RAP-Track: Efficient Control Flow Attestation via Parallel Tracking in Commodity MCUs

Antonio Joia Neto
University of Zurich
ajneto@ifi.uzh.ch

Adam Caulfield
University of Waterloo
acaulfie@uwaterloo.ca

Ivan De Oliveira Nunes
University of Zurich
nunes@ifi.uzh.ch

***Abstract*—Control Flow Attestation (CFA) has emerged as an important security service to enable remote verification of control flow paths in safety-critical embedded systems. However, current CFA for commodity devices suffers performance penalties due to code instrumentation and frequent context switches required to securely log control flow paths at runtime. Our work introduces RAP-Track, a technique leveraging commodity hardware extensions, namely Micro Trace Buffer and Data Watchpoint and Trace Unit, to track control flow paths in parallel with the execution of the attested program, thus avoiding aforementioned overheads present in state-of-the-art CFA. Our evaluation (based on an open-source prototype of RAP-Track) demonstrates substantial performance gains, enhancing practicality and security of CFA.**

I. Introduction

Embedded systems play vital roles in interfacing the physical and digital worlds. They are often deployed remotely for on-demand sensing and/or actuation in critical operations. In these settings, low-power micro-controller units (MCUs) are preferred for their energy efficiency. MCUs – typically having low CPU frequencies with minimal resources – lack the robust security features found in fully-fledged computers, such as memory management units (MMUs), virtual memory, and inter-process isolation. As a result, they become prime targets for attacks [1].

In this landscape, Remote Attestation (RA) [2] has become a popular method to remotely verify the integrity of MCUs. In RA, a Verifier (Vrf) wants to ascertain the current software state of a remotely deployed Prover MCU (Prv). This is realized as a challenge-response protocol in which Vrf sends a challenge (in the form of a cryptographic nonce) to Prv. Upon receiving the challenge, a Root of Trust (RoT) in Prv is responsible for computing an authenticated integrity-ensuring function (e.g., a MAC or digital signature) on the received challenge and Prv's program memory. Based on the result, Vrf can determine whether Prv's code has been illegally modified.

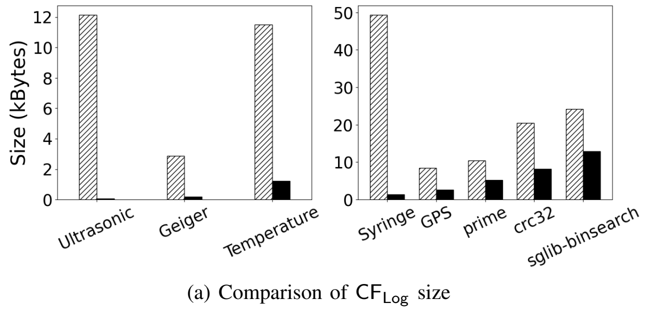
By itself, RA does not detect runtime attacks (e.g., Return/Jump-Oriented Programming [3], [4]) that corrupt software execution without modifying code. Control Flow Integrity (CFI) [5], [6], [7], [8], [9], [10], [11] can, to some extent, detect runtime attacks locally on Prv and abort execution. However, it does not produce attestable evidence of the unintended control flow paths that led to the exploit [12]. Alternatively, Control Flow Attestation (CFA) generates authenticated evidence of Prv's runtime behavior, granting remote visibility to runtime attacks. CFA augments RA by including Prv's execution control flow path (for a given Vrf-defined operation/function of interest) as part of the attestation evidence received by Vrf. To accomplish this, the RoT within Prv saves all control flow transfer destinations (e.g., due to jump, return, and call instructions) during an operation's execution in a control flow log ($\mathsf{CF_{Log}}$). Then, $\mathsf{CF_{Log}}$ is signed along with the received challenge and Prv's program memory and sent to Vrf for assessment.

Current CFA RoTs build $\mathsf{CF_{Log}}$ with either: (1) customized hardware that interfaces with the CPU to detect/log control flow instructions in parallel to their execution [13], [14], [15], [16]; or (2) instrumentation of the attested application with calls to Trusted Execution Environment (TEE)-protected software that tracks the control flow path before returning to the application [17], [18], [19], [20], [21]. Both approaches involve trade-offs. Mechanisms relying on custom hardware require new MCU chip fabrication, making them unsuitable for immediate deployment in commercial devices. On the other hand, CFA mechanisms that use TEEs can be readily deployed on current devices. However, associated code instrumentation leads to significant overheads due to added instructions and frequent context switches into TEE-protected mode for control flow logging. Only one prior technique [22] leverages a commodity hardware feature – Intel Processor Trace (PT) – for CFA, thus reducing code instrumentation. However, this feature is only available in high-end Intel CPUs, making the approach infeasible in MCUs.

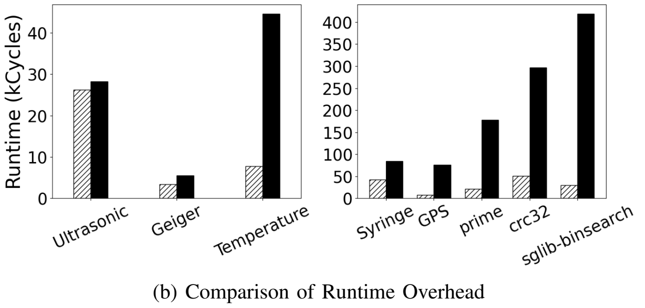
In terms of MCU-specific hardware features, ARM has introduced the Micro-Trace Buffer (MTB), enabling real-time control flow tracing of a program's execution [23]. Additionally, the Data Watchpoint and Trace (DWT) extension allows for monitoring of the processor's operations based on Program Counter (PC) values. Naturally, prior work has leveraged MTB for Control Flow Integrity (CFI) [5], [6] without instrumentation. Yet, the use of MTB for CFA introduces unique challenges that remain to be addressed. Unfortunately, MTB logs all control flow transfers, including (a very large number of) branches to fixed addresses (a.k.a. deterministic branches) that are unnecessary for CFA, given their fixed behavior. Since CFI processes this data locally on Prv, this overhead (while substantial) may be acceptable if Prv's execution can be periodically interrupted to examine and discard the generated control flow data. For CFA, since the entirety of $\mathsf{CF_{Log}}$ is transmitted to Vrf, the latter creates a significant bottleneck.

In sum, despite reducing runtime overhead due to context switches and instrumentation, the *naive* usage of MTB to log transfers would generate much larger $\mathsf{CF_{Log}}$-s than state-of-the-art TEE-based CFA. As a consequence, more memory to store $\mathsf{CF_{Log}}$ would be demanded (on already limited MCUs), or constant interruptions to clear unnecessary $\mathsf{CF_{Log}}$ entries would add delays to application runtime. Figure 1 illustrates this point, showing memory and timing demands of *naive* MTB-based logging in comparison to a state-of-the-art instrumentation-based CFA [24] on various open-source embedded applications: an ultrasonic ranger [25], a Geiger sensor [26], a Syringe Pump [27], a Temperature sensor [28], a GPS [29], as well as test applications from the BEEBs benchmark [30]. As seen in Figure 1(a), $\mathsf{CF_{Log}}$ sizes generated by a *naive* MTB-based approach are significantly larger (1.9 to 217× larger on tested applications). On the other hand, instrumentation-based CFA adds significant runtime overheads (e.g., 1.1 to 14.1× increase on tested applications), as shown in Figure 1(b).

In this work, we propose an approach to combine MTB and DWT hardware extensions with an optimal strategy for linking of attested code sections. With this approach, we are able to reduce both $\mathsf{CF_{Log}}$



(a) Comparison of $\mathsf{CF_{Log}}$ size



(b) Comparison of Runtime Overhead

▨ Naive MTB ■ Instrumentation-based CFA

Fig. 1: Comparison $\mathsf{CF_{Log}}$ sizes and runtime overhead of *naive* MTB- and Instrumentation-Based CFA

storage/transmission costs (due to *naive* MTB-based logging) and runtime overheads (due to code instrumentation and frequent context switches required by prior work).

Contribution. We propose RAP-Track: a method for Runtime Attestation via Parallel Tracking in commodity MCUs. RAP-Track leverages commodity hardware features of ARM MCUs – namely, MTB, DWT, and TrustZone (TZ) – in combination with strategic linking of attested code sections. DWT is used to define memory regions that control MTB activation and deactivation. RAP-Track offline static analysis module links code sections so that only branches that are essential for complete control flow reconstruction by Vrf are logged while also obviating context switches into TEE-protected mode during execution. We implement and evaluate RAP-Track on an ARM Cortex-M33 MCU using the V2M-MPS2-0318C prototyping system [31]. RAP-Track prototype is publicly available at [32].

II. Background & Related Work

A. Armv8-M Trustzone

TZ is a security extension for ARMv8 Cortex-M MCUs that creates a TEE by partitioning CPU resources into two separate domains: "Secure" and "Non-Secure" worlds. Each domain is configured with its own isolated memory and peripherals [33]. When the CPU is in a Non-Secure state, it can access only Non-Secure resources. In the Secure state, it can access both Secure and Non-Secure resources but is restricted to executing only Secure code. In the absence of virtual memory or MMUs, Cortex-M uses a Memory Protection Unit (MPU) for physical memory access control. With TZ, the MPU splits into Secure (S-MPU) and Non-Secure (NS-MPU) segments, with S-MPU access restricted to the Secure World and NS-MPU accessible to both worlds.

B. ARM Hardware Tracing Units

1) MicroTrace Buffer: The MTB [23] is a hardware extension introduced in ARM Cortex-M processors to enable lightweight, real-time control flow tracing of program execution. The MTB operates by recording every non-sequential memory address executed while it is activated. Unlike software-based instrumentation, which introduces significant performance overhead, the MTB operates with minimal impact on the CPU's performance.

The MTB uses a small, circular buffer located in the system's memory to store the trace data. As the program executes, the MTB captures branch instructions and writes the source address and destination address of the instruction into this buffer. In RAP-Track, this buffer contains $\mathsf{CF_{Log}}$. Once the buffer is full, it wraps around and overwrites the oldest data, ensuring that the most recent execution flow is always available for analysis. It is possible to configure a Watermark (using MTB_FLOW register) to set a limit to the buffer that will generate a debug exception when the buffer reaches the Watermark. RAP-Track leverages this feature to generate partial reports (discussed further in Sec. IV-E).

There are two primary methods to activate MTB tracing. The first method involves directly setting the 'TSTARTEN' bit in the 'MTB_MASTER' register, which enables the recording of all non-sequential branches from that point forward. Alternatively, the 'MTB_TSTART' and 'MTB_TSTOP' registers can be used to start and stop the MTB tracing based on signals from the Data Watchpoint and Trace comparators.

2) Data Watchpoint and Trace: The DWT unit in the ARM Cortex-M33 processor [23] is a hardware extension that provides a way to monitor processor operations to specific memory ranges (such as reads, writes, and executions). It includes up to four configurable comparators, each of which can configured to generate *watchpoint* events when a specific memory address is accessed [34]. The DWT is integrated with the MTB, allowing DWT comparators to activate and deactivate the MTB Tracing based on DWT events. For example, two DWT comparators can be set to monitor a specific address range. Then, when the software within this range executes, the DWT triggers an event to MTB_START, initiating control flow tracing. Another set of DWT comparators can be configured to trigger MTB_STOP, ending the trace when the PC exits the specified range. RAP-Track uses the DWT in this way to activate/deactivate the MTB (discussed further in Sec. IV-B).

C. Remote Attestation

Remote attestation (RA) is a security service that allows a Verifier (Vrf) to assess the integrity of the code of a remote device, known as the Prover (Prv). The RA protocol usually follows four steps:

(1) Vrf creates a unique Chal and sends an RA request to Prv.
(2) Prv receives the request and measures its code or internal state, generating the attestation proof.
(3) Prv produces a proof that is authenticated with a signature or message authentication code (MAC).
(4) Vrf receives the proof and verifies whether Prv has a valid internal state.

The signature/MAC operation attests to Vrf that Prv's RoT is the generator of the proof. For this purpose, a secret key must be securely stored and managed exclusively by the RoT, ensuring it remains protected from any untrusted software on Prv. As such, RA mechanisms usually depend on hardware support (whether a fully hardware-based mechanism using standalone cryptographic co-processors [35], [36], [37] or a hybrid scheme using partial hardware support [38], [39], [40], [41], [42], [43], [44], [45]).

D. Control Flow Attestation

Control Flow Attestation (CFA) extends RA by also providing evidence that an application's instructions were executed in a correct sequence. This capability provides detection of Return-Oriented Programming (ROP) [3] and Jump-Oriented Programming (JOP) [4] attacks that can manipulate existing return/jump instructions to achieve Turing-complete behavior without altering the code.

CFA is achieved by requiring the RoT in Prv to generate and measure the control flow path followed during the application's execution. This requires recording all control flow transfers (e.g, due to jumps, calls, or returns) into $\mathsf{CF}_{\mathsf{Log}}$. Current CFA approaches employ either hardware [16], [14], [13], [15], [46], [22], [47] or instrumentation-based methods [48], [49], [17], [18], [24], [19], [20], [21], [50], [51] to create $\mathsf{CF}_{\mathsf{Log}}$. Hardware-based methods introduce custom hardware modules that interface with the CPU to detect/log control flow transfers as they occur [16], [14], [13], [15]. While efficient, these methods are incompatible with commodity devices. In contrast, instrumentation-based methods use existing TEE support (such as ARM TZ) for secure logging of an instrumented application [17], [18], [24], [19], [20], [21], [50].

TEE-based methods usually instrument all the control flow instructions in the application code, adding instructions that call the TEE (via dedicated non-secure callable entry points) to record each branch destination into $\mathsf{CF}_{\mathsf{Log}}$-dedicated Secure World memory. Due to frequent context switches, these approaches typically add significant runtime overhead, as each branch requires a transition from the Non-Secure to the Secure World to record the event. To mitigate this overhead, different CFA methods operate at varying levels of control flow granularity, selectively logging only critical code sections to reduce $\mathsf{CF}_{\mathsf{Log}}$ overhead [19], [20]. We refer to these selective methods as "lossy" CFA, where Vrf cannot reconstruct the entire control flow path. In contrast, "lossless" CFA enables Vrf to fully recover the application's control flow path. The advantage of lossless CFA is that by recovering the full control flow path in $\mathsf{CF}_{\mathsf{Log}}$, Vrf can validate the entire execution path and observe any unintended/maliciously induced transitions. Additionally, lossless evidence could potentially grant visibility to attacks that do not directly overwrite branch destinations [12]. Naturally, $\mathsf{CF}_{\mathsf{Log}}$-s generated by lossless control flow mechanisms are larger, which correlates to more context switches for appending $\mathsf{CF}_{\mathsf{Log}}$ [17], [18], [20], [19], [21], [24], [50].

To avoid runtime costs of instrumentation, dedicated hardware tracing components can be used, when available. Papamartzivanos et al. [22] proposed a technique that leverages Intel PT to trace low-level mission-critical processes. However, since Intel PT monitors everything within a code region, applying it alone can lead to large $\mathsf{CF}_{\mathsf{Log}}$-s. Thus, the authors consider analysis within the context of one target function. Additionally, [22] cannot be directly applied to embedded systems since it relies on Intel PT, which is only available in high-end Intel processors. A similar strategy is deployed by LAHEL [46], which uses off-chip debug features of ARM Coresight System Trace Macrocell technology in combination with TrustZone to implement CFA. In this case, the Program Trace Macrocell (PTM) [52] debug component is used to monitor program's execution in parallel. LAHEL leverages the PTM to log all branches taken and records the count of all branches not taken. Consequently, LAHEL records repeated branch-taken addresses due to loops. Additionally, LAHEL relies on external debug features rather than on-chip tracing units, such as MTB and DWT.

To our knowledge, MTB and DWT have only been used as a building block for CFI schemes [6], [5]. However, CFI has a fundamentally different goal than CFA: CFI mechanisms aim to locally detect control flow violations and abort execution upon detection, whereas CFA aims to generate authenticated evidence of the control flow path that was followed, whether it is benign or malicious [12]. Given their distinct goals, schemes that used MTB/DWT for efficient CFI cannot be applied directly to achieve efficient CFA. For instance, CFI methods utilizing MTB/DWT add additional steps into the local verification process that first filter redundant information from the MTB. Since CFA performs no local verification, the communication cost of $\mathsf{CF}_{\mathsf{Log}}$ when applying MTB would be incredibly costly, nullifying improvements on the application's runtime.

III. System and Adversary Model

Prv is a single-core MCU equipped with a TEE (e.g., TZ in RAP-Track) running software at "bare-metal". The attested application (APP) executes in the Non-Secure World. RAP-Track implementation is housed in the Secure World (which is trusted). Prv is equipped with MTB and DWT hardware extensions. MTB and DWT are trusted to correctly implement their specification.

Adv fully controls Prv's Non-Secure World and is able to alter its code (e.g., via code injection attacks) or launch control flow/code reuse attacks. Given TZ protections (discussed in Section II-A), Adv cannot access or tamper with code or data in the Secure World. Similarly, it cannot deactivate or circumvent TZ hardware-enforced access control rules and assurances. Physical and hardware-modifying attacks require orthogonal tamper resistance measures [53] and are thus out our scope.

In this work, our treatment of CFA assumes that Non-Secure World interrupts are disabled during the execution of APP. We treat interrupt enablement as an orthogonal issue and refer the reader to [18], [54], [55] for a discussion on security implications and approaches that could complement RAP-Track by supporting safe interruptability.

IV. RAP-Track Design

RAP-Track tailors MTB/DWT usage for efficient CFA. RAP-Track's use of MTB reduces the number of Secure World calls for updating $\mathsf{CF}_{\mathsf{Log}}$, while its static linking strategy prevents logging irrelevant information (e.g., deterministic or repeated branch destinations) into $\mathsf{CF}_{\mathsf{Log}}$ by MTB. As a result, RAP-Track reduces both the runtime cost of instrumentation and the storage cost of MTB-generated $\mathsf{CF}_{\mathsf{Log}}$-s. As shown in Figure 2, RAP-Track high-level operation includes:

Offline Phase. Prior to code deployment, RAP-Track static analysis phase divides APP into two regions: the MTB Activation Region (MTBAR) and the MTB Deactivation Region (MTBDR) (see Section IV-B for details on each region). All deterministic control flow transfers of APP are placed in the MTBDR, while non-deterministic transfers (e.g., indirect jumps/calls, returns, and conditional branches) are moved to the MTBAR. Their previous locations in APP are replaced with direct branches (i.e., *trampolines*) targeting their new addresses in MTBAR (details in Section IV-C). When the MCU executes a trampoline instruction, it directly branches to the MTBAR region, where the original branch operation is executed and thus measured by the MTB. This approach ensures that only non-deterministic branch destinations are recorded to $\mathsf{CF}_{\mathsf{Log}}$.

Execution Phase. When Prv receives a request for CFA of APP, it initializes the CFA Engine (Section IV-A) to start the CFA process. In RAP-Track this includes configuring MTB to track APP 's control flow path. APP execution then starts in the Non-Secure World. During APP's execution, branches that were linked within MTBAR are logged to $\mathsf{CF}_{\mathsf{Log}}$ by the MTB. Once APP execution concludes, an authenticated CFA report (as outlined in Sections II-C and II-D) is generated and sent to Vrf.

A. CFA *Engine*

To start CFA of an APP, the CFA Engine disables all the Non-Secure interrupts and configures the NS-MPU to make APP's binary

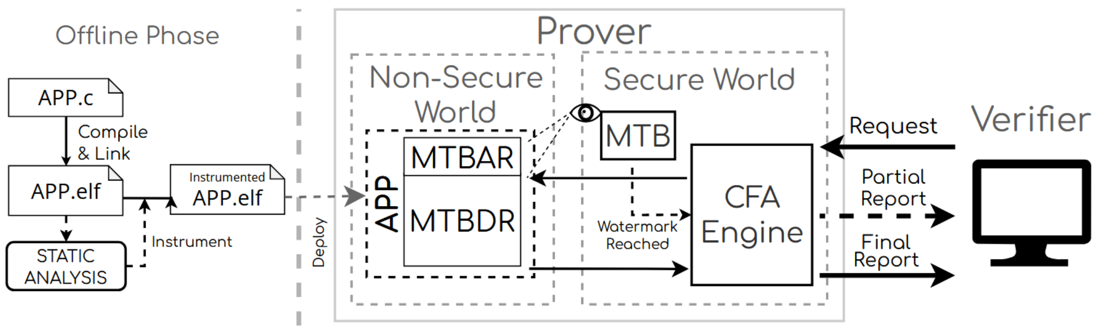


Fig. 2: System Model

non-writable. The NS-MPU is then locked to prevent modifications by the Non-Secure World, following prior CFA methods (e.g., [24]). Afterward, the CFA engine hashes all code associated with APP, producing H_{MEM}. Next, it configures the DWT and MTB so that MTB is enabled while executing within MTBAR and disabled while executing within MTBDR. After this setup, APP execution is called in the Non-Secure World. As a consequence of this configuration, for each section of APP code executed in MTBAR, the MTB logs the source and destination of all non-sequential branch operations to CF_{Log}. Once execution concludes, RAP-Track implementation in the Secure World signs a report containing Chal, H_{MEM}, and CF_{Log} using a private key stored within the Secure World and sends the report to Vrf.

B. DWT-based MTBAR and MTBDR Activation

RAP-Track static analysis divides APP code layout into MTBAR and MTBDR based on the instruction type, and inserts trampolines to connect them (discussed further in Section IV-C). All instructions with non-deterministic branch destinations are added into MTBAR, and everything else is added into MTBDR (i.e., non-branch instructions and deterministic branches). Based on this organization of instructions, RAP-Track must activate the MTB when executing the MTBAR region, and deactivate when executing the MTBDR region. Activation/deactivation is done via DWT as follows:

MTBAR: Two DWT comparators are used to define the MTBAR's boundaries: one for the base address and the other for the upper limit address. DWT then automatically activates the MTB when the address of the currently executing instruction (PC) is within these bounds. This is achieved by having the DWT set the 'MTB_TSTART' register when PC points to an address within MTBAR, thus activating the MTB. Based on this configuration, MTB does not record transitions from MTBDR into MTBAR.

MTBDR: Two additional DWT comparators are used in a similar fashion, except in this case they set 'MTB_TSTOP' register when PC is within the bounds of MTBDR. Based on this configuration, MTB does record transitions from MTBAR to MTBDR.

C. Branch Trampolines

This section explains how RAP-Track instruments different types of branch instructions to be able to track APP's control flow.

Statically Deterministic Branches. RAP-Track does not track static branches, such as jumps to constants and simple loops with fixed iteration counts because their destinations cannot be modified at runtime. Static branch instructions remain in MTBDR, as their targets are can be obtained statically by Vrf from APP's binary [19], [24] and need not be logged.

Non-Deterministic Branches includes branches caused by indirect jumps/calls, function returns, loops with variable iteration counts, and loops with internal branches, where the target address is not static/deterministic. Target addresses of these instructions must be recorded in CF_{Log}, as they are necessary for Vrf to reconstruct the entire control flow path. These instructions are moved from their original location in MTBDR to MTBAR. *Trampoline* instructions targeting the new addresses in MTBAR replace the original branch instructions in MTBDR. The *trampoline* type depends on the type of non-deterministic branch instruction.

Original Code

```
(1)   BLX Register
```

Modified Code

```
(1)   BL MTBAR_ADDRESS
              ...
MTBAR_ADDRESS:
          BX Register
```

Fig. 3: Trampoline for indirect call

1) Indirect Calls.: Indirect calls perform a call to a destination specified in a register. Figure 3 shows the trampoline placed by RAP-Track for indirect calls. Each indirect call is replaced with a direct call to a fixed address within the MTBAR (`MTBAR_ADDRESS` in Figure 3). Then, at `MTBAR_ADDRESS`, an indirect branch to the original register is placed.

Original Code

```
POP {PC}
```

Modified Code

```
B MTBAR_POP_ADDR
...
MTBAR_POP_ADDR:
      POP {PC}
```

Original Code

```
LDR PC, [sp, #4]
```

Modified Code

```
B MTBAR_LDR_ADDR
...
MTBAR_LDR_ADDR:
      LDR PC, [sp, #4]
```

Fig. 4: Trampolines for returns and indirect jumps

2) Returns and Indirect Jumps.: In ARMv8-M, when a call instruction is executed, the return address is stored in a reserved register called the Link Register (`LR`). The compiler pushes `LR` onto the stack only if the called function contains a nested function call; otherwise, the return is performed using a branch to `LR`. In this case, if no operations within the function modify `LR`, the return address remains unchanged throughout the function's execution, making it predictable. Therefore, RAP-Track does not monitor these cases and instead focuses on monitoring return operations only when the contents of `LR` have been pushed onto the stack.

Given this insight, monitored return instructions and indirect jumps (used for C-switch statements) have equivalent implementations as either: (I) a `POP` to the PC; (II) or a memory load `LDR` into PC.

RAP-Track inserts trampolines for both cases in the same way, as depicted in Figure 4. First, a direct branch to the MTBAR is inserted in place of the original return/indirect jump instruction. A direct branch to `MTBAR_POP_ADDR` (or `MTBAR_LDR_ADDR`) is used to replace a return/indirect jump via `POP` (or `LDR`) instruction. Once in the MTBAR, the original `POP` (or `LDR`) instruction is executed to complete the return/indirect jump.

3) Conditional Branches: In C, both if/else statements and any form of loop use conditional branch instructions. To optimize the information recorded by MTB to $\mathsf{CF}_{\mathsf{Log}}$, RAP-Track conditional branch trampoline varies in three cases:

(1) non-loop conditional branches
(2) loops conditional branches that jump "backward"
(3) loops conditional branches that jump "forward"

```
                                         Modified Code
 Original Code
                                      CMP  R0,#0
     CMP  R0,#0                  (1)  BEQ  MTBAR_ADDRESS
 (1) BEQ  taken_address                   ...
                                 MTBAR_ADDRESS:
                                     B  taken_address
```

Fig. 5: Trampoline for non-loop conditional branches

3.1) Non-loop Conditional Branches. When a conditional branch is not part of a loop (e.g., used to implement an if/else statement), compilers often optimize the conditional branches of if/else statements by branching to the less likely path, leaving the more probable path as a fall-through, which helps avoid CPU pipeline stalls. Based on this insight, RAP-Track inserts a trampoline so that MTB only records the branch-taken address to minimize the size of $\mathsf{CF}_{\mathsf{Log}}$, as it is the less likely direction of the instruction (where the most likely path is implicitly represented by the absence of an entry in $\mathsf{CF}_{\mathsf{Log}}$). In the example of Figure 5, the branch-taken address of conditional branch instruction (`BEQ`) is replaced with a fixed address in MTBAR. Then in MTBAR, a direct branch is added that jumps to the original branch-taken address.

```
                                     Modified Code
 Original Code                  loop_start:
                                       ...
 loop_start:                        CMP  R0,#0
        ...                    (1)  BEQ  MTBAR_ADDRESS
      CMP  R0,#0                       ...
 (1)  BEQ  loop_start           MTBAR_ADDRESS:
                                    B  loop_start
```

Fig. 6: Trampoline for backward loop conditional branches

3.2) Backward Loop Conditional Branches. In contrast to non-loop conditional branches, the more likely branch destination depends on whether the loop conditional branch is implemented with a "backward" or "forward" conditional branch. First, we consider a loop that is implemented with a backward conditional branch. We refer to "backward" conditional branches as those with a destination address that is smaller than the current instruction address. When implemented with a backward conditional branch, the branch destination indicates the loop's start, as depicted in Figure 6. RAP-Track should ensure $\mathsf{CF}_{\mathsf{Log}}$ reflects all iterations of the loop to ensure evidence is lossless. Therefore, a trampoline similar to the non-loop condition case is added to log each branch-taken address.

3.3) Forward Conditional Branches. We refer to conditional branches where the destination address is greater than the current address as "forward" branches. Loops, in some cases, can be implemented with forward branches. When this occurs, the branch-taken address is the loop's exit, and the loop is continued through the branch-not-taken path. Therefore in order to record each loop iteration in $\mathsf{CF}_{\mathsf{Log}}$, RAP-Track inserts trampolines for forward conditional loop branches to log the not-taken address. This is depicted in Figure 7.

```
                                      Modified Code
 Original Code                            CMP  R0,#0
                                          BEQ  loop_end
        CMP  R0,#0                (1)->   B  MTBAR_ADDRESS
        BEQ  loop_end             br_not_taken_addr:
        ADD  R1, R2, R3                   ADD  R1, R2, R3
        ...                               ...
 loop_end:                        loop_end:
        ...                               ...
                                  MTBAR_ADDRESS:
                                      B  br_not_taken_addr
```

Fig. 7: Trampoline for forward loop conditional branches

In the modified code, the compare and conditional branches are not modified since they implement the loop exit. Instead, RAP-Track inserts a direct branch directly after the conditional branch to the MTBAR to log the branch-not-taken address. This branch jumps to a fixed address in the MTBAR that trampolines back to the modified code at the original branch-not-taken address. In this example, it branches back to `br_not_taken_addr`, the location of the `ADD` instruction that previously followed the loop conditional branch.

Adding the loop trampolines depicted in Figures 6 and 7 is only required when there are non-deterministic branch instructions within the loop or when the loop condition is based on a variable.

D. Loop Optimization

Several CFA methods (e.g., [17], [19], [24], [15]) add loop optimizations to avoid redundant $\mathsf{CF}_{\mathsf{Log}}$ entries. To reduce the impact of loops on $\mathsf{CF}_{\mathsf{Log}}$, RAP-Track adds minimal instrumentation to log the loop condition itself, similarly to prior work [50], [24]. RAP-Track adds this instrumentation when the loop comparison is made to a fixed constant value, the loop iterator is based on register-only operations (e.g., only arithmetic instructions rather than loads/stores), and when its internal branches are all deterministic. These "simple" loop characteristics are very common in MCU applications (e.g., to implement data operations over buffers or control loops that sense data for fixed periods). The instrumentation adds a call to the CFA Engine before the loop entry to log the loop condition in $\mathsf{CF}_{\mathsf{Log}}$.

E. Partial Reports

Prv might be equipped with limited memory to store $\mathsf{CF}_{\mathsf{Log}}$ that fills in the middle of APP execution. To manage this, RAP-Track sets the 'MTB_FLOW' register to limit $\mathsf{CF}_{\mathsf{Log}}$'s size. When $\mathsf{CF}_{\mathsf{Log}}$ size equals the watermark, an exception is triggered in the Secure World to generate and transmit a partial report to Vrf. Afterward, the head pointer of $\mathsf{CF}_{\mathsf{Log}}$ is reset, and then APP resumes, allowing MTB to overwrite the same memory designated for storing $\mathsf{CF}_{\mathsf{Log}}$.

F. Security Analysis

To bypass RAP-Track, Adv must alter control flow in a way that goes undetected while producing valid proof accepted by the Vrf. One approach is modifying APP binary directly. However, CFA Engine hashes and locks APP 's memory: any changes trigger a memory fault, invalidating the report. Attempts to manipulate control flow without detection fail, as all indirect branches are logged to $\mathsf{CF}_{\mathsf{Log}}$ by MTB, and $\mathsf{CF}_{\mathsf{Log}}$ is built and stored within protected memory in the Secure World. Forging or replaying the report is also infeasible as long as a cryptographically secure signature (or MAC, in the symmetric setting) and fresh challenge are used for each CFA request. Finally, Adv cannot deactivate/misconfigure MTB or DWT because they are only configurable by the Secure World.

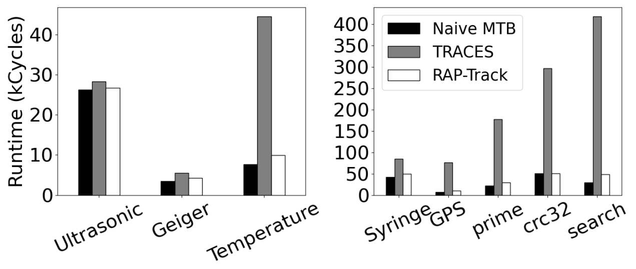


Fig. 8: Runtime Comparison

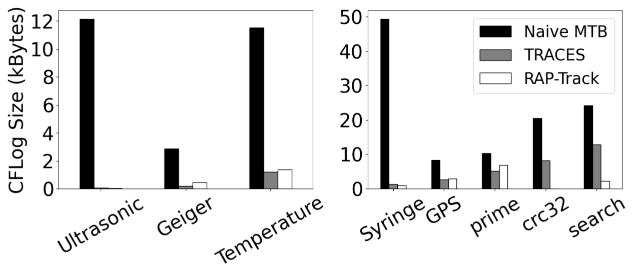


Fig. 9: $\mathsf{CF_{Log}}$ Size Comparison

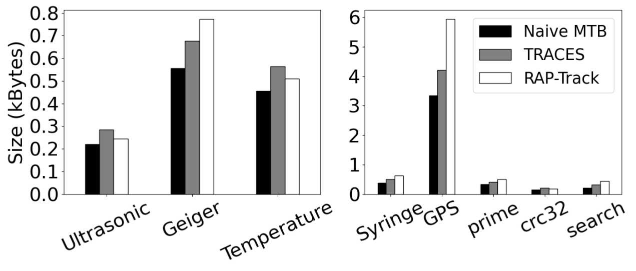


Fig. 10: Code size comparison

V. Implementation & Evaluation

We implemented RAP-Track proof-of-concept prototype (publicly available at [32]) on the V2M-MPS2-0318C prototyping system [31], which is equipped with an AN505 Arm Cortex-M33 FPGA image [56] based on the ARM Cortex-M33 (v8) architecture with support to TZ, MTB, and DWT extensions. RAP-Track Secure World code occupies 11 kilobytes (KB) in total with the CFA Engine occupying 2.8 KB. RAP-Track static analysis/linking code used in the Offline Phase is written in Python and operates directly on post-compiled binaries. We evaluate RAP-Track's memory and runtime overhead on real MCU applications (the same open-source applications listed in Section I). These applications are the same used in prior work, enabling direct comparison. RAP-Track performance is compared to TRACES [24]: a TEE-based CFA architecture implementing state-of-the-art $\mathsf{CF_{Log}}$ optimizations. We note that, while other CFA architectures (e.g., [17], [19]) also implement optimizations similar to TRACES, TRACES open-source availability enables our comparison (whereas we are unable to compare to closed source CFA architectures). In addition to TRACES, as a baseline for runtime, we consider unmodified APP-s without any CFA-enabling instrumentation and associated overheads. For the baseline on $\mathsf{CF_{Log}}$ size and code size, we use the *naive* MTB-based logging discussed in Section I.

A. Runtime Overhead

The runtime overhead of instrumentation-based CFA methods is primarily due to three factors: trampolines that invoke the CFA Engine API, context switches between the Non-Secure and Secure Worlds, and the logging algorithm (along with optimizations). In contrast, RAP-Track's overhead mainly comes from trampoline instructions for branching between MTBAR and MTBDR, with the only context switches required being to log loop conditions. The *naive* MTB approach incurs no overhead as it lacks any instrumentation. Figure 8 shows the CPU cycles required by each method in the sample APP-s. RAP-Track adds from 2% to 62% in CPU cycles over the *naive* MTB approach, whereas TRACES overhead ranges from 7% to 1309%. Differences across APP-s are due to the varying frequency of control flow transfers in their implementations.

B. $\mathsf{CF_{Log}}$ *Size*

The size of $\mathsf{CF_{Log}}$-s generated by each method is shown in Figure 9. $\mathsf{CF_{Log}}$ sizes of RAP-Track and TRACES across APP-s vary depending on the optimizations and execution paths taken in each case. For instance, RAP-Track only records the branch taken in if-else statements, so the frequency of each path directly impacts $\mathsf{CF_{Log}}$ size. Additionally, loop optimizations can substantially reduce $\mathsf{CF_{Log}}$ size, as demonstrated in the results from the Ultrasonic Sensor and Syringe applications, highlighting that RAP-Track produces significantly smaller $\mathsf{CF_{Log}}$-s than *naive* MTB. The differences in $\mathsf{CF_{Log}}$ size between RAP-Track and TRACES depend on the specific application and optimizations used. It is also possible to implement instrumentation-based CFA that records the exact branches tracked by RAP-Track. Results on *prime* and *gps* applications (from the BEEBs benchmark [30]) show that RAP-Track achieves considerably better runtime while producing similarly sized $\mathsf{CF_{Log}}$-s. We also note that in lossless CFA, Prv must pause APP to send partial reports when $\mathsf{CF_{Log}}$ assigned memory reaches capacity. Thus, $\mathsf{CF_{Log}}$ size directly impacts communication overhead/latency, often becoming the system's primary bottleneck (see [57]). For example, the ARM-M33 MTB used in our prototype has a 4KB limit, which would cause applications that use a *naive* MTB approach to frequently pause for partial report transmissions in most cases. In contrast, RAP-Track can fit all transfers within 4KB in most of the tested applications, only requiring one transmission to Vrf (once APP execution completes) and thus avoiding pauses.

C. *Program Memory Overhead*

Figure 10 presents the code size of each application, showcasing the impact of instructions introduced by RAP-Track trampolines and the TRACES instrumentation. In most APP-s, RAP-Track incurs slightly more code size overhead, with a significant variation based on the density of indirect and conditional branches. This additional overhead is primarily due to RAP-Track requiring extra instructions in conditional loops compared to TRACES. Additionally, 'nop' instructions were added in MTBAR trampolines to allow the MTB sufficient time to activate before logging a branch, as the MTB hardware activation is not immediate upon entering the MTBAR.

VI. Conclusion

This paper presents RAP-Track, a TEE-based CFA mechanism to minimize instrumentation and context switches in "off-the-shelf" commodity MCUs. RAP-Track static linking organizes all non-deterministic branches into a monitored region to optimize MTB-logging for CFA by leveraging PC-based hardware controls from the DWT extension. RAP-Track offers considerably more efficient runtime while maintaining a similar $\mathsf{CF_{Log}}$ size as state-of-the-art methods. RAP-Track prototype is publicly available at [32].

Acknowledgments

We thank DAC anonymous reviewers for their feedback. Part of this work was conducted while the authors were at RIT.

References

[1] M. N. Nafees *et al.*, "Smart grid cyber-physical situational awareness of complex operational technology attacks: A review," *CSUR*, 2023.

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

[3] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *TISSEC*, 2012.

[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *AsiaCCS*, 2011.

[5] X. Tan and Z. Zhao, "Sherloc: Secure and holistic control-flow violation detection on embedded systems," in *CCS*. ACM, 2023.

[6] Y. Wang, C. L. Mack, X. Tan, N. Zhang, Z. Zhao, S. Baruah, and B. C. Ward, "Insectacide: Debugger-based holistic asynchronous cfi for embedded system," in *RTAS*. IEEE, 2024.

[7] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," *SIGPLAN Notices*, 2017.

[8] ARM, "BTI," https://developer.arm.com/documentation/ddi0602/2021-12/Base-Instructions/BTI--Branch-Target-Identification-, 2020, [Online; accessed 13-February-2023].

[9] ——, "Return Address Signing using ARM Pointer Authentication," https://gcc.gnu.org/legacy-ml/gcc-patches/2018-11/msg00104.html, 2018, [Online; accessed 13-February-2023].

[10] Microsoft, "Control Flow Guard for platform security," https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, 2022, [Online; accessed 13-February-2023].

[11] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing Embedded Systems with Return Address Integrity," in *NDSS*, 2020.

[12] M. Ammar, A. Caulfield, and I. D. O. Nunes, "Sok: Integrity, attestation, and auditing of program execution," in *S&P*. IEEE, 2025.

[13] 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*, 2017.

[14] S. Zeitouni *et al.*, "Atrium: Runtime attestation resilient under memory attacks," in *ICCAD*. IEEE, 2017.

[15] A. Caulfield, N. Rattanavipanon, and I. D. O. Nunes, "ACFA: Secure runtime auditing & guaranteed device healing via active control flow attestation," in *USENIX Security Symposium*, 2023.

[16] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: lightweight hardware-assisted attestation of program execution," in *ICCAD*. IEEE, 2018.

[17] 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 *ACM CCS*, 2016.

[18] A. J. Neto and I. D. O. Nunes, "Isc-flat: On the conflict between control flow attestation and real-time operations," in *RTAS*. IEEE, 2023.

[19] Z. Sun, B. Feng, L. Lu, and S. Jha, "Oat: Attesting operation integrity of embedded devices," in *S&P*. IEEE, 2020.

[20] 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 *USENIX Security Symposium*, 2023.

[21] N. Yadav and V. Ganapathy, "Whole-program control-flow path attestation," in *ACM CCS*, 2023.

[22] D. Papamartzivanos, S. A. Menesidou, P. Gouvas, and T. Giannetsos, "Towards efficient control-flow attestation with software-assisted multi-level execution tracing," in *MeditCom*. IEEE, 2021.

[23] A. Developer, *MTB-M33 Technical Reference Manual*, 2024.

[24] A. Caulfield *et al.*, "Traces: Tee-based runtime auditing for commodity embedded systems," 2024.

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

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

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

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

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

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

[31] ARM, *Versatile Express V2M-MPS2 motherboard*, 2024. [Online]. Available: https://www.keil.com/boards2/arm/v2m_mps2_iot_/#/eula-container

[32] A. J. Neto, A. Caulfield, and I. D. O. Nunes, "RAP-Track prototype repository," https://github.com/SPINS-RG/RAP-Track, 2025.

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

[34] ——, "Dwt functional descriptionl," 2024. [Online]. Available: https://developer.arm.com/documentation/ddi0439/b/Data-Watchpoint-and-Trace-Unit/DWT-functional-description?lang=en

[35] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New results for timing-based attestation," in *S&P*. IEEE, 2012.

[36] D. Schellekens, B. Wyseur, and B. Preneel, "Remote attestation on legacy operating systems with trusted platform modules," *Science of Computer Programming*, 2008.

[37] 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," *TOPS*, 2017.

[38] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified Hardware/Software Co-Design for remote attestation," in *USENIX Security Symposium*, 2019.

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

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

[41] K. Eldefrawy *et al.*, "HYDRA: hybrid design for remote attestation (using a formally verified microkernel)," in *WiSec*. ACM, 2017.

[42] M. Ammar *et al.*, "Delegated attestation: scalable remote attestation of commodity cps by blending proofs of execution with software attestation," in *ACM WiSec*, 2021, pp. 37–47.

[43] I. De Oliveira Nunes *et al.*, "Casu: Compromise avoidance via secure update for low-end embedded systems," in *IEEE/ACM ICCAD*, 2022, pp. 1–9.

[44] ——, "Parsel: Towards a verified root-of-trust over sel4," in *IEEE/ACM ICCAD)*. IEEE, 2023.

[45] I. D. O. Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the toctou problem in remote attestation," *arXiv preprint arXiv:2005.03873*, 2020.

[46] O. Arias, D. Sullivan, H. Shan, and Y. Jin, "Lahel: Lightweight attestation hardening embedded devices using macrocells," in *HOST*. IEEE, 2020.

[47] M. Geden and K. Rasmussen, "Hardware-assisted remote runtime attestation for critical embedded systems," in *PST*. IEEE, 2019.

[48] Y. Zhang *et al.*, "ReCFA: resilient control-flow attestation," in *ACSAC*, 2021.

[49] F. Toffalini *et al.*, "ScaRR: Scalable runtime remote attestation for complex systems," in *RAID*, 2019.

[50] I. D. O. Nunes *et al.*, "Tiny-CFA: Minimalistic control-flow attestation using verified proofs of execution," in *DATE*. IEEE, 2021.

[51] I. D. O. Nunes, S. Jakkamsetti, and G. Tsudik, "Dialed: Data integrity attestation for low-end embedded devices," in *DAC*. IEEE, 2021.

[52] ARM, "Coresight system trace macrocell technical reference manual," 2024. [Online]. Available: https://developer.arm.com/documentation/ddi0444/latest/

[53] J. Obermaier *et al.*, "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, 2018.

[54] A. J. Neto *et al.*, "Pearts: Provable execution in real-time embedded systems," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 47–47.

[55] A. Caulfield *et al.*, "Asap: reconciling asynchronous real-time operations and proofs of execution in simple embedded systems," in *DAC*, 2022, pp. 721–726.

[56] ARM, *Cortex™-M33 with IoT kit FPGA for MPS2+ Version 2.0 (AN505)*, 2024. [Online]. Available: https://developer.arm.com/downloads/view/AN505?sortBy=availableBy&revision=r0p0-00rel0

[57] A. Caulfield *et al.*, "Speccfa: Enhancing control flow attestation/auditing via application-aware sub-path speculation," *ACSAC*, 2024.