacks, is also an interesting area for future work.

Acknowledgments. We thank ACNS anonymous reviewers and shepherd for their constructive comments.

References

1. Ece264: Huffman coding. `https://engineering.purdue.edu/ece264/17au/hw/HW13?alt=huffman` (2017)
2. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. TISSEC **13**(1), 1–40 (2009)
3. Abera, T., Asokan, N., Davi, L., Ekberg, J.E., Nyman, T., Paverd, A., Sadeghi, A.R., Tsudik, G.: C-FLAT: control-flow attestation for embedded systems software. In: CCS. pp. 743–754. ACM (2016)
4. Abera, T., Bahmani, R., Brasser, F., Ibrahim, A., Sadeghi, A.R., Schunter, M.: DIAT: Data integrity attestation for resilient collaboration of autonomous systems. In: NDSS (2019)
5. Ammar, M., Abdelraoof, A., Vlasceanu, S.: On bridging the gap between control flow integrity and attestation schemes. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 6633–6650 (2024)
6. Ammar, M., Caulfield, A., Nunes, I.D.O.: Sok: Integrity, attestation, and auditing of program execution. In: S&P. pp. 3255–3272. IEEE (2025)
7. Brasser, F., El Mahjoub, B., Sadeghi, A.R., Wachsmann, C., Koeberl, P.: TyTAN: Tiny trust anchor for tiny devices. In: DAC. pp. 1–6 (2015)

8. Brasser, F., Rasmussen, K.B., Sadeghi, A.R., Tsudik, G.: Remote attestation for low-end embedded devices: the prover's perspective. In: DAC (2016)
9. Castelluccia, C., Francillon, A., Perito, D., Soriente, C.: On the difficulty of software-based attestation of embedded devices. In: CCS. pp. 400–409. CCS '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1653662.1653711, http://doi.acm.org/10.1145/1653662.1653711
10. Caulfield, A., Neto, A.J., Rattanavipanon, N., Nunes, I.D.O.: TRACES: Tee-based runtime auditing for commodity embedded systems. ACSAC (2024)
11. Caulfield, A., Rattanavipanon, N., De Oliveira Nunes, I.: ASAP: reconciling asynchronous real-time operations and proofs of execution in simple embedded systems. In: Proceedings of the 59th ACM/IEEE Design Automation Conference. pp. 721–726 (2022)
12. Caulfield, A., Rattanavipanon, N., Nunes, I.D.O.: ACFA: Secure runtime auditing & guaranteed device healing via active control flow attestation. In: USENIX Security. pp. 5827–5844 (2023)
13. Caulfield, A., Tyler, L., Nunes, I.D.O.: SpecCFA: Enhancing control flow attestation/auditing via application-aware sub-path speculation. ACSAC (2024)
14. De Oliveira Nunes, I., Jakkamsetti, S., Kim, Y., Tsudik, G.: Casu: Compromise avoidance via secure update for low-end embedded systems. In: Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design. pp. 1–9 (2022)
15. Dessouky, G., Abera, T., Ibrahim, A., Sadeghi, A.R.: LiteHAX: lightweight hardware-assisted attestation of program execution. In: ICCAD. pp. 1–8. IEEE (2018)
16. Dessouky, G., Zeitouni, S., Ibrahim, A., Davi, L., Sadeghi, A.R.: CHASE: A configurable hardware-assisted security extension for real-time systems. In: ICCAD. pp. 1–8. IEEE (2019)
17. Dessouky, G., Zeitouni, S., Nyman, T., Paverd, A., Davi, L., Koeberl, P., Asokan, N., Sadeghi, A.R.: LO-FAT: Low-overhead control flow attestation in hardware. In: DAC. pp. 1–6 (2017)
18. Eldefrawy, K., Tsudik, G., Francillon, A., Perito, D.: SMART: Secure and minimal architecture for (establishing dynamic) root of trust. In: NDSS. vol. 12, pp. 1–15 (2012)
19. Ge, X., Cui, W., Jaeger, T.: GRIFFIN: Guarding control flows using intel processor trace. ACM SIGPLAN Notices **52**(4), 585–598 (2017)
20. Goh, W., Dannenberg, A., He, J.: Application report: Msp430 fram technology - how to and best practices. https://www.ti.com/lit/an/slaa628b/slaa628b.pdf (2014)
21. Hart, M.: Tinygps++. http://arduiniana.org/libraries/tinygpsplus/ (2014)
22. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proceedings of the IRE **40**(9), 1098–1101 (1956)
23. Instruments, T.: Application report slaa685: Msp code protection features. https://www.ti.com/lit/an/slaa685/slaa685.pdf (2015)
24. Instruments, T.: Msp430 microcontrollers. https://www.ti.com/microcontrollers-mcus-processors/msp430-microcontrollers/overview.html (Accessed: 2025)
25. Kayan, H., Nunes, M., Rana, O., Burnap, P., Perera, C.: Cybersecurity of industrial cyber-physical systems: a review. CSUR (2022)
26. Kovah, X., Kallenberg, C., Weathers, C., Herzog, A., Albin, M., Butterworth, J.: New results for timing-based attestation. In: S&P. pp. 239–253. IEEE (2012)

27. Lelewer, D.A., Hirschberg, D.S.: Data compression. ACM Computing Surveys (CSUR) **19**(3), 261–296 (1987)
28. Ltd, A.: ARM security technology - building a secure system using TrustZone technology. https://developer.arm.com/documentation/PRD29-GENC-009492/latest/ (2009)
29. Ltd, A.: Arm cortex-m33 devices generic user guide. https://developer.arm.com/documentation/100235/0004/the-cortex-m33-peripherals/nested-vectored-interrupt-controller (2018), section: Nested Vectored Interrupt Controller
30. Ltd, A.: Arm cortex-m7 processor technical reference manual. https://developer.arm.com/documentation/ddi0489/f/nested-vectored-interrupt-controller/nvic-functional-description (2018), section: NVIC functional description
31. Ltd, A.: Trustzone technology for armv8-m architecture version 2.1. https://developer.arm.com/documentation/100690/0201/ (2019)
32. Ltd, A.: Armv8-m architecture reference manual. https://developer.arm.com/documentation/ddi0553/latest (2023), section C1.4.6
33. Ltd, A.: Introduction to the armv8-m architecture and its programmers model user guide. https://developer.arm.com/documentation/107656/0101/Getting-started-with-Armv8-M-based-systems/Arm-Compiler-for-Embedded/Application-development (2023), section: Application development
34. Ltd, A.: Trustzone technology microcontroller system hardware design concepts user guide. https://developer.arm.com/documentation/107779/0100/Implementation-Defined-Attribution-Unit--IDAU-/Armv8-M-Processors (2023), section: Armv8-M Processors
35. Ltd, A.: Arm cortex-m33 processor technical reference manual. https://developer.arm.com/documentation/100230/latest (2024)
36. Luo, L., Shao, X., Ling, Z., Yan, H., Wei, Y., Fu, X.: faslr: Function-based aslr via trustzone-m and mpu for resource-constrained iot systems. IEEE Internet of Things Journal **9**(18), 17120–17135 (2022)
37. Microchip: Avr microcontrollers (mcus). https://www.microchip.com/en-us/products/microcontrollers/8-bit-mcus/avr-mcus (Accessed: 2025)
38. Neto, A.J., Caulfield, A., Nunes, I.D.O.: RAP-Track: Efficient control flow attestation via parallel tracking in commodity mcus. In: 2025 62nd ACM/IEEE Design Automation Conference (DAC). pp. 1–7. IEEE (2025)
39. Neto, A.J., Nunes, I.D.O.: ISC-FLAT: On the conflict between control flow attestation and real-time operations. In: RTAS. pp. 133–146. IEEE (2023)
40. Neto, A.J., Rattanavipanon, N., Nunes, I.D.O.: PEARTS: Provable execution in real-time embedded systems. In: 2025 IEEE Symposium on Security and Privacy (SP). pp. 3765–3782. IEEE (2025)
41. Noorman, J., Bulck, J.V., Mühlberg, J.T., Piessens, F., Maene, P., Preneel, B., Verbauwhede, I., Götzfried, J., Müller, T., Freiling, F.: Sancus 2.0: A low-cost security architecture for iot devices. TOPS **20**(3), 1–33 (2017)
42. Nunes, I.D.O., Eldefrawy, K., Rattanavipanon, N., Steiner, M., Tsudik, G.: VRASED: A verified Hardware/Software Co-Design for remote attestation. In: USENIX Security. pp. 1429–1446 (2019)
43. Nunes, I.D.O., Eldefrawy, K., Rattanavipanon, N., Tsudik, G.: APEX: A verified architecture for proofs of execution on remote devices under full software compromise. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 771–788 (2020)

44. Nunes, I.D.O., Hwang, S., Jakkamsetti, S., Tsudik, G.: Privacy-from-birth: Protecting sensed data from malicious sensors with versa. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 2413–2429. IEEE (2022)
45. Nunes, I.D.O., Jakkamsetti, S., Rattanavipanon, N., Tsudik, G.: On the toctou problem in remote attestation. arXiv preprint arXiv:2005.03873 (2020)
46. Nunes, I.D.O., Jakkamsetti, S., Tsudik, G.: DIALED: Data integrity attestation for low-end embedded devices. In: DAC. pp. 313–318. IEEE (2021)
47. Nunes, I.D.O., Jakkamsetti, S., Tsudik, G.: Tiny-CFA: Minimalistic control-flow attestation using verified proofs of execution. In: DATE. pp. 641–646. IEEE (2021)
48. Nyman, T., Ekberg, J.E., Davi, L., Asokan, N.: Cfi care: Hardware-supported call and return enforcement for commercial microcontrollers. In: RAID. pp. 259–284. Springer (2017)
49. Pan, R., Parmer, G.: SBIs: Application access to safe, baremetal interrupt latencies. In: RTAS. pp. 82–94. IEEE (2022)
50. Pinto, S., Santos, N.: Demystifying arm trustzone: A comprehensive survey. CSUR **51**(6), 1–36 (2019)
51. Pinto, S., Araujo, H., Oliveira, D., Martins, J., Tavares, A.: Virtualization on trustzone-enabled microcontrollers? voilà! In: RTAS. pp. 293–304 (2019). https://doi.org/10.1109/RTAS.2019.00032
52. Ramalingam, G.: The undecidability of aliasing. TOPLAS **16**(5), 1467–1471 (1994)
53. Schellekens, D., Wyseur, B., Preneel, B.: Remote attestation on legacy operating systems with trusted platform modules. Science of Computer Programming **74**(1-2), 13–22 (2008)
54. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In: S&P. pp. 745–762. IEEE (2015)
55. Seeed-Studio: Temperature Sensor Github Repository. `https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/temp_humi_sensor` (2022)
56. Seeed-Studio: Ultrasonic Ranger Github Repository. `https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/ultrasonic_ranger` (2022)
57. Seshadri, A., Luk, M., Perrig, A.: SAKE: Software attestation for key establishment in sensor networks. In: DCOSS, pp. 372–385 (2008)
58. Seshadri, A., Luk, M., Shi, E., Perrig, A., Van Doorn, L., Khosla, P.: Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In: SOSP. pp. 1–16. ACM (2005)
59. Seshadri, A., Perrig, A., Van Doorn, L., Khosla, P.: SWATT: Software-based attestation for embedded devices. In: S&P. pp. 272–282. IEEE (2004)
60. Seto, A., Duran, O.K., Amer, S., Chuang, J., van Schaik, S., Genkin, D., Garman, C.: Wiretap: Breaking server sgx via dram bus interposition. In: 2025 SIGSAC Conference on Computer and Communications Security (CCS '25). Association for Computing Machinery (2025), `https://wiretap.fail`
61. STMicroelectronics: Arm cortex-m0 in a nutshell. `https://www.st.com/content/st_com/en/arm-32-bit-microcontrollers/arm-cortex-m0.html` (Accessed: 2025)
62. STMicroelectronics: Arm cortex-m7 in a nutshell. `https://www.st.com/content/st_com/en/arm-32-bit-microcontrollers/arm-cortex-m7.html` (Accessed: 2025)
63. Sun, Z., Feng, B., Lu, L., Jha, S.: OAT: Attesting operation integrity of embedded devices. In: S&P. pp. 1433–1449. IEEE (2020)

64. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: S&P. pp. 48–62. IEEE (2013)
65. Tan, X., Zhao, Z.: SHERLOC: Secure and holistic control-flow violation detection on embedded systems. In: CCS. pp. 1332–1346. ACM (2023)
66. Toffalini, F., Losiouk, E., Biondo, A., Zhou, J., Conti, M.: ScaRR: Scalable runtime remote attestation for complex systems. In: RAID. pp. 121–134 (2019)
67. Tournade, Y.: ArduinoPocketGeiger Github Repository. `https://github.com/MonsieurV/ArduinoPocketGeiger` (2020)
68. Tyler, L., Caulfield, A., De Oliveira Nunes, I.: RESPEC-CFA prototype repository. `https://github.com/SPINS-RG/RESPEC-CFA` (Jan 2026)
69. Van Der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., Giuffrida, C.: Drammer: Deterministic rowhammer attacks on mobile platforms. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. pp. 1675–1689 (2016)
70. Vlasák, M.: arduino-joystick-mouse. `https://github.com/Krakenus/arduino-joystick-mouse/blob/master/joystick_mouse.ino` (2019)
71. Walker, T.: OpenSyringePump Github Repository. `https://github.com/manimino/OpenSyringePump` (2022)
72. Wang, J., Li, A., Li, H., Lu, C., Zhang, N.: RT-TEE: Real-time system availability for cyber-physical systems using arm trustzone. In: S&P. pp. 352–369 (2022). https://doi.org/10.1109/SP46214.2022.9833604
73. Wang, J., Wang, Y., Li, A., Xiao, Y., Zhang, R., Lou, W., Hou, Y.T., Zhang, N.: ARI: Attestation of real-time mission execution integrity. In: USENIX Security. pp. 2761–2778 (2023)
74. Wang, Y., Mack, C.L., Tan, X., Zhang, N., Zhao, Z., Baruah, S., Ward, B.C.: InsectACIDE: Debugger-based holistic asynchronous cfi for embedded system. In: RTAS. pp. 360–372. IEEE (2024)
75. Witten, I.H., Neal, R.M., Cleary, J.G.: Arithmetic coding for data compression. Communications of the ACM **30**(6), 520–540 (1987)
76. Wood, A.D., Stankovic, J.A.: Denial of service in sensor networks. computer **35**(10), 54–62 (2002)
77. Yadav, N., Ganapathy, V.: Whole-program control-flow path attestation. In: CCS. pp. 2680–2694. ACM (2023)
78. Zeitouni, S., Dessouky, G., Arias, O., Sullivan, D., Ibrahim, A., Jin, Y., Sadeghi, A.R.: ATRIUM: Runtime attestation resilient under memory attacks. In: ICCAD. pp. 384–391. IEEE (2017)
79. Zhang, Y., Liu, X., Sun, C., Zeng, D., Tan, G., Kan, X., Ma, S.: ReCFA: Resilient control-flow attestation. In: ACSAC. pp. 311–322. ACM (2021)