SpecCFA: Enhancing Control Flow Attestation/Auditing via Application-Aware Sub-Path Speculation

Adam Caulfield, Liam Tyler, and Ivan De Oliveira Nunes
Rochester Institute of Technology, USA

***Abstract*—At the edge of modern cyber-physical systems, Micro-Controller Units (MCUs) are responsible for safety-critical sensing/actuation. However, MCU cost constraints rule out the usual security mechanisms of general-purpose computers. Thus, various low-cost security architectures have been proposed to remotely verify MCU software integrity. Control Flow Attestation (*CFA*) enables a Verifier ($\mathcal{V}$rf) to remotely assess the run-time behavior of a prover MCU ($\mathcal{P}$rv), generating an authenticated trace of all of $\mathcal{P}$rv control flow transfers (CF_{Log}). Further, Control Flow Auditing architectures augment *CFA* by guaranteeing the delivery of evidence to $\mathcal{V}$rf.**

Unfortunately, a limitation of existing *CFA* lies in the cost to store and transmit CF_{Log}, as even simple MCU software may generate large traces. Given these issues, prior work has proposed static (context-insensitive) optimizations. However, they do not support configurable program-specific optimizations. In this work, we note that programs may produce unique predictable control flow sub-paths and argue that program-specific predictability can be leveraged to dynamically optimize *CFA* while retaining all security guarantees. Therefore, we propose *SpecCFA*: an approach for dynamic sub-path speculation in *CFA*. *SpecCFA* allows $\mathcal{V}$rf to securely speculate on likely control flow sub-paths for each attested program. At run-time, when a sub-path in CF_{Log} matches a pre-defined speculation, the entire sub-path is replaced by a reserved symbol. *SpecCFA* can speculate on multiple variable-length control flow sub-paths simultaneously. We implement *SpecCFA* atop two open-source control flow auditing architectures: one based on a custom hardware design [1] and one based on a commodity Trusted Execution Environment (ARM TrustZone-M) [2]. In both cases, *SpecCFA* significantly lowers storage/performance costs that are critical to resource-constrained MCUs.

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

1. Introduction

Micro-Controller Units (MCUs) are part of cyber-physical systems and implement the *de-facto* interface between the physical and digital worlds. Therefore, they are relied upon to implement safety-critical sensing and actuation tasks [3], [4], [5]. However, due to energy, size, and cost constraints, MCUs lack security features common to general-purpose computers. In particular, they usually run software at bare-metal, lacking Memory Management Units (MMUs), fine-grained inter-process isolation, or strong privilege separation. Given their importance to modern systems and lack of security features, MCUs have become attractive targets for attacks [6], [7].

In this context, Remote Attestation (*RA*) [5], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31] and Proofs of Execution (*PoX*) [32], [33] were proposed as inexpensive means to detect software compromises on MCUs. In a typical *RA* protocol, a *Verifier* ($\mathcal{V}$rf) aims to determine if the software state of a remote *Prover* MCU ($\mathcal{P}$rv) is trustworthy. *RA* produces an authenticated and unforgeable "snapshot" of $\mathcal{P}$rv's current software image. *PoX* builds atop *RA* to also prove that a particular function within this software image has been executed in a timely manner.

On the other hand, neither *RA* nor *PoX* offers evidence about the order in which instructions have been executed. Thus, out-of-order execution attacks, including control flow hijacking [34], [35], Return-Oriented Programming (ROP) [36], and Jump-Oriented Programming (JOP) [37], remain oblivious to $\mathcal{V}$rf with *RA* and *PoX*.

While Control Flow Integrity (CFI) [38], [39], [40], [41], [42], [43], [44] can detect some attacks locally at $\mathcal{P}$rv, they do not provide $\mathcal{V}$rf with evidence about the malicious control flow path taken, precluding analysis of the anomalous behavior [45]. Therefore, Control Flow Attestation (*CFA*) [2], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56] was proposed to provide precise evidence of the execution's control flow path to a remote $\mathcal{V}$rf. Until recently, *CFA* was limited by the fact that a fully compromised $\mathcal{P}$rv could ignore $\mathcal{V}$rf requests or refuse to deliver a CF_{Log} that indicates a compromised state. This problem is recognized and addressed in recent work [1] by incorporating reliable delivery of CF_{Log} as part of the *CFA*'s root of trust (RoT) functionality to enable "control flow auditing" as opposed to best-effort *CFA*. We revisit *CFA* details in Sec. 2.

Regardless of specifics, *CF*-Attestation/-Auditing [1] requires the storage of CF_{Log} and its eventual transmission to $\mathcal{V}$rf, which becomes a bottleneck on the resource-constrained $\mathcal{P}$rv. For this reason, earlier *CFA* methods [46], [50], [53] proposed to build CF_{Log} as a hash-chain to

1. In the remainder of this paper (unless explicitly stated) we use *CFA* to refer to both Control Flow Attestation and Control Flow Auditing schemes.

compress the sequence of control flow transfers into a single hash digest. While this approach minimizes storage/transmission costs, it requires $\mathcal{V}$rf to derive the exact control flow path that could have led to the received hash digest (for the entire execution). The complexity of this task grows exponentially with the number of control flow transfers, leading to the well-known path explosion problem [57], [58]. Due to path explosion, more recent *CFA* approaches opt to store CF_{Log} verbatim [2], [47], [51], [52] or in compressed form without loss of information [48], [49]. Consequently, the aforementioned costs limit their applicability to small operations [48], [47]. While simple optimizations to reduce CF_{Log} (such as replacing simple loops with counters [1], [47], [53]; or storing decision bits instead of destination addresses for simple conditional branches [48], [49]) have been proposed, they are static. In other words, they are hard-coded on the *CFA* RoT and unaware of the application being attested in a "one size fits all" fashion.

Contrary to static approaches, our work is motivated by two key premises: **(1)** Expected control flow sub-paths are often predictable and repetitive. Therefore, with knowledge of the attested program's expected behavior (e.g., based on prior execution observation), one can learn likely sub-paths with reasonable accuracy; **(2)** Anomalous and malicious CF_{Log}-s are rare. While $\mathcal{V}$rf should still receive **<u>all</u>** control flow information, attacks do not happen as often as benign executions. Therefore, it should be possible to replace expected benign sub-paths with small reserved symbols to indicate that an entire benign/expected sub-path has occurred (instead of filling CF_{Log} with redundant data). Based on these premises, we propose *SpecCFA*: an approach to enable dynamic sub-path speculation policies in *CFA*.

SpecCFA provides $\mathcal{V}$rf with the ability to speculate on expected control flow transfers for the attested operation. If speculations match an execution sub-path taken by $\mathcal{P}$rv at run-time, the entire matching sub-path is replaced by a reserved symbol, greatly reducing CF_{Log} size without loss of information. *SpecCFA* allows simultaneous speculation on multiple variable-sized sub-paths for a given attested operation.

Unsurprisingly, the ability to dynamically speculate on sub-paths also opens attack vectors that (if left unprotected) could be exploited by an adversary ($\mathcal{A}$dv) to forge/spoof execution traces. This leads to non-trivial challenges that must be overcome to realize *SpecCFA* securely. Therefore, we also specify and implement architectural measures to guarantee that *SpecCFA* retains the same security as the underlying *CFA* architectures while achieving performance gains.

We note that existing *CFA* is either based on Trusted Execution Environments (TEEs) or custom hardware support. To demonstrate *SpecCFA*'s generality, we implement it atop one representative of each category, namely: **(1)** ACFA [1] which targets lowest-end MCUs and employs custom hardware support; and **(2)** ISC-FLAT [2] which targets "off-the-shelf" MCUs, leveraging a commodity TEE (TrustZone for ARM Cortex-M). In the former, *SpecCFA* is instantiated as a custom hardware design. In the latter, *SpecCFA* is implemented within the TEE's trusted world. We choose these architectures due to their public availability. Nonetheless, we believe *SpecCFA*'s concepts to be broadly applicable to any *CFA* architecture.

Finally, we also propose and evaluate several approaches to determine effective path speculations. We consider and compare automated approaches based on previously observed CF_{Log}-s and static analysis, in addition to manual analysis of expected control flow paths. In several cases, *SpecCFA* leads to order-of-magnitude improvements in storage, bandwidth, and communication latency while retaining all *CFA* guarantees. We make *SpecCFA* prototype implementation publicly available at [59].

2. Background & Related Work

2.1. Remote Attestation (*RA*)

RA is a challenge-response protocol in which a $\mathcal{V}$rf aims to check the software image currently installed on $\mathcal{P}$rv, i.e., the content of $\mathcal{P}$rv's program memory (PMEM). A typical *RA* protocol is performed as follows:

1) $\mathcal{V}$rf sends $\mathcal{P}$rv a unique cryptographic challenge (*Chal*).
2) After authenticating $\mathcal{V}$rf's request containing *Chal*, an RoT in $\mathcal{P}$rv computes an authenticated integrity-ensuring function over PMEM and *Chal* to produce a response (H).
3) $\mathcal{P}$rv sends H to $\mathcal{V}$rf.
4) $\mathcal{V}$rf compares H to its expected value.

Step (2) can be implemented using a Message Authentication Code (MAC) or digital signature. The secret key used in this operation must be securely stored by the RoT on $\mathcal{P}$rv to ensure it is inaccessible to untrusted software.

RA architectures are usually classified into three categories: software-based, hardware-based, or hybrid, depending on how their RoT is implemented. Software-based *RA* [5], [16], [19], [21], [23], [24], [25], [26] does not rely on specialized hardware but relies on strong assumptions about $\mathcal{A}$dv and the system. Hardware-based approaches [15], [18], [27], [28], [29] rely on support from dedicated hardware, TPMs [60], or instruction set features [61] to achieve attestation with stronger security guarantees at a higher cost. Hybrid *RA* schemes [9], [10], [11], [12], [30] combine hardware and software to achieve security guarantees comparable to hardware-based *RA* while minimizing the hardware cost. Hybrid approaches perform the *RA* measurement in software while protecting its execution and cryptographic key(s) through custom hardware.

2.2. Control Flow Attestation/Auditing (*CFA*)

CFA [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [62] augments *RA* to detect control flow attacks. In addition to proving if the correct software image is installed on $\mathcal{P}$rv, *CFA* produces an authenticated log (CF_{Log}) of all executed control flow transfers. Existing *CFA* methods use either (1) binary instrumentation and TEE support [48],

[51], [52], [53]; or (2) custom hardware modifications [1], [46], [47], [49], [50], [55] to securely create and store CF_{Log}. After execution, $\mathcal{P}$rv's RoT MACs/signs CF_{Log} along with the *RA* evidence to produce H. Upon receiving CF_{Log} and H, $\mathcal{V}$rf inspects their contents to detect control flow attacks.

Early *CFA* techniques used hash-chains to return a single hash [46], [53] (or multiple hashes [50]) of CF_{Log} to $\mathcal{V}$rf. To verify the execution, $\mathcal{V}$rf checks if the received hash digest corresponds to a valid path. Although this approach reduces CF_{Log} to a small fixed size, it is limited in its scalability to more complex software, as $\mathcal{V}$rf might face path explosion when attempting to produce a complete set of valid paths [57], [58]. Because of this, more recent *CFA* designs produce *verbatim* CF_{Log}-s that include the destination address of each control flow transfer. *Verbatim* CF_{Log}-s ease verification, but storage and transmission of CF_{Log} become challenging in branch-intensive attested programs. While one could consider standard compression algorithms (e.g., based on Huffman codes [63]) to reduce CF_{Log}, these algorithms are too heavy to run locally on a resource-constrained $\mathcal{P}$rv. Instead, prior work has introduced simpler strategies to reduce CF_{Log}. LiteHAX [49] is a hardware-based approach that records a reduced-sized bitstream of CF_{Log}. LiteHAX logs a single bit for each direct jump/call and conditional branch (1/0 if the branch was/was not taken, respectively). As indirect branches can have multiple destinations, their full address is recorded in the bitstream. OAT [48] records branch destinations in a similar manner to LiteHAX and reduces the bitstream size further by producing a hash-chain of return addresses instead of adding them to the bitstream. ARI [54] follows the same logging scheme as OAT while only recording the control flow transfers that enter, exit, or occur within user-defined mission-critical components.

An alternative approach to encoding the CF_{Log} as a bitstream is to continuously transmit a series of reduced-sized CF_{Log}-s (slices) to $\mathcal{V}$rf. ScaRR [51] provides *CFA* with CF_{Log} slicing for complex systems such as cloud-based virtual machines. ACFA [1] is a hardware-software co-design for low-end MCUs that uses custom hardware to actively interrupt execution to transmit fixed-sized CF_{Log} slices. ACFA is the first technique to enable control flow *auditing* by incorporating reliable evidence delivery as part of its RoT.

To our knowledge, no prior works consider the specificity of each software being attested to reduce CF_{Log} size. For example, while utilizing loop counters is generally applicable, it offers little optimization to programs with few or complex loops. Similarly, this generality may miss out on unique program behaviors that offer better reductions. Our work bridges this gap by enabling *secure and configurable program-specific speculations* that lead to significant performance improvements in *CFA*.

3. *SpecCFA* at a High-Level

As part of a *CFA* request, *SpecCFA* allows $\mathcal{V}$rf to specify which control flow sub-paths are expected to occur the most during the execution of the attested program. Upon receiving an authenticated request from $\mathcal{V}$rf (along with the specified sub-paths), *SpecCFA* RoT saves the $\mathcal{V}$rf-defined paths to protected memory and starts the attested execution of the requested program. As the program executes, the underlying *CFA* architecture saves control flow transfers to CF_{Log}. Whenever a matching control flow sub-path is found in CF_{Log}, it is replaced by a short reserved symbol, indicating one occurrence of a $\mathcal{V}$rf-specified sub-path in CF_{Log}.

Remark 1: *as $\mathcal{V}$rf is aware of the path/symbol correspondence, SpecCFA does not result in any loss of information in CF_{Log}. $\mathcal{V}$rf can simply replace the symbols in the received CF_{Log} to derive the full CF_{Log} for verification.*

Fig. 1 presents a high-level example of *SpecCFA*'s intended behavior. In this example, CF_{Log} is being monitored by *SpecCFA* to optimize the $\mathcal{V}$rf-specified sub-path $\{A, B, D, G\}$. As the program executes, the underlying *CFA* architecture appends data to CF_{Log} after each control flow transfer. When a matching instance of the sub-path appears in CF_{Log} (stage `(a)`), *SpecCFA* detects and replaces the sub-path with its associated symbol: an ID equal to 1 in this example (stage `(b)`). As execution continues, CF_{Log} continues to grow with new transfers (stage `(c)`) and *SpecCFA* does not modify CF_{Log} until the next match occurs (stage transition `(d)` $\rightarrow$ `(e)`). After the sub-path replacement, execution continues and new transfers are appended normally (stage `(f)`). This process continues until the end of execution. For simplicity, this example shows an optimization of a single sub-path containing only four transfers. However, *SpecCFA* supports speculation of multiple sub-paths of arbitrary length. As noted earlier, path speculations (e.g., $\{A, B, D, G\}$ in this example) are defined by $\mathcal{V}$rf and sent to $\mathcal{P}$rv along with *CFA* requests (recall Sec. 2). Importantly, $\mathcal{V}$rf is not always required to send speculation paths, but only when it decides to update a speculation strategy.

As we demonstrate in Sec. 8, this intuitive idea results in significant performance improvements in terms of storage, bandwidth, and latency. However, from a security standpoint, *SpecCFA*'s design must overcome several non-trivial challenges. In particular, it must ensure that the ability to speculate on sub-paths does not lead to exploitable attack vectors (on a $\mathcal{P}$rv whose software is, by assumption, potentially compromised). For instance, $\mathcal{A}$dv could attempt to compromise CF_{Log} integrity by replacing illegal paths with expected sub-path symbols. The latter can be accomplished by a variety of means, depending on $\mathcal{A}$dv's strategy. Therefore, Sec. 4 and Sec. 5 detail *SpecCFA* design in order to prevent any such attempt, while facilitating *SpecCFA* performance gains. Then, in Sec. 6, we argue the security of the overall constructions for the two classes of *CFA* considered in this work (i.e., based on custom hardware and TEEs).

Remark 2: *While SpecCFA savings do not extend to malicious or anomalous sub-paths (as $\mathcal{V}$rf would not speculate on unknown anomalies/attacks), such instances are rare. In these cases, SpecCFA still ensures accurate detection. Conversely, during $\mathcal{P}$rv's predictable and intended executions,*

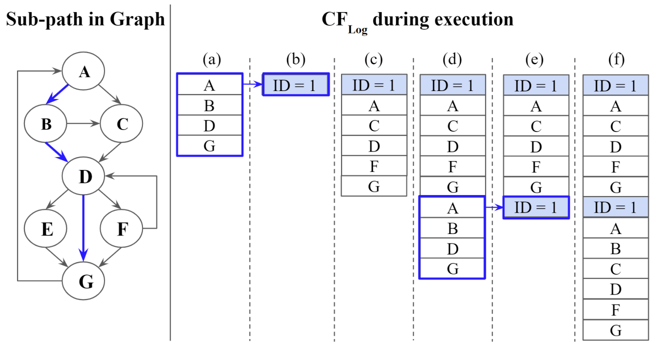


Figure 1: Example optimization made by *SpecCFA*

SpecCFA enhances performance.

Remark 3: *SpecCFA inherits its support for interrupts from the underlying CFA architecture. Some CFA schemes [2] allow but do not record external interrupts to CF_{Log}. As such, interrupts do not affect SpecCFA sub-path speculations. For architectures that record all interrupts in CF_{Log} [1], SpecCFA can speculate on interrupt routine paths if included in the sub-path definition.*

3.1. System Model & Scope

As noted in Sec. 1, this work focuses on resource-constrained edge embedded devices, implemented using MCUs. These MCUs are single-core and run software atop "bare-metal" (executing instructions physically from program memory). They lack MMUs and strong privilege separation to support virtual memory or secure micro-kernels. This scope is aligned with the related work discussed in Sec. 2. As mentioned earlier, our work targets both types of *CFA* architectures considered in the literature: those employing custom hardware designs and those based on TEEs.

Custom hardware approaches assume no reliance on pre-existent hardware features. On the other hand, they employ small hardware modifications to implement *CFA*. In this case, *SpecCFA* features are also implemented in hardware, as part of the *CFA* RoT. This model is equivalent to prior related work, without additional assumptions. We present this version of *SpecCFA* in Sec. 4.

For the TEE-based version of *SpecCFA*, discussed in Sec. 5, our implementation instantiates unmodified ARMv8 MCUs equipped with TrustZone-M. Attested programs execute in the "Non-Secure" World, while the "Secure World" is trusted and used to implement the *CFA* RoT (including *SpecCFA* new features). This $\mathcal{A}$dv model is equivalent to prior work on TEE-based *CFA*, without additional assumptions.

3.2. Adversary ($\mathcal{A}$dv) Model

SpecCFA with custom hardware considers a strong $\mathcal{A}$dv that can exploit software vulnerabilities in $\mathcal{P}$rv to (1) modify any writable memory that is not explicitly protected by hardware-enforced access controls; (2) cause malicious control flow transfers; and (3) attempt to hide their malicious

Table 1: Notation Summary

Symbol	Definition
PC	Program Counter (points to the current instruction).
W_{en}	MCU write enable bit (set when writing to memory)
D_{addr}	the MCU address being read or written from/to.
DMA_{en}	DMA write enable bit (set when DMA writes to memory)
DMA_{addr}	the DMA address being read or written from/to.
BlockMem	Reserved memory for storing sub-path speculations
CF_{Log}	log that stores attested control flow
CF_{Size}	current size of CF_{Log}
$(src, dest)$	source and destination of the current control flow transfer
hw_{en}	set when *CFA* module is appending CF_{Log} with $(src, dest)$
$Block_i$	sequence of transfers defining sub-path$_i$ speculation
$block_{base}$	the base address of a block with respect to *BlockMem*.
$block_{ptr}$	points to ($block_{src}$, $block_{dest}$) being checked.
$block_{ID}$	the ID of a sub-path
$block_{len}$	the length of the sub-path
$(block_{src}, block_{dest})$	the source/destination addresses in a given sub-path entry
$detect_{active_i}$	set by Block$_i$ Detect module when sub-path$_i$ has occurred
$active_{addr_i}$	address of sub-path$_i$ in CF_{Log} when $detect_{active_i}$ is set
$active_{ID}$	the selected $block_{ID_i}$ when $detect_{active_i}$ is set
$detect_{any}$	a signal set when any sub-path has been detected
$spec_{en}$	a signal set when any speculation (or repeat) is detected
$(spec_{addr}, spec_{value})$	the location and value of the optimization and in CF_{Log}

actions (in the form of injected code or hijacked control-flows). Unless prevented, modifications to program memory can change instructions, and modifications to data memory can corrupt intermediate computation results and affect the program's control flow. The TEE-based version of *SpecCFA* considers that $\mathcal{A}$dv can fully compromise the Non-Secure World on $\mathcal{P}$rv. $\mathcal{A}$dv can exploit vulnerabilities to launch code injection attacks, hijack control flow, or perform code reuse attacks. In addition, $\mathcal{A}$dv can manipulate any Non-Secure World configuration registers. In each case, our $\mathcal{A}$dv model remains consistent with existing *CFA* architectures. In both cases, hardware attacks that require physical access/modification to circumvent $\mathcal{P}$rv hardware protections (or hardware-protected software) are out-of-scope. Protection against physical hardware attacks involves orthogonal physical access control measures [64].

4. *SpecCFA* with Custom Hardware

Fig. 2 depicts a *CFA*-enabled MCU architecture extended with *SpecCFA*'s custom hardware design. Table 1 summarizes the notation used in the rest of the paper.

The underlying *CFA* architecture monitors several CPU signals at runtime. By checking the program counter (PC) as well as the *opcode* of the currently executing instruction, it obtains control flow transfers' source and destination $(src, dest)$ addresses to append them to CF_{Log} as they occur. When writing the $(src, dest)$ pair to CF_{Log}, it sets a flag (denoted hw_{en}). *SpecCFA* interacts with the *CFA* architecture by monitoring hw_{en} and $(src, dest)$ being written to CF_{Log}. In addition to these signals from the *CFA* architecture, *SpecCFA* hardware also monitors certain signals from the MCU to provide additional properties. PC is used to determine which part of the software is executing. Signals regarding memory accesses, such as the write- and read-enable bits (W_{en}, R_{en}, respectively) indicate that the MCU is currently writing to memory ($W_{en} = 1$) or reading from memory ($R_{en} = 1$). Whenever a read/write occurs,

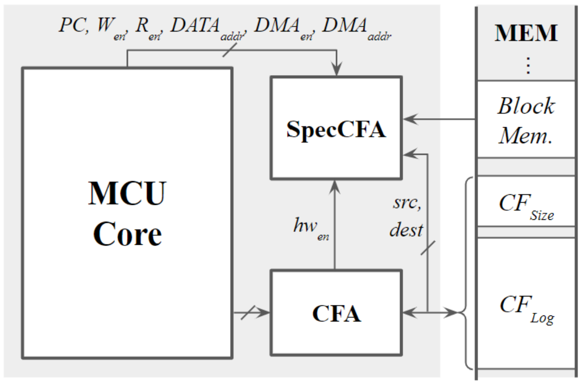


Figure 2: System overview of *SpecCFA* hardware

the D_{addr} signal contains the address of the read/written memory address. Similarly, *SpecCFA* monitors Direct Memory Access (DMA) related signals (DMA_{addr}, DMA_{en}) to verify memory accesses performed by DMA. Whenever DMA accesses (either reads or writes) memory, DMA_{addr} contains the address being accessed and $DMA_{en} = 1$.

Both *SpecCFA*-specific modules and the underlying *CFA* architecture can modify CF_{Log}'s designated memory. In addition, both also monitor the current size of CF_{Log} (denoted $\mathcal{C}F_{Size}$), which is incremented by the *CFA* architecture whenever it appends a new entry to CF_{Log} to track the next available memory address. Finally, *SpecCFA* interfaces with and monitors *BlockMem*, which contains memory blocks dedicated to storing the sub-path speculation definition(s), as received from $\mathcal{V}$rf in the *CFA* requests. *BlockMem*, CF_{Log}, and $\mathcal{C}F_{Size}$ are stored in hardware-protected and dedicated addresses in memory that are inaccessible to untrusted software in $\mathcal{P}$rv and can only be written by the *CFA* and *SpecCFA* modules.

4.1. Memory Organization & Sub-Modules

In its hardware-based version, *SpecCFA* is composed of four internal sub-modules, namely: Memory Interface, Block Detect module(s), Repeat Detect, and Memory Monitor. Fig. 3 further details their interconnections.

Memory Interface performs accesses to CF_{Log} and $\mathcal{C}F_{Size}$ (in memory) to replace matching sub-paths, when detected. *SpecCFA* instantiates one Block Detect module per speculated sub-path (referred to as a "block" of control flow transfers) in *BlockMem*, and each instance is responsible for detecting the occurrence of their associated sub-path in CF_{Log}. Each Block Detect module interfaces with *BlockMem* directly to read the ID, len, and sub-path (src, $dest$) pairs of a given block. For the remainder of this section, we refer to the data read from *BlockMem* by each Block Detect module as ($block_{ID}$, $block_{len}$, $block_{src}$, $block_{dest}$).

Every Block Detect module compares each current (src, $dest$) being written to CF_{Log} to their ($block_{src}$, $block_{dest}$) pairs in *BlockMem* to determine if their sub-path has occurred. At each match, they iterate through *BlockMem* to determine the next expected ($block_{src}$, $block_{dest}$) pair to be checked. When a full sub-path is detected, they output the following signals:

1) $detect_{active}$: a flag indicating that a sub-path has been detected.
2) $active_{addr}$: the memory address of the sub-path in CF_{Log} as an offset to the base address of CF_{Log}.
3) $block_{ID}$: the ID associated with the detected sub-path.

All Block Detect modules' outputs are selected by a multiplexer (MUX), using $detect_{active}$ bits as a selector.

The Repeat Detect module monitors detected sub-paths for repeated adjacent speculations. The module replaces repeated speculations with a counter of how many times the sub-path occurred successively to further reduce CF_{Log} size (instead of logging the same symbol multiple times). The module receives a logic OR of all $detect_{active}$ signals, indicating that one of the Block Detect modules detected a sub-path. In addition, it receives $active_{ID}$ and $active_{addr}$ as outputs from the MUX, which contain the $block_{ID}$ and memory address in CF_{Log} of the detected sub-path. Repeat Detect then compares the detected sub-path with the previous speculation and sets three output signals: $spec_{value}$, $spec_{addr}$, and $spec_{en}$. If a repeat is detected, the module increments an internal repeat counter, sets $spec_{value}$ to the value of this counter, and $spec_{addr}$ to the address of the counter in CF_{Log}. Otherwise, $spec_{value}$ and $spec_{addr}$ are set to $active_{ID}$ and $active_{addr}$, respectively. In either case, $spec_{en}$ is set to indicate a speculation match was found.

The Memory Interface module receives $spec_{en}$ along with $spec_{value}$ and $spec_{addr}$ which contain the current match and address of the match in CF_{Log}. When $spec_{en}$ is set, The Memory Interface then writes $spec_{value}$ to CF_{Log} at $spec_{addr}$ and decrements $\mathcal{C}F_{Size}$ according to the size reduction due to the matching sub-path replacement.

Finally, the Memory Monitor ensures that no untrusted software on $\mathcal{P}$rv can edit *BlockMem*. It triggers a hardware exception whenever CPU-writes or DMA-writes to *BlockMem* are attempted unless they originate from within the *CFA* RoT. To detect writes within the bounds of *BlockMem* and determine whether they originate from the *CFA*'s RoT, Memory Monitor checks the signals: PC, W_{en}, D_{addr}, DMA_{en}, and DMA_{addr} (see Sec. 4.3 for details).

4.2. Sub-Path Detection Finite State Machine

The sub-path detection logic is defined in the finite state machine (FSM) depicted in Fig. 4 with three states: $Idle$, $Monitor$, and $Detect$. Each Block Detect module is implemented as one instance of this FSM and is initialized in the $Idle$ state. All FSM transitions occur based on comparing the current entry in CF_{Log} to $(block_{src}, block_{dest})$. Each Block Detect module maintains a pointer, $block_{ptr}$, which points to one $(block_{src}, block_{dest})$ pair in *BlockMem*. For example, for a Block Detect module to monitor $(block_{src_i}, block_{dest_i})$, it sets $block_{ptr} = i$. Given the current $block_{ptr}$, a module can compare the current CF_{Log} entry to the appropriate sub-path transfer to determine if the current CF_{Log} entry:

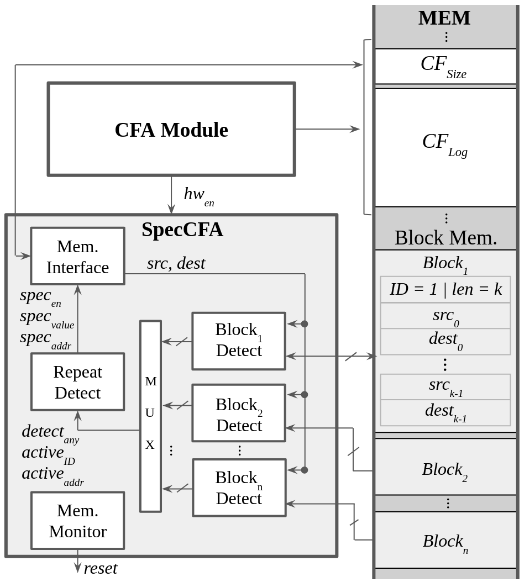


Figure 3: *SpecCFA* hardware components

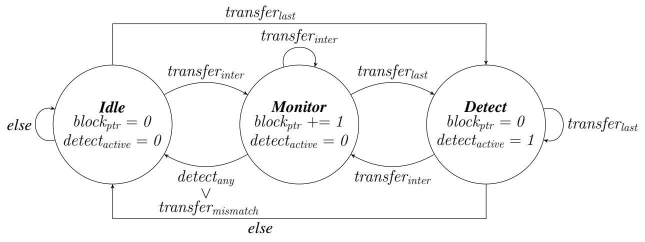


Figure 4: State Machine of the Block Detection module(s)

- matches the first or an intermediate $(block_{src}, block_{dest})$ in the sub-path (denoted $transfer_{inter}$ in Fig. 4);
- matches the last $(block_{src}, block_{dest})$ in the sub-path (denoted $transfer_{last}$ in Fig. 4); or
- is a mismatch (denoted $transfer_{mismatch}$ in Fig. 4).

In addition, each Block Detect module receives the signal $detect_{any}$ from the outer *SpecCFA* module, which is set when another Block Detect module has detected its sub-path.

When the FSM is in the $Idle$ state, $block_{ptr} = 0$ and $(block_{src},\ block_{dest})$ contains the first transfer of the sub-path. The FSM transitions when the first $(src, dest)$ occurs. If $(src, dest)$ matches $(block_{src},\ block_{dest})$ and the $\mathcal{V}$rf-specified sub-path contains just one control flow transfer, a transition to $Detect$ will occur because the whole path (one transfer) has been detected. Otherwise, a transition to $Monitor$ occurs. If $(src, dest)$ does not match $(block_{src}, block_{dest})$, the FSM stays in $Idle$.

Upon entering $Monitor$, *SpecCFA* has identified that the current CF_{Log} entry matches the first transfer of the sub-path. To confirm a sub-path has occurred in CF_{Log}, the Block Detect module must identify the exact remaining sequence of transfers in the sub-path. Therefore, while in $Monitor$, the Block Detect module increments $block_{ptr}$ for each match, to compare the next CF_{Log} entry to the next transfer in the sub-path. For as long as $(block_{src_i}, block_{dest_i})$ matches CF_{Log} entries, the FSM stays in the $Monitor$ state. If a mismatch occurs or another sub-path is detected ($detect_{any} = 1$), the FSM transitions back to $Idle$ and $block_{ptr}$ is reset to 0. If the last transfer in the sub-path is reached, indicating a complete match, the FSM transitions to $Detect$.

HW Specification: Monitor Boundaries of *BlockMem*

$$(\neg(PC \in TCB) \wedge (W_{en} \wedge (D_{addr} \in \textit{BlockMem})))$$
$$\vee\ (DMA_{en} \wedge (DMA_{addr} \in \textit{BlockMem})) \rightarrow reset$$

Figure 5: Memory Monitor specification

HW Specification: Base address of Block$_i$ in *BlockMem*:

$$block_{base_i} = \begin{cases} 0, & \text{if } i = 0 \\ block_{base_{(i-1)}} + (2 \times block_{len_{(i-1)}}) + 1 & \text{otherwise} \end{cases}$$

HW Specification: Given $block_{base_i}$ and $block_{ptr}$, get block signals

$$block_{ID} := BlockMem[block_{base_i}][15:8]$$
$$block_{len} := BlockMem[block_{base_i}][7:0]$$
$$block_{src} := BlockMem[block_{base_i} + block_{ptr} + 1]$$
$$block_{dest} := BlockMem[block_{base_i} + block_{ptr} + 2]$$

Figure 6: Block$_i$ Detect Module signals from *BlockMem*

In $Detect$, *SpecCFA* hardware optimizes CF_{Log} by replacing the sub-path with the $block_{ID}$. $detect_{active}$ is set to 1, triggering the Memory Interface module, which optimizes CF_{Log} according to the $spec_{addr}$ and $spec_{value}$ values for the detected sub-path. After the replacement, $\mathcal{C}F_{Size}$ is updated to reflect the new reduced CF_{Log} size. In addition, the $block_{ptr}$ is reset to 0. If the next transition is the first transfer of the sub-path, the FSM transitions to $Monitor$ or $Detect$ depending on the size of the sub-path. Otherwise, the FSM transitions back to the $Idle$ state.

4.3. Hardware Specification Details

Memory Monitor: Per Fig. 5, this module monitors MCU signals to identify attempts to tamper with *BlockMem* during the attested execution. To detect when the MCU is not executing the *CFA* RoT, the Memory Monitor compares PC to the bounds of the memory region storing the *CFA* RoT code, denoted Trusted Computing Base (TCB). If W_{en} is set and D_{addr} is within the bounds of *BlockMem*, a CPU write to *BlockMem* is occurring. If PC is outside TCB, *SpecCFA* interprets this action as an attempt by $\mathcal{A}$dv to tamper with *BlockMem*; thus, it triggers a hardware exception. Similarly, if DMA_{en} is set and DMA_{addr} is within the bounds of *BlockMem*, DMA is modifying *BlockMem* (note that *BlockMem* is a fixed reserved memory region). This triggers an exception at any time during execution. Similar to other hardware exceptions on the target MCU (TI MSP430), this exception causes a system-wide reset. Note that similar controls from the underlying *CFA* architecture [1] prevent unauthorized modification of CF_{Log}.

Block Detect Module(s): Each instance of this module abides by the state machine in Fig. 4. The hardware specifications for interfacing with *BlockMem* are shown in Fig. 6.

Defintions: Intermediate, Last, and Mismatch Sub-Path Transfers

$$\begin{aligned} transfer_{mismatch} &:= hw_{en} \wedge ((src \neq block_{src}) \vee (dest \neq block_{dest})) \\ transfer_{inter} &:= hw_{en} \wedge (src = block_{src}) \wedge (dest = block_{dest}) \\ &\quad \wedge (block_{ptr} < block_{len} - 1) \\ transfer_{last} &:= hw_{en} \wedge (src = block_{src}) \wedge (dest = block_{dest}) \\ &\quad \wedge (block_{ptr} = block_{len} - 1) \end{aligned}$$

Sub-Path Tracking:

$$block_{ptr} = \begin{cases} block_{ptr} + 1, & \text{if } transfer_{inter} \\ 0, & \text{else if } transfer_{mismatch} \\ block_{ptr} & \text{otherwise} \end{cases}$$

Active Sub-Path Detection:

$$transfer_{last} \rightarrow detect_{active}$$

$$active_{addr} = \begin{cases} (\mathcal{C}F_{Size} - 2) - (2 \times block_{len}), & \text{if } transfer_{last} \\ & \quad \wedge (repeat_{ctr} = 2) \\ active_{addr} & \text{otherwise} \end{cases}$$

Figure 7: Block Detection specification

Each Block Detect module must determine the sub-path block's base address ($block_{base}$) relative to the entire *BlockMem* region to read the correct signals from *BlockMem*. This address is zero for the first sub-path block. The locations of all subsequent sub-path blocks depend on the length of the previous block(s), which are variable depending on $\mathcal{V}$rf's sub-path specification. Therefore, for the remaining blocks, the $block_{base}$ is calculated by incrementing the previous block's base address by its size. Each block contains a $block_{ID}$, $block_{len}$, and a series of control flow transfers. Since $block_{len}$ refers to the number of 32-bit control flow transfers in the sub-path, the size of the transfers in memory is $2 \times block_{len}$ given 16-bit addresses in the target MCU. Both $block_{ID}$ and $block_{len}$ are stored within a single address. As such $block_{base_i} = block_{base_{(i-1)}} + (2 \times block_{len_{(i-1)}}) + 1$.

With $block_{base_i}$ and $block_{ptr}$, the remaining entries can be read from *BlockMem*. The first signal returned is $block_{ID}$, which references an 8-bit sub-path ID. The second signal is $block_{len}$, which references the 8-bit length of the sub-path. Since MSP430 has 16-bit addresses, the upper bits of one address store ID, while the lower bits store len. Next, $block_{src}$ and $block_{dest}$ signals are set based on both the $block_{base_i}$ and $block_{ptr}$ values. Since the first address stored in a block is ($ID \mid len$) (as shown in Fig. 3), the first $(src, dest)$ are stored starting in an offset by 1 and 2 addresses, respectively. Therefore, to retrieve the expected $block_{src}$, this module reads from *BlockMem* at the location defined by $block_{base_i} + block_{ptr} + 1$. Similarly, the location defined by $block_{base_i} + block_{ptr} + 2$ is referenced to return the expected $block_{dest}$.

Fig. 7 shows the hardware specifications for different types of transfers in a sub-path. With the signals from *BlockMem*, each Block Detect module compares the current $(src, dest)$ pair to ($block_{src}$, $block_{dest}$) If the current $(src, dest)$ does not match the current sub-path pair $(block_{src}, block_{dest})$, then the current control flow transfer is identified as a mismatch ($transfer_{mismatch}$ bit is set). If the two pairs match, the module must determine if this is the last transfer in the sub-path or an intermediate transfer. The type of match is determined by comparing $block_{ptr}$ and $block_{len}$. $transfer_{inter}$ is set when an intermediate sub-path transfer is matched. $transfer_{last}$ is set when the last sub-path transfer match is reached. In accordance with the aforementioned logic, $block_{ptr}$ is incremented whenever an intermediate sub-path transfer match occurs and reset whenever a mismatch is reached.

The internal signal $transfer_{last}$ will only be set when the last sub-path transfer has occurred, implying that all prior transfers were matches. At that stage, the output signal $detect_{active}$ is set. The address of the active sub-path ($active_{addr}$) is also updated with the start location of the sub-path in CF_{Log}. This address is determined by subtracting $2 \times block_{len}$ from the location of the last $(src, dest)$ pair in CF_{Log} ($\mathcal{C}F_{Size} - 2$).

Memory Interface: This module overwrites CF_{Log} based on $spec_{value}$ and $spec_{addr}$. As described in Sec. 4.1, $spec_{value}$ contains the ID of the detected sub-path, and $spec_{addr}$ holds the memory address in CF_{Log} where the sub-path starts. When the $spec_{en}$ signal is set, $spec_{value}$ is written to CF_{Log} at the memory address offset specified by $spec_{addr}$. Performing this write to CF_{Log} makes $spec_{value}$ the last entry in CF_{Log}. Therefore, $\mathcal{C}F_{Size}$ is updated to $spec_{addr} + 2$. Finally, in the specific case of multiple consecutive matches of the same sub-path, the Repeat Detect module is responsible for replacing repeated ID entries for a count of the number of repetitions (see Appendix A for details).

5. TEE-based *SpecCFA*

ARMv8 Cortex-M MCUs are equipped with TrustZone (TrustZone-M) [65]: an architectural security extension to create an isolated execution environment. Through this extension, hardware, software, and data on the MCU are divided into two worlds: "Secure" and "Non-Secure" Worlds. TrustZone ensures that Secure World code and data cannot be tampered with by code residing in the Non-Secure World (e.g., vulnerable/compromised MCU application code). Thus, security-critical data and code can be safely stored in the Secure World. Furthermore, the Non-Secure World can only call Secure World functions from secure entry points specified as Non-Secure-Callables (NSC), enabling controlled invocation of Secure World code from the Non-Secure World.

CFA techniques that leverage TrustZone-M [2], [48], [53] or other TEE support [51], [52] leverage the Secure World to implement the *CFA* RoT while untrusted application resides in the Non-Secure World. To record control flow transfers, the application is instrumented before deployment, and a call to the Secure World (via an NSC) is placed before each branching instruction. This NSC invokes a Secure World function for logging the branch destination to a Secure World CF_{Log}. This design prevents unauthorized writes to CF_{Log}.

Fig. 8 presents the workflow of *SpecCFA* in a TrustZone-based *CFA* architecture. The *CFA* RoT and *SpecCFA*'s functionality are implemented within the Secure World. Each

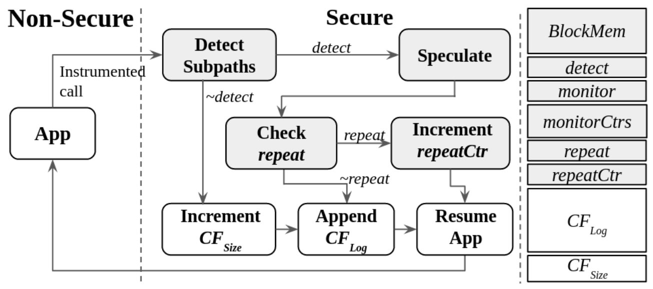


Figure 8: *SpecCFA* workflow on TrustZone-M

NSC switches to the Secure World to append CF_{Log} with the new transfer. TrustZone makes the instrumented code unmodifiable between the point when it is measured (i.e., MAC-ed/signed by the *CFA* RoT) and the point when the attested execution completes [2], guaranteeing to $\mathcal{V}$rf that all branches in the attested code are appropriately instrumented.

Without *SpecCFA*, an NSC would invoke the *CFA* RoT to append CF_{Log} and resume the Non-Secure World execution. *SpecCFA* modifies this behavior to detect matching sub-paths. Similar to the custom hardware version (Sec. 4), *BlockMem* contains the definitions of $\mathcal{V}$rf's expected sub-paths, associated IDs, and sizes. For n sub-paths, it maintains two n-bit values where each index corresponds to the *detect* and *monitor* signals of each monitored sub-path. Each bit signals when a particular sub-path is currently detected or monitored, respectively. For all sub-paths, a counter is stored in *monitorCtrs*, holding the current number of consecutive transfers from that sub-path that have occurred in CF_{Log}. For the most recently detected sub-path, a flag (*repeat*) is maintained to signal when a sub-path repeats in adjacent CF_{Log} locations, and a counter (*repeatCtr*) stores the number of times it has repeated. All data is stored in the Secure World and thus is made inaccessible to a compromised application.

When an NSC triggers the *CFA* RoT in *SpecCFA*, it first uses the reported destination address to monitor and detect sub-paths. During this phase, the destination address is compared to the next address of each sub-path, determined from *monitorCtrs* for each sub-path. When a match occurs for the next entry in any sub-path, the corresponding bit in *monitor* is set, and its counter in *monitorCtrs* is incremented. A sub-path match is detected if the new count equals the sub-path size, and the corresponding bit in *detect* is set. Otherwise, *SpecCFA* executes normally to append CF_{Log}.

Once a sub-path is detected, *SpecCFA* modifies CF_{Size} to reduce CF_{Log} and writes the sub-path *ID* to CF_{Log}. Then, it compares the current sub-path *ID* to the last logged value. If they are equal, *SpecCFA* increments *repeatCtr* and returns to the application. When a sub-path is no longer repeating, its *repeatCtr* is written to CF_{Log} (conveying the number of consecutive repetitions of a sub-path).

6. Security Analysis

We argue that *SpecCFA* added capability does not affect the security of the underlying *CFA* architectures in producing a correct and authentic CF_{Log}. To see why, note that $\mathcal{A}$dv (in the form of compromised software in $\mathcal{P}$rv, as defined in Sec. 3.2) may attempt to leverage *SpecCFA* to launch attacks against *CFA* in the following ways:

1) $\mathcal{A}$dv could attempt to modify CF_{Log} directly. If successful, this could cause CF_{Log} to be reset and/or overwritten with entries that do not faithfully correspond to the executed control flow path.
2) $\mathcal{A}$dv could attempt to corrupt the content of *BlockMem*, modifying the $\mathcal{V}$rf-specified sub-path speculation to include the malicious transfers. A corrupted *BlockMem* would cause *SpecCFA* to optimize away the malicious path from CF_{Log}, hiding the attack.
3) $\mathcal{A}$dv could attempt to impersonate $\mathcal{V}$rf over the network to provide $\mathcal{P}$rv with false sub-path speculations, hence overwriting *BlockMem* to hide malicious sub-paths with symbols that would normally denote benign sub-paths.

Case 3 is prevented by ensuring that the *CFA* RoT always authenticates $\mathcal{V}$rf requests (and sub-path speculations therein) before overwriting *BlockMem*. Hence, the remainder of this analysis focuses on cases 1 and 2 for each type of architecture.

***SpecCFA* on Custom Hardware Architectures:** In the hardware-based design of *SpecCFA*, CF_{Log} is protected against direct software writes by the underlying *CFA*. Therefore, $\mathcal{A}$dv must at run-time force an incorrect value of $spec_{addr}$, $spec_{value}$, or $spec_{en}$ to corrupt CF_{Log}. Since *SpecCFA* hardware controls these signals, they cannot be tampered with by any MCU software. Thus, they are unaffected by the untrusted software on $\mathcal{P}$rv and $\mathcal{A}$dv cannot corrupt CF_{Log}. Unlike *SpecCFA* signals, the contents of *BlockMem* reside in the address space of the MCU. However, it is impossible for $\mathcal{A}$dv to corrupt *BlockMem* since the Memory Monitor in *SpecCFA* prevents untrusted software modifications (by CPU or DMA) to *BlockMem*.

***SpecCFA* on TEE-based Architectures:** The TEE-based version of *SpecCFA* stores CF_{Log} in the Secure World. Therefore, CF_{Log} is inaccessible to $\mathcal{A}$dv in the Non-Secure World. Since the code of the attested program is unmodifiable during the attested execution and the application binary is instrumented (see Sec. 5), modifications to CF_{Log} can only occur through the instrumented NSC calls that follow a control flow transfer. Thus, CF_{Log} is only appended with proper control flow transfer destinations, and $\mathcal{A}$dv cannot reset/overwrite entries. Similarly, $\mathcal{A}$dv cannot modify *BlockMem* or *SpecCFA*'s implementation (code) because they are also stored in the Secure World. *BlockMem* specifications (i.e., sub-path definitions) are received from $\mathcal{V}$rf as part of the *CFA* request and authenticated by the *CFA* RoT on $\mathcal{P}$rv before the attested execution. Therefore, they are unmodifiable to the Non-Secure World.

7. Selecting Sub-Path Speculations

Fundamentally, *SpecCFA* performance depends on the effective selection of speculation sub-paths based on program and execution characteristics. As such, we examine multiple possible speculation strategies.

7.1. Program Analysis

Static analysis of the attested code can be used for sub-path speculation selection. This approach is valuable when $\mathcal{V}$rf lacks access to previous CF_{Log}-s and must rely solely on program source code. Our static analysis-based approach examines both the `C` source code and the compiled program binary. Metadata from each function is collected, including the number of branching instructions, the number of loops, how many other functions call it, and how many times it calls other functions.

In our approach, we first extract the program's CFG. Once built, our implementation splits the CFG into "Segments," inspired by prior work in [50]. Segments are determined by splitting the CFG into subgraphs at either the first node of the graph, the last node of the graph, the first node in a loop, or the last node in a loop. Splitting the CFG in this way ensures all Segments are forward-edge sub-graphs. Our implementation also splits the graph at calls and returns in order to avoid path explosion due to indirect calls and returns. Once the Segments are determined, the sub-paths within each Segment are enumerated and collected. After determining the Segment sub-paths for all functions, the set of Segments is optimized by combining those with just one successor Segment. The resulting set of candidate sub-paths is then sorted based on the following priorities:

1) sub-paths that exist within loops;
2) sub-paths in the max-branching function;
3) sub-paths in a function that is called within a loop or within the max-branching function in the code.

Sub-paths in functions that are never called or do not have any internal branches are not considered for the initial set of candidate sub-paths.

Based on the automatically generated candidate sub-paths, smaller sub-paths are first selected to optimize the utilization of *BlockMem* in the initial set. After that, since the exact path that will occur might be highly unpredictable, non-overlapping paths are next selected to increase the initial coverage of the program. The sub-path selection can subsequently refined based on received CF_{Log}-s.

7.2. Automated CF_{Log} Analysis

We also implement automated sub-path selection strategies that utilize past CF_{Log}-s. Three policies were created to examine CF_{Log}-s and recommend the best sub-paths: "Top", "Minimize", and "Select".

Top selects the most occurring non-overlapping sub-paths in the prior CF_{Log}-s. However, *Top* ignores sub-path sizes and can incur large memory overhead due to the increased size of *BlockMem* to store sub-path speculations on $\mathcal{P}$rv.

Table 2: Characteristics of Evaluated Applications

App	Binary Size in Bytes	CF_{Log} data (Bytes)
Ultrasonic [66]	366	4160
Syringe [3]	518	54600
Temperature [67]	564	2508
Geiger [68]	772	1740
Mouse [69]	1119	50116
GPS [70]	6474	19876

Minimize attempts to maximize CF_{Log} reductions while minimizing *BlockMem* sizes. It prioritizes small sub-path sizes first, then sub-path frequency. *Minimize* chooses the N most occurring smallest paths in CF_{Log}, then iterates through all remaining sub-paths. Each remaining path is compared to the least-occurring selected path and replaces it if the candidate path occurs $t\%$ more frequently than the selected path, where t is a configurable threshold.

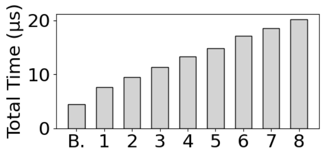
Select picks the most frequent sub-path that fits within the remaining memory in *BlockMem*. It operates on the insight that, despite "Minimize" reducing *BlockMem* usage, the fixed-size *BlockMem* remains part of *SpecCFA*'s memory overhead. Therefore, optimizing memory usage may involve filling the allocated space rather than minimizing *BlockMem* use.

7.3. Manual Inspection

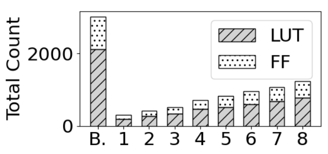
Another possible approach is to manually analyze previously received CF_{Log}-s and the program's binary. Developers may create custom speculations based on their insights into the attested program's behavior, expected inputs, or specific memory constraints. While effective, this approach demands substantial human effort, limiting its scalability. Moreover, as more CF_{Log}-s are received, finding optimal speculation paths may become challenging.

8. Prototypes and Evaluation

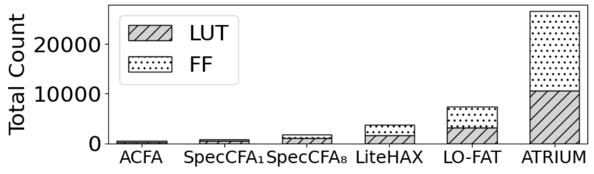
For *SpecCFA*'s version based on custom hardware, we use Xilinx Vivado tool-set [71] to synthesize *SpecCFA* atop the ACFA architecture [1], which targets the openMSP430 core [72]. *SpecCFA* functionality was tested using the Vivado simulator to ensure its correctness. We then synthesized and deployed *SpecCFA* on the Basys3 prototyping board, which features an Artix-7 FPGA. We implement the TEE-based version of *SpecCFA* using a NUCLEO-L552ZE-Q development board equipped with an STM32L552ZE MCU which supports ARM TrustZone-M and is based on the ARM Cortex-M33 (v8) operating at 110 MHz. We integrate *SpecCFA* with the ISC-FLAT [2] open-source TEE-based *CFA* architecture. Both implementations use UART-to-USB as a communication interface with a baud rate of 38400. To evaluate *SpecCFA* on real MCU software, we port the open-source applications to both MSP430 and ARM Cortex-M. Table 2 shows the evaluated applications and their characteristics.



(a) Added NSC Time of *SpecCFA*-TZ



(b) Added cost of *SpecCFA*-HW



(c) *SpecCFA*-HW vs. Related HW-CFA

Figure 9: *SpecCFA* cost analysis: (a) added NSC time (in μs) for 1-8 sub-paths atop baseline CFA logging; (b) additional HW cost of 1-8 sub-paths atop baseline of openMSP430+ACFA; (c) Comparison to related HW-based *CFA*.

8.1. *SpecCFA* HW/Run-time Overheads

The TEE-based version of *SpecCFA* does not modify hardware (thus imposing no hardware overhead). However, it requires additional processing time for each CF_{Log} entry, as matches are processed by the Secure World TCB (implemented in software). Thus, we assess the average time spent by NSC calls to the Secure World. Each NSC call must check the new entry against all active speculation sub-paths (until the first sub-path match – or no match – is found) to appropriately append to/optimize CF_{Log}. Fig. 9(a) shows this cost as a function of the number of sub-paths checked for matches. On average, baseline *CFA* without any speculation (B.) requires $4.4\mu s$ to append an entry to CF_{Log}. With 1 sub-path checked, the average time increases to $7.6\mu s$. After that, each additional sub-path check adds $1.8\mu s$, with $20.8\mu s$ to check 8 independent sub-paths per NSC call.

For the version of *SpecCFA* based on custom hardware, similar to the related work [1], [46], [49], [50], we measure the hardware overhead in terms of added Look-Up Tables (LUTs) and Flip-Flops (FFs). The increase in LUTs estimates the additional chip cost and size due to combinatorial logic, while the added FFs estimate the state overhead for sequential logic. We vary the number of supported speculation sub-paths from 1 to 8. The results are presented in Fig. 9(b). As a conservative baseline, we show the cost of the openMSP430 core equipped with the underlying *CFA* architecture [1], excluding any hardware peripherals (e.g., general purpose I/O, communication interfaces, timers, etc.) that would normally add to the baseline hardware cost (B.) of the MCU. To support one sub-path speculation, *SpecCFA* adds 190 LUTs and 107 FFs. Each additional sub-path incurs an overhead of approximately 85 LUTs and 49 FFs. The custom hardware-based design of *SpecCFA* incurs no runtime overhead to speculate on sub-paths because its modules operate in parallel to the MCU core.

To put *SpecCFA*'s hardware overhead into context, we compare its cost with related *CFA* architectures (that do

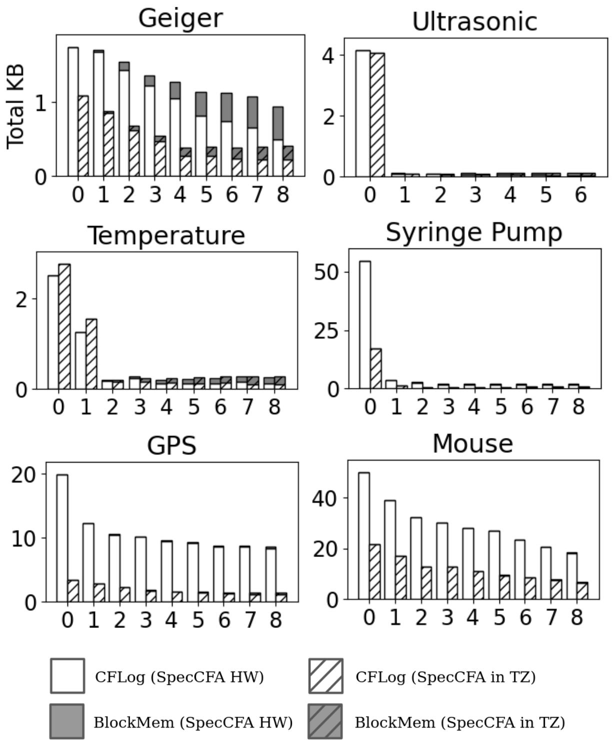


Figure 10: CF_{Log} and *BlockMem* sizes (KB) for 1-8 simultaneous sub-path speculations.

not support configurable path speculations) in Fig. 9(c). As *SpecCFA* implementation is built atop ACFA [1], its relative cost increases accordingly. However, compared to other prior work in hardware-based *CFA* (LiteHAX [49], LO-FAT [46], and ATRIUM [50]), *SpecCFA* incurs relatively low overhead. Even with support 8 sub-paths ("SpecCFA$_8$" in Fig. 9(c)), *SpecCFA*'s cost remains low in comparison. This indicates the feasibility of also deploying *SpecCFA* on top of existing hardware-based *CFA* at a relatively low overhead. Note that we could not implement *SpecCFA* directly on top of these hardware-based *CFA* architectures because they are not open-source. However, we see no reason why *SpecCFA* design would not apply to them.

8.2. Storage & Communication Savings

To evaluate *SpecCFA*'s effectiveness, we start by selecting and configuring sub-paths based on manual inspection of the program source code and previously generated CF_{Log}-s. In Sec. 8.3, we revisit the automated methods discussed in Sec. 7. We measure the overall reduction of CF_{Log} for 1-8 sub-path speculations. We also contrast the reduced CF_{Log} size with *BlockMem* size (required to store the $\mathcal{V}$rf-defined speculation paths). This allows us to observe the total memory required to store both the optimized CF_{Log} and respective sub-path specifications. Fig. 10 shows the memory requirement (in Bytes) to store CF_{Log} and *BlockMem* (stacked in each bar) for hardware-based and TEE-based *SpecCFA* for 1-to-8 sub-paths.

In both designs, the total memory overhead is reduced as *SpecCFA* speculates on more sub-paths. Since the optimizations are application-aware, the savings vary across each application. For example, the first configured sub-path (selected as the most repetitive sequence of transfers) for Ultrasonic Sensor and Syringe Pump alone leads to significant savings. For these applications, optimizing based on 1 sub-path reduces the CF_{Log} by 97.9-97.5% and 93.3-92.2%, respectively. This reduction is due to these applications executing many repeated control flow paths (e.g., repetitive iterations of signal processing functions and busy-wait loops). For the same reason, the Temperature Sensor gains most of its savings (94.4-93.1% CF_{Log} reduction) after speculating on two paths. The GPS, Geiger Counter, and Mouse operations save at a steady rate as more sub-paths are configured. This increase in savings occurs because different sub-paths occur at similar rates.

Due to differences in the two underlying instruction sets (MSP430 vs. ARMv8 Cortex-M), the exact size of CF_{Log} varies for each application in each architecture. CF_{Log}-s for the TEE-based approach are on average smaller due to a more efficient instruction set and because instrumentation can allow the underlying CFA to ignore static branches. The hardware-based approach is implemented alongside MSP430, which has a reduced and less efficient instruction set. The hardware detects control flow transfers through the opcodes. Therefore, it records additional transfers for both direct and conditional jumps since they share the same MSP430 instruction opcode. We refer the reader to Appendix B for additional architectural differences and implementation details.

Naturally, the memory required to store *BlockMem* increases as more sub-path speculations are used. However, the additional memory to store the sub-paths specifications is minimal compared to the savings made when speculating on them.

The memory savings shown in Fig. 10 are also crucial for overall attested operation execution latency. Due to the limited amount of memory on $\mathcal{P}$rv, execution may need to be interrupted to transmit a partial snapshot of CF_{Log} to $\mathcal{V}$rf and free storage for additional transfers. This imposes significant delays to the attested execution, which can be avoided with *SpecCFA*. We revisit this point in Sec. 8.4.

8.3. Path Selection Policy Comparison

Fig. 11 shows the resulting memory overhead for each selection method as a sum of *BlockMem* to store the suggested sub-paths and the optimized CF_{Log} after speculation. Manual Inspection by a developer, in most cases, gains large CF_{Log} optimizations due to prior knowledge of expected input/program behavior and recognizable patterns. However, this approach is not scalable and requires potentially impractical human efforts. Of the four automated processes, the static analyzer is the least performant. Since the static analysis does not consider any prior execution context, it is inherently limited in what it can learn about the program. However, it remains a suitable choice as a starting point when CF_{Log}-s from prior executions do not yet exist or are unavailable. It performs best on programs that heavily execute repeated tasks, such as the Ultrasonic and Temperature Sensors.

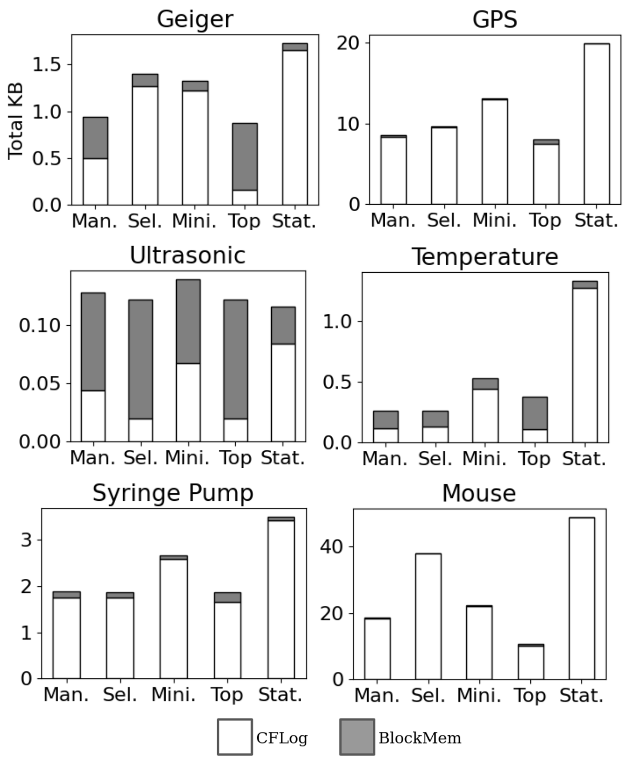
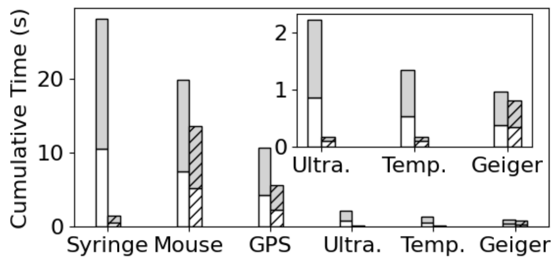


Figure 11: CF_{Log} and *BlockMem* total size (KB) for each sub-path selection method: Manual Inspection (*Man.*), Select (*Sel.*), Minimize (*Mini.*), *Top*, and Static Analysis (*Stat.*)

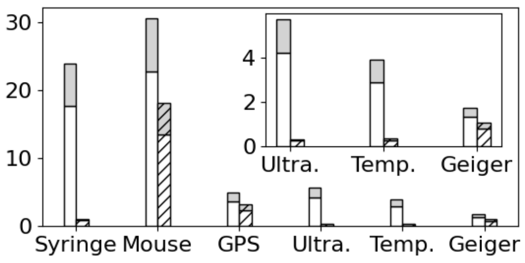
All methods that inspect prior CF_{Log}-s achieve optimizations comparable to or better than Manual Inspection. In most cases, *Top* achieves comparable CF_{Log} reductions to Manual Inspection. However, as it does not consider sub-path size, it normally incurs the largest *BlockMem* overhead of the three CF_{Log} inspection methods. *Minimize* reduces the memory overhead associated with storing sub-paths. In prioritizing smaller sub-paths, *Minimize* often misses better optimizations from longer sub-paths, decreasing CF_{Log} reductions. Of the three methods, *Select* achieves the most consistent optimizations, comparable to *Top* while using less *BlockMem*. For Geiger Counter, GPS, and Mouse, *Top* beats *Select* and *Minimize* since the most occurring sub-paths in these programs have larger lengths. Similarly, *Minimize* outperforms *Select* for Mouse because *Minimize* uses more *BlockMem* than *Select*.

8.4. End-to-End Latency

To assess *SpecCFA*'s end-to-end effect on attested execution performance, we measure the total time taken to perform an attested execution of different operations on $\mathcal{P}$rv. In this case, a series of authenticated CF_{Log} segments must be transmitted to $\mathcal{V}$rf throughout the execution of the



(a) Custom hardware-based *SpecCFA*



(b) TEE-based *SpecCFA*

□ Prv Run-Time (baseline) ▧ Prv Run-Time (SpecCFA)

■ Transmission (baseline) ▨ Transmission (SpecCFA)

Figure 12: End-to-end latency of attested operations compared to baseline (without *SpecCFA*).

application, whenever the CF_{Log}'s designated memory is full. This is required before subsequent transfers can be appended to CF_{Log}. As a consequence, a significant portion of the attested execution time is spent on interruptions by the *CFA* RoT to MAC/sign and transmit a CF_{Log} slice to $\mathcal{V}$rf.

Given *SpecCFA*'s reduced CF_{Log} sizes, fewer transmissions are required. To evaluate this impact on the overall *CFA* performance, we measure the total attested execution time from when $\mathcal{V}$rf requests *CFA* until when attested execution completes (including $\mathcal{V}$rf receipt of all execution evidence). Fig. 12 shows times of the custom hardware-based version and the TEE-based version and zooms in on Ultrasonic Sensor, Temperature Sensor, and Geiger Counter applications.

In this experiment, CF_{Log} slice sizes are set to 256 Bytes, and *SpecCFA* is equipped to support 2 sub-paths. We then measure the total attested execution time (including the execution of the attested program, building CF_{Log}, and generating a signature/MAC over CF_{Log}). This total time is denoted "$\mathcal{P}$rv Run-Time". We also measure the time spent transmitting the CF_{Log}. Several applications present significant reductions in the overall attested execution time results. The Ultrasonic Sensor, Syringe Pump, and Temperature Sensor applications present $\approx$81-95% reduction in $\mathcal{P}$rv Run-Time and $\approx$90-97% reduction in transmission. This performance improvement follows directly from the reduced amount of CF_{Log} data that $\mathcal{P}$rv must authenticate and transmit. With only 2 sub-paths, Geiger, Mouse, and GPS show less pronounced savings of $\approx$13-47% for run-time and $\approx$17-47% for transmission time. This follows from these applications requiring more than 2 sub-paths to exhibit pronounced CF_{Log} reductions (recall Fig. 6). Appendix B contains additional implementation details of cryptographic operations and communication, as implemented by the underlying *CFA* architectures [1], [2].

9. Limitations & Potential Improvements

The initial concept proposed in *SpecCFA* presents several avenues for future work.

Sub-path representation: *SpecCFA*'s initial design represents and stores sub-path speculations verbatim on $\mathcal{P}$rv. It would be interesting to propose other sub-path representations (e.g., using regular expressions or wildcards) that could enable 1 sub-path to be matched to multiple PMEM addresses based on offsets.

Linear Hardware Cost: In the initial design, the cost of *SpecCFA* hardware increases linearly as one Block Detect module is required for each speculated sub-path. Therefore, future work could propose hardware optimizations to reduce *SpecCFA* hardware overhead by serializing the sub-path detection and reducing the number of Block Detect modules.

Additional Automated Sub-path Selection Methods: This work presented four methods for automated selection: one based on binary analysis and three based on inspecting prior CF_{Log}-s. Future work should expand upon these methods to provide additional application-specific suggestions. Since our current program analysis is static, a potential future direction is to develop a system for dynamic analysis of a program and its sub-paths to gain insight into the likelihood of its execution. In addition, our current methods that inspect prior CF_{Log}-s perform pattern matching. In future work, these methods can be extended to perform more advanced feature extraction from CF_{Log}-s, prior input data, and the source code to make stronger predictions about possible future sub-paths.

10. Conclusion

We propose *SpecCFA*: an approach to enable configurable application-aware sub-path speculations in *CFA*. *SpecCFA* provides $\mathcal{V}$rf with the ability to speculate on a program's likely sub-paths to reduce CF_{Log} size significantly. Through *SpecCFA* systematic design, $\mathcal{V}$rf can speculate on various sub-paths of any length without loss of information in CF_{Log}. We implement two versions of *SpecCFA*, based on custom hardware and on TEEs. Our evaluation, performed on *SpecCFA*'s publicly available prototypes [59], demonstrates significant performance improvement for various MCU applications while retaining all standard *CFA* security guarantees.

Acknowledgements

We thank ACSAC'24 anonymous reviewers for their constructive comments. This work was partly funded by the National Science Foundation (SaTC award #2245531).

References

[1] 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

[2] 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.

[3] T. Walker, "OpenSyringePump Github Repository," https://github.com/manimino/OpenSyringePump, 2022.

[4] A. T. Noman, S. Hossain, S. Islam, M. E. Islam, N. Ahmed, and M. M. Chowdhury, "Design and implementation of microcontroller based anti-theft vehicle security system using gps, gsm and rfid," in *2018 4th International Conference on Electrical Engineering and Information & Communication Technology (iCEEiCT)*. IEEE, 2018, pp. 97–101.

[5] 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.

[6] 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.

[7] 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.

[8] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "Toward remotely verifiable software integrity in resource-constrained iot devices," *IEEE Communications Magazine*, vol. 62, no. 7, pp. 58–64, 2024.

[9] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing dynamic) root of trust," in *NDSS*. Internet Society, 2012.

[10] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," *USENIX Security'19*, 2019.

[11] 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.

[12] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *EuroSys*. ACM, 2014.

[13] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "HYDRA: hybrid design for remote attestation (using a formally verified microkernel)," in *Wisec*. ACM, 2017.

[14] 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.

[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 TOPS*, 2017.

[16] 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.

[17] 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.

[18] 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.

[19] M. Ammar, M. Washha, and B. Crispo, "Wise: Lightweight intelligent swarm attestation scheme for iot (the verifier's perspective)," in *2018 14th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, 2018, pp. 1–8.

[20] S. Surminski, C. Niesler, L. Davi, and A.-R. Sadeghi, "Dma'n'play: Practical remote attestation based on direct memory access," in *International Conference on Applied Cryptography and Network Security*. Springer, 2023, pp. 32–61.

[21] 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.

[22] 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," *USENIX Security Symposium*, 2022.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Y. Li, J. M. McCune, and A. Perrig, "Viper: Verifying the integrity of peripherals' firmware," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 3–16.

[31] S. Surminski, C. Niesler, S. Linsner, L. Davi, and C. Reuter, "Scattman: Side-channel-based remote attestation for embedded devices that users understand," in *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*, 2023, pp. 225–236.

[32] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "APEX: A verified architecture for proofs of execution on remote devices under full software compromise," in *USENIX Security*, 2020.

[33] A. Caulfield, N. Rattanavipanon, and I. D. O. Nunes, "Asap: Reconciling asynchronous real-time operations and proofs of execution in simple embedded systems," 2022.

[34] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *IEEE SP*. IEEE, 2015, pp. 745–762.

[35] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 901–913.

[36] 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.

[37] 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.

[38] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[39] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 15–26.

[40] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix: Hardware-assisted flow integrity extension," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[41] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of {Coarse-Grained}{Control-Flow} integrity protection," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 401–416.

[42] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 585–598, 2017.

[43] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX security symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[44] T. Mishra, J. Wang, T. Chantem, R. Gerdes, and N. Zhang, "A procrastinating control-flow integrity framework for periodic real-time systems," in *RTNS*, 2023.

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

[46] 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 *DAC*. ACM, 2017, p. 24.

[47] 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.

[48] 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.

[49] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: lightweight hardware-assisted attestation of program execution," in *2018 IEEE/ACM ICCAD*, 2018, pp. 1–8.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] G. Dessouky, S. Zeitouni, A. Ibrahim, L. Davi, and A.-R. Sadeghi, "CHASE: A configurable hardware-assisted security extension for real-time systems," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[56] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A.-R. Sadeghi, and M. Schunter, "Diat: Data integrity attestation for resilient collaboration of autonomous systems." in *NDSS*, 2019.

[57] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[58] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[59] A. Caulfield, L. Tyler, and I. De Oliveira Nunes, "SpecCFA Prototype Repository," 2024. [Online]. Available: https://github.com/RIT-CHAOS-SEC/SpecCFA/

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

[61] Intel, "Intel Software Guard Extensions (Intel SGX)." [Online]. Available: https://software.intel.com/en-us/sgx

[62] 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.

[63] J. S. Vitter, "Design and analysis of dynamic huffman codes," *Journal of the ACM (JACM)*, vol. 34, no. 4, pp. 825–845, 1987.

[64] 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.

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

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

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

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

[69] M. Vlasák, "arduino-joystick-mouse," 2019. [Online]. Available: https://github.com/Krakenus/arduino-joystick-mouse/blob/master/joystick_mouse.ino

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

[71] Xilinx, "Vivado design suite user guide," 2017.

[72] O. Girard, "openMSP430," 2009.

[73] 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.

[74] K. MacKay, "Github repository for Micro-ECC," https://github.com/kmackay/micro-ecc, 2022.

Appendix A. Repeat Detect Module

Fig. 13 depicts the Repeat Detect module in more detail. As discussed in Sec. 4.1, the Repeat Detect module receives $detect_{any}$ indicating that a sub-path has been detected in CF_{Log}. The module also receives $active_{ID}$ and $active_{addr}$ as an output from the MUX representing the ID of the occurring sub-path and its address in CF_{Log}. Repeat Detect uses these signals to determine if a sub-path has repeated by comparing the current speculation to the previous one. For ease of representation, we depict this logic as the Repeat Counter in Fig. 13. If a repeat is detected, the address of the optimization $last_{addr}$ is determined, and a counter containing the number of consecutive repeats *(*$repeat_{ctr}$*)* is recorded. In addition, two signals are set ($first_{repeat}$ and $subseq_{repeat}$), whether this represents the first time the repeated sub-path occurred.

The final signals $active_{addr}$, $active_{ID}$, $last_{addr}$, and $repeat_{ctr}$ are passed to a MUX using the $detect_{active}$, $first_{repeat}$, and $subseq_{repeat}$ signals as a selector. The MUX determines the values of $spec_{en}$, $spec_{value}$, and $spec_{addr}$ depending on if the detected sub-path is repeating. Upon the first occurrence of a sub-path, $active_{ID}$ and $active_{addr}$ are written to $spec_{value}$ and $spec_{addr}$. Then, if the sub-path repeats, they are set to $repeat_{ctr}$ and $last_{addr}$ instead. In both cases, $spec_{en}$ is set. Then, $spec_{en}$, $spec_{addr}$, and $spec_{value}$ are sent as output to the Memory Interface.

A.1. Repeat Detect Hardware Specifications

Fig. 14 shows the additional hardware specifications for detecting when a sub-path repeats in adjacent CF_{Log} locations. After a sub-path is detected and written to CF_{Log} for the first time, $repeat_{ctr} = 2$. At this moment, the internal value $last_{addr}$ is set in order to save the previous value of $active_{addr}$, and $last_{ID}$ records the previous $active_{ID}$. These values and $repeat_{ctr}$ are then used to detect when the first ($first_{repeat}$) or subsequent repeats ($subseq_{repeat}$) of the sub-path occur.

To detect the first repeat, the hardware checks if the next $active_{ID}$ is the same as $last_{ID}$ and if $repeat_{ctr} = 2$. If the IDs match, it then checks if $active_{addr}$ and $last_{addr}$ are adjacent by comparing $last_{addr}+2$ to $active_{addr}$. If all three conditions are met, the first repeat has occurred. $repeat_{ctr}$ is then written to CF_{Log} and incremented. To detect all subsequent repeats, a similar process occurs. If $last_{ID}$ and $active_{ID}$ are equal, $last_{addr}$ and $active_{addr}$ are adjacent, and $repeat_{ctr} > 2$, a subsequent repeat has occurred.

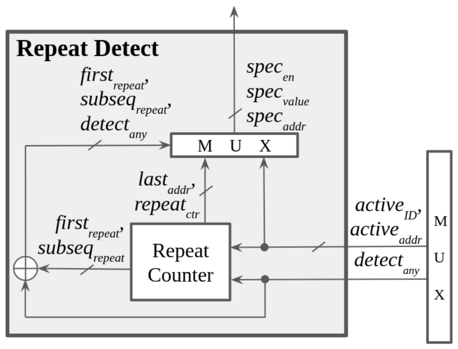


Figure 13: Repeat Detect module internals

HW Specification: Select active block signals

$$active_{ID} = \begin{cases} block_{ID_0}, & \text{if } detect_{active_0} \\ \vdots & \\ block_{ID_n}, & \text{else if } detect_{active_n} \end{cases} \quad (1)$$

$$active_{addr} = \begin{cases} active_{addr_0}, & \text{if } detect_{active_0} \\ \vdots & \\ active_{addr_n}, & \text{else if } detect_{active_n} \end{cases} \quad (2)$$

$$(detect_{active_0} \lor ... \lor detect_{active_n}) \rightarrow detect_{any} \quad (3)$$

Definitions: Previous speculation address and ID

$$last_{addr} = \begin{cases} active_{addr}, & \text{if } detect_{any} \land (repeat_{ctr} = 2) \\ last_{addr} & \text{otherwise} \end{cases} \quad (4)$$

$$last_{ID} = \begin{cases} active_{ID}, & \text{if } detect_{any} \land (repeat_{ctr} = 2) \\ last_{ID} & \text{otherwise} \end{cases} \quad (5)$$

Definitions: Detect First and Subsequent Sub-Path Repeats

$$\begin{aligned} first_{repeat} :=& (repeat_{ctr} = 2) \land (active_{ID} = last_{ID}) \\ & \land ((last_{addr} + 2) = active_{addr}) \end{aligned} \quad (6)$$

$$\begin{aligned} subseq_{repeat} :=& (repeat_{ctr} > 2) \land (active_{ID} = last_{ID}) \\ & \land ((last_{addr} + 2) = active_{addr}) \end{aligned} \quad (7)$$

$$repeat := detect_{any} \land (first_{repeat} \lor subseq_{repeat}) \quad (8)$$

$$repeat_{ctr} = \begin{cases} 2, & \text{if } (detect_{any} \land \neg repeat) \\ repeat_{ctr} + 1 & \text{else if } detect_{any} \land repeat \end{cases} \quad (9)$$

Figure 14: Hardware Spec: Repeat Detection

HW Specification: Output speculation values

$$spec_{en} = detect_{any} \lor first_{repeat} \lor subseq_{repeat} \quad (10)$$

$$spec_{value} = \begin{cases} repeat_{ctr}, & \text{if } first_{repeat} \lor subseq_{repeat} \\ active_{ID}, & \text{else if } detect_{any} \end{cases} \quad (11)$$

$$spec_{addr} = \begin{cases} last_{addr} + 2, & \text{if } first_{repeat} \\ last_{addr}, & \text{else if } subseq_{repeat} \\ active_{addr}, & \text{else if } detect_{any} \end{cases} \quad (12)$$

Figure 15: Hardware Spec.: Interface with *BlockMem*

Repeat Detect outputs $spec_{en}$, $spec_{addr}$, and $spec_{value}$ to optimize CF_{Log}. The hardware specifications for these signals are shown in Fig. 15. $spec_{en}$ is set as the logical OR of the signals $detect_{any}$, $first_{repeat}$, and $subseq_{repeat}$ as any of these signals indicate a sub-path has been detected. When any repeat is occurring ($first_{repeat} \lor subseq_{repeat}$), $repeat_{ctr}$ is used to set $spec_{value}$. When no repeat occurs, and $detect_{any}$ is set, then $spec_{value}$ is set as $active_{ID}$. Similarly, $spec_{addr}$ depends on these three signals ($detect_{any}, first_{repeat}, subseq_{repeat}$). When the first repeat occurs, the counter must be logged to the address adjacent to the first sub-path ID, which is determined by $last_{addr}+2$. On subsequent repeats, $last_{addr}$ contains the address of the counter in CF_{Log} and can be overwritten directly with the new count. Finally, when there is no repeat but $detect_{any}$ is set, the address stored in $active_{addr}$ is used.

Appendix B. Additional Implementation Details

Platform differences effecting CF_{Log} size: Compared to MSP430, ARMv8 Cortex-M has a richer instruction set. Because of this, some control flows can be optimized to make execution more efficient. For example, simple if statements and logical operations can be replaced with the execution of conditional instructions, removing certain control flow transfers. We observe the effect of this difference in programs like Mouse and GPS. Several components of these programs, such as integer division and logical operations (such as setting a variable to the result of a comparison), are optimized in ARM, reducing the number of transfers. In addition, the STM32L552ZE MCU is equipped with an FPU, allowing floating point operations to require little to no control flow transfers. In MSP430, divisions require several branch instructions, and logical operations require control flow transfers. Hence, the CF_{Log} reductions and selected sub-paths vary because some recurrent sub-paths in the control flow path of MSP430 binaries do not exist in ARMv8 binaries.

Authenticating and Transmitting CF_{Log}-s: Both implementations use UART-to-USB as a communication interface with a baud rate of 38400. The two use different cryptographic functions to authenticate CF_{Log}-s. The underlying *CFA* RoT [2] for the TEE-based prototype uses a digital signature. It is implemented with SHA256 from HACL* [73] for hashing and Micro-ECC [74] for the Elliptic Curve Digital Signature Algorithm (ECDSA). It operates on an SECP256R1 curve with a 256 bits private key to generate a 64 Byte signature. For the custom hardware-based design, the underlying *CFA* RoT[1] uses SHA256-HMAC from HACL* [73] to produce a 32 Byte MAC. Because of these differences, cryptographic operations consume most of the protocol time on the TEE-based version (due to the use of an asymmetric primitive) whereas transmission of data is the most time-consuming on the custom hardware-based version, as depicted in Fig 12.