# CARAMEL: Boosting Device Utilization in Control Flow Auditing

Alexandra Lengert
University of Zurich
Zurich, Switzerland
alexandra.lengert@uzh.ch

Adam Ilyas Caulfield
University of Waterloo
Waterloo, Ontario, CA
acaulfield@uwaterloo.ca

Ivan De Oliveira Nunes
University of Zurich
Zurich, Switzerland
ivan.deoliveiranunes@uzh.ch

## Abstract

Micro-Controller Units (MCUs) are widely used in safety-critical systems, making them attractive targets for attacks. This calls for lightweight defenses that remain effective despite software compromise. Control Flow Auditing (*CF-Aud*) is one such mechanism wherein a remote Verifier ($\mathcal{V}$rf) is guaranteed to receive evidence about the control flow path taken on a Prover MCU ($\mathcal{P}$rv), even when $\mathcal{P}$rv software is compromised. Despite promising benefits, current *CF-Aud* architectures unfortunately require a "busy-wait" phase where a hardware-anchored root-of-trust (RoT) in $\mathcal{P}$rv retains execution control to ensure delivery of control flow evidence to $\mathcal{V}$rf. This drastically reduces CPU utilization on $\mathcal{P}$rv.

In this work, we address this limitation with an architecture for Contention Avoidance in Runtime Auditing with Minimized Execution Latency (*CARAMEL*). *CARAMEL* is a hardware-software RoT co-design that enables $\mathcal{P}$rv applications to resume while control flow evidence is transmitted to $\mathcal{V}$rf. This significantly reduces contention due to transmission delays and improves CPU utilization without giving up on security. Key to *CARAMEL* is our design of a new RoT with a self-contained (and minimal) dedicated communication interface. *CARAMEL*'s implementation and accompanying evaluation are made open-source. Our results show substantially improved CPU utilization at modest hardware cost.

**ACM Reference Format:**
Alexandra Lengert, Adam Ilyas Caulfield, and Ivan De Oliveira Nunes. 2026. CARAMEL: Boosting Device Utilization in Control Flow Auditing. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3804209

## 1 Introduction

The use of embedded devices (a.k.a., IoT or "smart" devices) is rapidly increasing. They are crucial parts of modern systems with various tasks, including safety-critical operations [4, 39, 40]. Embedded devices are commonly implemented using Micro-Controller Units (MCUs) designed for energy efficiency and low cost, making them useful for lengthy remote deployments. However, these resource constraints lead to limited security features.

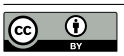
In this landscape, Remote Attestation (*RA*) [31] offers an inexpensive security mechanism to remotely assess software integrity. *RA* is a challenge-response protocol in which a Verifier ($\mathcal{V}$rf) challenges a remotely deployed Prover MCU ($\mathcal{P}$rv) to provide cryptographic proof of their currently installed code. By inspecting the proof, $\mathcal{V}$rf can remotely decide on $\mathcal{P}$rv's code integrity. Yet, in standard *RA*, $\mathcal{V}$rf remains oblivious to attacks that do not modify code [1]. For example, an adversary ($\mathcal{A}$dv) can launch control flow hijacking or code-reuse attacks [9, 36], corrupting branch instructions (e.g., calls, jumps, returns) to execute unintended functions, skip security-critical code, or other arbitrary (malicious) behavior without detection. Control Flow Integrity (CFI) mechanisms [10] offer a local countermeasure, but often they neither verify complete control flow paths nor produce authenticated path evidence, preventing $\mathcal{V}$rf from subsequent examination to determine attack root causes.

Control Flow Attestation (*CF-Att*) [6] extends standard *RA* to provide $\mathcal{V}$rf with authenticated control flow path evidence. In *CF-Att*, a runtime trace of the control flow path ($CF_{Log}$) taken during the attested program's execution is also included in the cryptographic proof sent to $\mathcal{V}$rf. Upon receiving $CF_{Log}$, $\mathcal{V}$rf can inspect it to determine whether the program was executed correctly or if remedial actions are required.

Most previous *CF-Att* are best-effort, assuming that $CF_{Log}$ would reach $\mathcal{V}$rf, as a persistent lack of a response to $\mathcal{V}$rf requests for evidence suggests an issue (e.g., a compromised state) with $\mathcal{P}$rv. However, $\mathcal{A}$dv may deliberately not send $CF_{Log}$ to conceal the control flow path that led to the compromise, revealing an anomalous state, but preventing analysis of $CF_{Log}$ for vulnerability (root cause) identification. To address this, Control Flow Auditing (*CF-Aud*) [12, 14] was proposed to ensure reliable delivery of $CF_{Log}$. Additionally, a *CF-Aud* RoT can guarantee the ability to take corrective action when an invalid execution path is detected.

*CF-Aud* obtains its guarantees by actively triggering a (hardware-supported) RoT to generate and transmit runtime evidence (containing $CF_{Log}$) to $\mathcal{V}$rf upon key events at runtime (e.g., when $\mathcal{P}$rv storage for $CF_{Log}$ is full, requiring transmission before adding new entries). Current *CF-Aud* RoTs "busy-wait" in a secure state after evidence transmission until receiving an acknowledgment from $\mathcal{V}$rf. This ensures that a compromised $\mathcal{P}$rv cannot tamper, delete, or replace $CF_{Log}$ before it has reached $\mathcal{V}$rf. Clearly, this approach diminishes $\mathcal{P}$rv utilization. Indeed, prior work [12] acknowledged trade-offs between "best-effort" *CF-Att* vs. allowing untrusted execution to resume after a maximum waiting period in *CF-Aud*. Yet, it is thus far unclear how $CF_{Log}$ transmission and untrusted code resumption can be achieved simultaneously.

This work tackles the aforementioned challenge with *CARAMEL*: an architecture for Contention Avoidance in Runtime Auditing with Minimized Execution Latency. *CARAMEL*'s fundamental shift from prior RoT models lies in a dedicated and RoT-integrated trusted communication interface. Notably, this integration requires addressing non-trivial challenges. In particular, $CF_{Log}$'s joint memory+transmission management requires the creation of partial authenticated report units (denoted $CF_{Slice}$). This ensures that new entries, added to $CF_{Log}$ by the resumed application, can still be




*DAC '26, Long Beach, CA, USA*

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3804209

securely stored as a new $CF_{Slice}$ while prior reports/responses to/from $\mathcal{V}$rf are transmitted, thus avoiding contention due to lack of storage for $CF_{Log}$. In sum, our anticipated contributions are:

- *CARAMEL* Design: we design a hardware-software RoT architecture for *CF-Aud* that maintains security and boosts $\mathcal{P}$rv utilization during *CF-Aud* cycles. *CARAMEL*'s low-cost hardware records attested program control flow paths, triggers generation and parallel transmission of reports, and handles reliable communication with $\mathcal{V}$rf. Its software is responsible for cryptographic operations, which are relatively expensive to implement in hardware.
- *CARAMEL* Implementation and Evaluation: we analyze the cost and security of *CARAMEL*. As a case study, we implement a working prototype deployed on the Basys3 FPGA prototyping board (available at [24]). Evaluating end-to-end runtime auditing instances of real-world sensor applications [35, 37, 38, 44] confirms *CARAMEL* increases utilization with low hardware cost.

## 2 Background: Attestation & Auditing

Runtime Attestation methods, including *CF-Att* [1, 8, 12, 14, 17, 18, 26, 29, 42, 43, 47–49], are concerned with remotely verifying runtime properties beyond the presence of the correct code in $\mathcal{P}$rv's program memory. *CF-Att* typically involves the following steps:

(1) $\mathcal{V}$rf requests *CF-Att* from $\mathcal{P}$rv by sending an attestation challenge (*Chal*);
(2) After receiving *Chal*, $\mathcal{P}$rv executes its software while an RoT on $\mathcal{P}$rv records an execution trace ($T$);
(3) $\mathcal{P}$rv's RoT computes an authenticated measurement ($\sigma$) over $\mathcal{P}$rv's program memory, $T$, and *Chal* to produce report $H = [\sigma, T]$, which it then sends to $\mathcal{V}$rf;
(4) Upon receiving $H$, $\mathcal{V}$rf validates it by comparing $\sigma$ to its expected value and examining $T$ to determine if $T$ represents expected program execution properties.

In *CF-Att* and *CF-Aud*, $T$ corresponds to $CF_{Log}$. The authenticated measurement ($\sigma$) computed in Step 3 is typically implemented as a Message Authentication Code (MAC) or digital signature. The secret key used for the measurement is kept inaccessible to all software on $\mathcal{P}$rv, except for the isolated RoT implementation, typically requiring hardware support. Current architectures build $CF_{Log}$ by either (1) instrumenting the program binary with calls to a logging software inside a Trusted Execution Environment (TEE) or (2) custom hardware modules that detect and log control flow transfers.

Early *CF-Att* techniques [1, 17, 48] send $\mathcal{V}$rf a single hash digest produced by computing a hash-chain of control flow transfers, yielding a fixed-sized representation of the program path. Since deriving $CF_{Log}$ from the hash becomes infeasible as the number of control flow transfers grows [34], most of the recent *CF-Att* techniques transmit $CF_{Log}$ *verbatim* to $\mathcal{V}$rf [12, 14, 18, 29, 42, 43, 46, 47, 49]. Transmission and storage costs of $CF_{Logs}$, can be reduced by compressing data written to $CF_{Log}$ without loss of information [12–14, 18, 42, 46, 47, 49] or by transmitting partial $CF_{Log}$ slices when dedicated $CF_{Log}$ storage has reached capacity [12, 14, 18, 43].

*CF-Att* mechanisms were previously limited by their *passive* nature, relying on a compromised $\mathcal{P}$rv to participate in a *CF-Att* protocol and send the runtime evidence. However, there is no guarantee that $\mathcal{A}$dv who has full control over $\mathcal{P}$rv software will allow $CF_{Log}$ to be sent. The latter prevents $\mathcal{V}$rf from (1) receiving evidence of the exact path deviation, (2) learning any resulting consequences, or (3) remediating attack root causes.

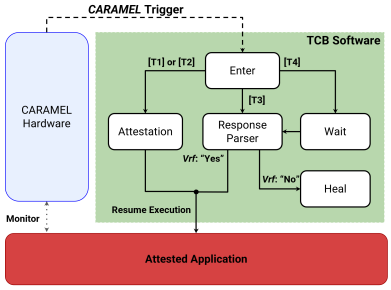



**Figure 1: *CARAMEL* high-level workflow**

*CF-Aud* architectures [12, 14] incorporate mechanisms to actively interrupt the attested application's execution to transmit $CF_{Log}$ reliably upon significant runtime events (e.g., storage full, timeout, execution completed). While waiting for $\mathcal{V}$rf confirmation of evidence receipt, current *CF-Aud* RoTs retain execution control of $\mathcal{P}$rv, ensuring that application execution only resumes once $\mathcal{V}$rf confirms its trust in the evidence sent by $\mathcal{P}$rv. However, as noted in Sec. 1, this results in loss of $\mathcal{P}$rv utilization, especially for longer programs that require several partial $CF_{Log}$ transmissions.

## 3 *CARAMEL* Overview

*CARAMEL* avoids $CF_{Log}$ contention in *CF-Aud* by splitting $CF_{Log}$ storage into two $CF_{Slices}$. During runtime, *CARAMEL* hardware detects when the first $CF_{Slice}$ is ready for transmission and initiates transmission via an RoT-integrated communication interface. *CARAMEL* allows the application's execution to resume in parallel (which in turn fills the second $CF_{Slice}$) while waiting for a response from $\mathcal{V}$rf. It generates a secure (i.e., non-maskable) interrupt when a message is received through the RoT communication interface to begin its processing. *CARAMEL* software Trusted Computing Base (TCB) is only required to (1) compute/validate authentication tokens and (2) execute $\mathcal{V}$rf-specified remediation action(s), in case of a compromise. The complete workflow is shown in Fig. 1.

During execution, *CARAMEL* actively triggers its TCB via a custom non-maskable interrupt upon the following events:
**[T1]** the audited application concludes its execution;
**[T2]** a $CF_{Slice}$ is full and ready for transmission to $\mathcal{V}$rf;
**[T3]** a message is received by the RoT communication interface;
**[T4]** $CF_{Log}$ storage (both $CF_{Slices}$) fills up while no $\mathcal{V}$rf response is received. In this case, *CARAMEL* must wait before reusing $CF_{Log}$.

Per Fig. 1, *CARAMEL* TCB's execution path depends on the trigger source. When triggered by **[T1]** or **[T2]**, an attestation report must be created. After the report is ready, the TCB exits to resume the application execution. *CARAMEL* hardware ensures new control flow transfers do not overwrite any $CF_{Slice}$ that is pending verification by $\mathcal{V}$rf (further details discussed in Sec. 4.5).

When triggered by **[T3]**, *CARAMEL* has received a message through its communication interface, and thus it authenticates and parses the message. If $\mathcal{V}$rf approved the previous report, the TCB exits and resumes the application, making the previous $CF_{Slice}$ memory available for reuse. Alternatively, a $\mathcal{V}$rf-configurable remediation is triggered if the report indicates a compromise.

If TCB is triggered by **[T4]**, *CARAMEL* has not received a timely response from $\mathcal{V}$rf, both $CF_{Slices}$ have been used, and it must enter

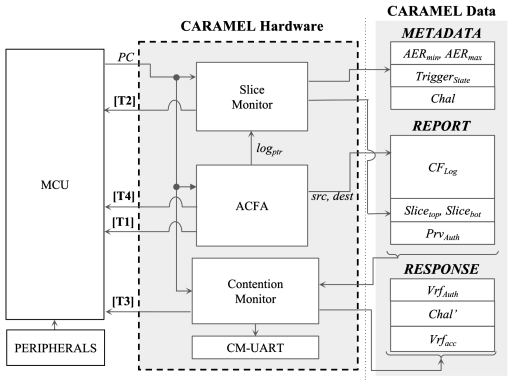



**Figure 2: *CARAMEL* system architecture**

a secure "busy-wait" state as a last resort, falling back to the same mechanism used by prior *CF-Aud* architectures by default [12, 14]. The TCB stays in this state until a message is received from $\mathcal{V}$rf approving the pending $CF_{Slice}$.

# 4 *CARAMEL* Design Details

## 4.1 System & Adversary Models

We focus on resource-constrained MCUs (e.g., AVR ATmega [22] and TI MSP430 [23] families). MCUs within this class are typically equipped with 8- or 16-bit cores that run on low clock frequencies (1-16 MHz). Due to a lack of MMUs or virtualization support, they run software at "bare metal" (i.e., without virtual-to-physical address translations). *CARAMEL* prototype is built on the openMSP430 MCU implemented in Verilog [21]. We chose openMSP430 due to its public availability, though *CARAMEL* design is equally applicable to other MCUs.

We consider that $\mathcal{A}$dv can fully compromise any software on $\mathcal{P}$rv that is not explicitly protected by trusted hardware. $\mathcal{A}$dv can exploit vulnerabilities in $\mathcal{P}$rv software to inject code [20], hijack the application's control flow [25], or perform code-reuse attacks [9, 36]. Additionally, $\mathcal{A}$dv can manipulate interrupts and their associated routines, if unprotected. $\mathcal{A}$dv may also discard, inject, or attempt to modify messages in the network between $\mathcal{P}$rv and $\mathcal{V}$rf. In line with prior works [1, 5, 12, 15, 18, 30, 42], physical attacks are out of scope as they require orthogonal measures [32].

## 4.2 *CARAMEL* Architecture

*CARAMEL* interfaces with the MCU core and monitors some of its signals, as depicted in Fig. 2. MCU memory consists of program (PMEM) and data memory (DMEM). As shown in Fig. 1, PMEM is divided between *CARAMEL*'s hardware-protected TCB code and untrusted software (*S*) containing applications, including an application to be audited. The PMEM section used to store the audited application is called Audited Executable Region (*AER*). *AER* boundaries can be configured by $\mathcal{V}$rf if specified as part of the initial *CF-Aud* request. *CARAMEL* data in DMEM is used to exchange data between *CARAMEL*'s hardware and TCB (more details in Sec. 4.5). The rest of DMEM is available for *S* allocation.

*CARAMEL* hardware consists of three main modules: the underlying auditing architecture *ACFA* [12], the Slice Monitor, and the Contention Monitor. *CARAMEL* modifies *ACFA* for compatibility with other *CARAMEL* components (described further in Sec. 4.5). *CARAMEL* uses *ACFA* sub-modules to:

- generate **[T1]** and **[T4]** to trigger execution of *CARAMEL*'s TCB;
- protect *CARAMEL*'s TCB code and its execution from direct tampering by *S*;
- detect control flow transfers during *AER*'s execution and log them into $CF_{Log}$.

*CARAMEL*'s Slice Monitor monitors $CF_{Log}$ and sets **[T2]** trigger when a $CF_{Slice}$ is full (i.e., a partial report should be generated for $\mathcal{V}$rf). The Contention Monitor checks for when a partial *CF-Aud* report is ready to be transmitted via its dedicated communication interface: the Contention Monitor UART (CM-UART). The Contention Monitor also receives responses from $\mathcal{V}$rf and sets **[T3]** to trigger response parsing (described further in Sec. 4.5.)

## 4.3 TCB Software

*CARAMEL*'s TCB implements five procedures, as shown in Fig. 1. The `Enter` procedure determines the execution path based on the trigger source. The `Heal` procedure is configurable by $\mathcal{V}$rf based on the appropriate remediation for the deployment context (some examples include shut down, reset, erase DMEM, or software update). The `Wait` procedure is implemented as a "busy-wait" loop. The `Attestation` and `Response Parser` procedures incorporate components from VRASED [27] formally verified RA architecture. VRASED routines are used to produce authenticated memory measurements and to authenticate $\mathcal{V}$rf messages. This is because ACFA itself is built upon VRASED.

The TCB exit context affects *CARAMEL* hardware behavior:

(1) if exiting after `Response Parser` procedure, an authenticated $\mathcal{V}$rf message was received and $\mathcal{V}$rf approved the previous report (thus, *CARAMEL* hardware can now overwrite/reuse the memory corresponding to the accepted $CF_{Slice}$);
(2) if exiting after the `Attestation` procedure, an authentication token for the current report has been fully produced. Thus, hardware should start transmitting the authenticated report.

For *CARAMEL* hardware to be aware of these conditions, the TCB redirects its execution to two corresponding trampoline instructions at the following fixed addresses:

(1) $accepted_{addr}$: visited when $\mathcal{V}$rf was authenticated and $\mathcal{V}$rf accepted the previous report;
(2) $send_{addr}$: visited when the authentication token has been generated and is ready for transmission.

*CARAMEL* hardware monitors the program counter (*PC*). PC contains the address of the currently executing instruction. Based on *PC*, the Slice Monitor detects visits to $accepted_{addr}$ to release the prior $CF_{Slice}$ memory for reuse. The Contention Monitor begins transmitting data for the report while the authentication token is computed, and waits for visits to $send_{addr}$ to continue transmission of the corresponding token. Sec. 4.5 discusses further details on the exit conditions checked by *CARAMEL* hardware.

## 4.4 *CARAMEL* Data

The dedicated memory for *CARAMEL* data contains three subdivisions: *METADATA*, *REPORT*, and *RESPONSE*, as depicted in Fig. 2.

The ***METADATA*** region contains data that is used internally by *CARAMEL*. It has the following components:

**Definition:** Wrapping increment

$$\texttt{incr}(base, \delta) := base + \delta \mod CF_{size}$$

**Definition:** Modified definition of $log_{full}$:

$$log_{full} := (\texttt{incr}(Log_{ptr}, 2) = Slice_{top})$$

**HW Specification:** TCB Trigger **[T1]** and **[T4]**

$$(PC = AER_{max}) \rightarrow [T1], log_{full} \rightarrow [T4]$$

**Figure 3: Increment Definition and Modification to *ACFA***

- $AER_{min}$, $AER_{max}$: store the bounds of *AER*, specified by its minimum and maximum memory address, respectively;
- $Trigger_{state}$: stores the trigger source that caused the latest TCB to execution;
- *Chal*: stores the latest challenge received from $\mathcal{V}$rf.

The ***REPORT*** region contains all data to be eventually sent to $\mathcal{V}$rf alongside the metadata in audited execution report(s):

- $CF_{Log}$: the memory region for storing control flow transfers;
- $Slice_{top}$, $Slice_{bot}$: the bounds of the $CF_{Slices}$ in $CF_{Log}$ that should be used for the auditing report, denoting the top (first address) and bottom (last address) of a $CF_{Slice}$, respectively;
- $\mathcal{P}\text{rv}_{Auth}$: memory for storing the report authentication token computed by the TCB's `Attestation` procedure.

The ***RESPONSE*** region contains the data that should be received as a part of $\mathcal{V}$rf's response. Within it are the following:

- $\mathcal{V}\text{rf}_{Auth}$: token sent by $\mathcal{V}$rf to authenticate the message;
- *Chal'*: fresh attestation challenge for the next report;
- $\mathcal{V}\text{rf}_{check}$: byte denoting whether $\mathcal{V}$rf accepted ($\mathcal{V}\text{rf}_{check} = 1$) or rejected ($\mathcal{V}\text{rf}_{check} = 0$) the report.

## 4.5 Specification of *CARAMEL* Components

This subsection defines the hardware specifications for *CARAMEL*'s internal sub-modules depicted in Fig. 2. Several variables that reference $CF_{Log}$ bounds are incremented to wrap around rather than overflowing the size of $CF_{Log}$. We describe this by defining the incrementing function $\texttt{incr}(\cdot)$ in Fig. 3.

***ACFA Module.*** *ACFA*'s hardware [12] interfaces directly with *CARAMEL*'s Slice Monitor, writes to *REPORT*, and outputs **[T1]** and **[T4]**. **[T1]** is set when *PC* reaches the last instruction of *AER* ($AER_{max}$) **[T4]** originates from *ACFA*'s signal $log_{full}$. *ACFA* maintains a signal ($Log_{ptr}$) that tracks the total data added to $CF_{Log}$ thus far. In original *ACFA*, $log_{full}$ was set when the next $Log_{ptr}$ value ($Log_{ptr}$ +2) equaled $CF_{Log}$'s max size ($CF_{size}$). In *CARAMEL*, $Log_{ptr}$ is defined to wrap when reaching $CF_{size}$, and $CF_{Log}$ is considered full when no $CF_{Slice}$ is available. Hence, we modify this definition to set $log_{full}$ when $Log_{ptr}$ +2 equals $Slice_{top}$, as shown in Fig. 3.

***Slice Monitor.*** The Slice Monitor has two main roles:

(1) sets **[T2]** when TCB should generate a partial report;
(2) tracks which $CF_{Slice}$ is used for the attestation report by updating $Slice_{top}$ and $Slice_{bot}$.

The Slice Monitor (see hardware specification in Fig. 4) maintains bit vector $Trigger_{state}$ to keep track of the current active trigger, as mentioned in Sec. 4.4.

The remaining specification relates to monitoring the $CF_{Slice}$ that is in use for logging and transmission, via three pointers:

(1) $Slice_{top}$, pointing to the top limit of the $CF_{Slice}$ that should be used in the next report;

**HW Specification:** Maintain trigger enumeration ($Trigger_{state}$)

$$Trigger_{state} := select\{[T1], [T2], [T3], [T4]\}$$

**HW Specification:** The current $CF_{Slice}$ is full

$$(Log_{ptr} = bound_{low}) \rightarrow slice_{full}$$

**HW Specification:** Track if $\mathcal{V}$rf accepted the previous report ($Vrf_{acc}$)

$$Vrf_{acc} := \begin{cases} 1 & \text{if } PC = accepted_{addr} \\ 0 & \text{if } [T2] \\ Vrf_{acc} & \text{otherwise} \end{cases}$$

**HW Specification:** TCB Trigger [T2]

$$slice_{full} \wedge Vrf_{acc} \rightarrow [T2]$$

**HW Specification:** Update pointer to $CF_{Slice}$ top ($Slice_{top}$)

$$(PC = accepted_{addr}) \rightarrow (Slice_{top} := \texttt{incr}(Slice_{top}, Slice_{size}))$$

**HW Specification:** Update internal monitor of $CF_{Slice}$ limit ($bound_{low}$)

$$slice_{full} \wedge \neg \log_{full} \rightarrow (bound_{low} := \texttt{incr}(bound_{low}, Slice_{size}))$$

**HW Specification:** Update pointer to $CF_{Slice}$ bottom ($Slice_{top}$)

$$[T1] \wedge Vrf_{acc} \rightarrow (Slice_{bot} := Log_{ptr})$$
$$[T2] \rightarrow (Slice_{bot} := bound_{low})$$

**Figure 4: Slice Monitor hardware specification**

(2) $Slice_{bot}$, pointing to the bottom of the $CF_{Slice}$ that should be used in the next report.
(3) $bound_{low}$, pointing to the bottom limit of the $CF_{Slice}$ that is currently being written to by *CARAMEL* hardware;

The pointers ($Slice_{top}$,$Slice_{bot}$) are in memory and read by TCB software (as shown in Fig. 2), whereas $bound_{low}$ is used internally. To track when the limit of the current $CF_{Slice}$ has been reached, Slice Monitor sets $slice_{full}$ when $Log_{ptr}$ reaches $bound_{low}$.

$Vrf_{acc}$ represents whether the previous report is pending ($Vrf_{acc}$ = 0) or accepted ($Vrf_{acc}$ = 1). The Slice Monitor clears $Vrf_{acc}$ when **[T2]** has been set (i.e., a new report was generated). It sets $Vrf_{acc}$ when $PC = accepted_{addr}$, signaling that *CARAMEL*'s TCB has received and authenticated $\mathcal{V}$rf's response (recall Sec. 4.3). When both $slice_{full}$ and $Vrf_{acc}$ are set, the Slice Monitor sets **[T2]**.

The Slice Monitor updates the $CF_{Slice}$ boundary points based on the previously described signals. $Slice_{top}$ alternates between the tops of the two $CF_{Log}$ halves (i.e., either points to index 0 or $Slice_{size}$) when $\mathcal{V}$rf has accepted the report (i.e., $PC = accepted_{addr}$).

The Slice Monitor updates $bound_{low}$ to point to the bottom of the next slice when $slice_{full}$ and $\neg log_{full}$ (i.e., when the current $CF_{Slice}$ is full, but the next one is still available for use).

$Slice_{bot}$ is updated upon the **[T1]** and **[T2]** triggers to ensure the next report references the latest $CF_{Slice}$. When **[T1]** has occurred and $\mathcal{V}\text{rf}_{acc}$ is set, $Slice_{bot}$ is set to $Log_{ptr}$, to account for **[T1]** occurring in the middle of $CF_{Slice}$ boundaries. When **[T2]** has occurred, $Slice_{bot}$ is set to $bound_{low}$.

***Contention Monitor.*** The Contention Monitor is responsible for controlling all communication with $\mathcal{V}$rf. It reads from *REPORT* and writes to *RESPONSE*, which triggers **[T3]**. It transmits the report in parallel to software execution and is configured to send new bytes upon a pulse based on the CM-UART's baud rate ($trigger_{tx}$).

Fig. 5 shows the specification for the Contention Monitor to track which triggers require a transmission, to iterate through *REPORT* memory for transmission, and to pass bytes from *REPORT* to the CM-UART transmit buffer. It is synchronized on $trigger_{tx}$ and implemented with the following internal signals and registers:

**HW Specification:** Register report transmission due to **[T2]** and **[T1]**

$$[T2] \rightarrow t2_{pend}, [T1] \rightarrow t1_{pend}$$
$$t2_{pend} \wedge (read_{idx} = REPORT_{size}) \rightarrow \neg t2_{pend}$$
$$t1_{pend} \wedge \neg t2_{pend} \wedge (read_{idx} = REPORT_{size}) \rightarrow \neg t1_{pend}$$
$$t1_{pend} \vee t2_{pend} \rightarrow tx_{pend}$$

**HW Specification:** Control transmission of $CF_{Log}$

$$tx_{pend} \wedge \neg start_{CF_{Log}} \wedge \mathcal{V}\mathrm{rf}_{acc} \rightarrow start_{CF_{Log}} \wedge (read_{idx} := Slice_{top})$$
$$start_{CF_{Log}} \wedge (read_{idx} \neq REPORT_{size}) \rightarrow (read_{idx} := \mathtt{incr}(read_{idx}, 1))$$
$$start_{CF_{Log}} \wedge (read_{idx} = Slice_{bot}) \rightarrow finish_{CF_{Log}} \wedge \neg start_{CF_{Log}}$$

**HW Specification:** Control transmission of remaining report data

$$finish_{CF_{Log}} \wedge (PC = send_{addr}) \rightarrow start_{rem} \wedge (read_{idx} := CF_{size})$$
$$start_{rem} \wedge (read_{idx} \neq REPORT_{size}) \rightarrow (read_{idx} := read_{idx} + 1)$$
$$(read_{idx} = REPORT_{size}) \rightarrow \neg start_{rem} \wedge \neg finish$$
$$data_{TX} := REPORT[read_{idx}]$$

**Figure 5: Contention Monitor transmission control**

- $(t1_{pend}, t2_{pend})$: registers corresponding to whether a transmission is required due to trigger **[T1]** or **[T2]**, respectively;
- $read_{idx}$: an index to read from *REPORT*;
- $start_{CF_{Log}}$: flag to start sending data from $CF_{Log}$;
- $finsh_{CF_{Log}}$: flag denoting finished $CF_{Log}$ data transmission;
- $start_{rem}$: flag to start sending the remaining *REPORT* data;
- $data_{TX}$: the transmission buffer of CM-UART.

The Contention Monitor begins transmitting after $tx_{pend}$, which is the accumulation of $t1_{pend}$ and $t2_{pend}$. These registers are not cleared until their corresponding reports have been sent. Since **[T1]** may occur during transmission of **[T2]** report, clearing of $t1_{pend}$ is also conditioned on $t2_{pend}$ already being cleared.

The relevant $CF_{Slice}$ (s) are transmitted during computation of $\mathcal{P}\mathrm{rv}_{auth}$ to improve performance. The Contention Monitor initializes this by setting $start_{CF_{Log}}$ when:

- a transmission has been registered ($tx_{pend}$ is set);
- $\mathcal{V}$rf has accepted the previous report ($\mathcal{V}\mathrm{rf}_{acc}$ is set);
- and there is no ongoing transmission ($start_{CF_{Log}}$ is cleared);

Additionally, at this moment, the Contention Monitor sets $read_{idx}$ to start reading from $Slice_{top}$. The Contention Monitor detects the completed transmission of $CF_{Slice}$ (s) when $read_{idx}$ equals $Slice_{bot}$, at which point $start_{CF_{Log}}$ is cleared and $finish_{CF_{Log}}$ is set.

The remaining *REPORT* data is transmitted when $finish_{CF_{Log}}$ is set and the $\mathcal{P}\mathrm{rv}_{auth}$ has been generated. The latter is detected when *PC* equals $send_{addr}$ (recall Sec. 4.3). When these two conditions occur, the Contention Monitor sets $start_{rem}$ to initiate transmission of remaining *REPORT* data, and it sets $read_{idx}$ to the starting point of $\mathcal{P}\mathrm{rv}_{auth}$ in *REPORT*: the base of the $CF_{Log}$ (depicted in Fig. 2).

To complete the transmission of a byte, the Contention Monitor indexes *REPORT* memory using $read_{idx}$ to obtain the next byte, and is passes it to the CM-UART data transmission buffer ($data_{TX}$).

The specification for the Contention Monitor to handle $\mathcal{V}$rf's response is outlined in Fig. 6. The CM-UART passes the following receive signals to the Contention Monitor:

(1) $trigger_{rx}$: set when the CM-UART has finished receiving a byte;
(2) $data_{RX}$: the received byte value itself.

The Contention Monitor maintains $write_{idx}$ as an internal write index for writing bytes to *RESPONSE* memory region. With each

**HW Specification:** Write received byte to *RESPONSE*

$$trigger_{rx} \rightarrow (RESPONSE[write_{idx}] := data_{RX}) \wedge (write_{idx} := write_{idx} + 1)$$

**HW Specification:** Handle TCB Trigger **[T3]**

$$(write_{idx} = RESP_{Size}) \rightarrow resp_{pend} \wedge (write_{idx} := 0)$$
$$\neg(write_{idx} = RESP_{Size}) \rightarrow \neg resp_{pend}$$

**Figure 6: Contention Monitor receive hardware specification**

received byte, *RESPONSE* region is updated and $write_{idx}$ is incremented. When $write_{idx}$ equals the *RESPONSE* size limit ($RESP_{Size}$), receiving is concluded, **[T3]** is set, and $write_{idx}$ is cleared.

## 5 Prototype & Evaluation

We synthesize *CARAMEL* hardware using the Xilinx Vivado toolset, implementing all *CARAMEL* sub-modules in Verilog according to the logic in Sec. 4. Our open-source prototype [24] is based on openMSP430 core [21]. We configure openMSP430 with 24 KB of PMEM and 30 KB of DMEM, retain its baseline GPIO, Timer, and UART peripherals, and add a second UART instance as the CM-UART operating at 115200 baud as part of *CARAMEL* 's RoT (Sec. 4.5). Within *CARAMEL* 's TCB, $\mathcal{V}$rf authentication and authenticated integrity measurement (see Sec. 2) employ a SHA-256–based HMAC from the formally verified HACL* library [50].

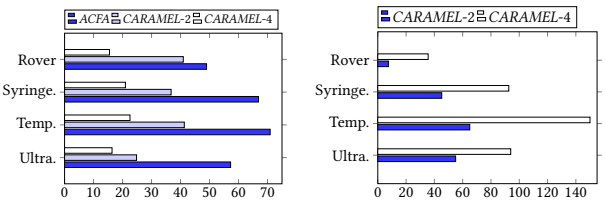
### 5.1 Performance and Cost Evaluation

***Memory Requirements.*** *CARAMEL*'s PMEM consists of the TCB software, which requires 5.3KB (4.5 KB for verified SHA-256-based HMAC). The remaining 830 bytes contain the CARAMEL workflow described in Sec. 3.*CARAMEL*'s DMEM size is determined by memory regions from Fig. 2: *METADATA*, *REPORT*, and *RESPONSE*. For our prototype, we use 32-byte attestation challenges and authentication tokens (i.e., 32-byte HMAC). As a result, *METADATA* uses 38 bytes, and *RESPONSE* uses 66 bytes. *CARAMEL*'s *REPORT* requires 4 bytes for $Slice_{top}$ and $Slice_{bot}$, 32 bytes for $\mathcal{P}\mathrm{rv}_{Auth}$, and the remaining size is based on the configuration of $CF_{size}$, resulting in $CF_{size}+36$ bytes for *REPORT*. Therefore, *CARAMEL*'s DMEM size is the sum of $CF_{size}$ and all mentioned data sizes (140 bytes).

***Runtime Costs.*** *CARAMEL* aims to reduce the added runtime of *CF-Aud* due to its busy-wait communication cycles. For evaluation, we measure the runtime of audited executions of real-world sensor applications, namely: an ultrasonic sensor application [38], a temperature sensor application [37], an automated medical syringe pump [7], and a motor control on a self-driving rover [35].

We choose these applications because they were also used to evaluate the prior work [1, 12, 14, 42], allowing direct comparison. These applications are deployed atop *CARAMEL*, *ACFA*, and best-effort *CF-Att*. The latter provides no delivery guarantees and thus no "busy wait" overheads are associated with it. We also report on two *CARAMEL* settings for $CF_{size}$: $CF_{size}$ of 2KB (*CARAMEL*-2, i.e., two slices of 1KB each) and 4KB (*CARAMEL*-4, i.e., two slices of 2KB each). *ACFA* is set with a $CF_{size}$ of 2KB. The network latency for $\mathcal{P}\mathrm{rv} \leftrightarrow \mathcal{V}\mathrm{rf}$ round-trip time (RTT) is set to 100 *ms* [19].

Fig. 7(a) presents the comparison of the runtime increase by the applications during audited execution. *ACFA* increases runtime by ≈48.9-70.9%, the highest compared to *CF-Att*, whereas *CARAMEL*-2 only increases runtime by ≈24.8-41.3%. With additional $CF_{size}$, *CARAMEL*-4 reduces this further to a ≈15.5-22.6% increase. The




(a) % Runtime increase vs. *CF-Att* (b) % Utilization increase vs. *ACFA*

**Figure 7: Execution Runtime/Utilization vs. State-of-the-Art.**

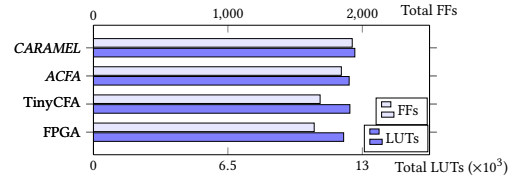



**Figure 8: Hardware costs compared to related works**

results indicate that *CARAMEL* reduces the runtime compared to *ACFA*, and can reduce it further with more memory for $CF_{Log}$.

To evaluate the impact on CPU utilization, we compare the time ratio between application and *CF-Aud* TCB code execution for both *CARAMEL* configurations. Fig. 7(b) shows the utilization compared to *ACFA*. *CARAMEL*-2 increases utilization across the selected applications by ≈7.4-65.0%. As shown with *CARAMEL*-4, a larger $CF_{size}$ gives an additional utilization increase for all applications. For *CARAMEL*-4 the increased utilization ranges from ≈35.6-150%.

***Hardware Cost.*** Like previous works [3, 11, 12, 28], we measure the hardware cost based on the number of Look-up Tables (LUTs) and flip-flops/registers (FFs), to indicate additional combinatorial logic cost and additional state, respectively. We present the cost of *CARAMEL* and related works when deployed on a fully-fledged MCU chip: CPU core plus peripherals (GPIO, UART, etc.).

The hardware cost results are shown in Fig. 8. In total, *CARAMEL* requires an additional 541 LUTs and 283 FFs, 4.5% LUTs and 17.2% FF over the unmodified openMSP430 core. Compared to ACFA, which provides *CF-Aud* with less CPU utilization, *CARAMEL* incurs an additional 2.2% LUTs and 4.3% FFs. This increase can be primarily attributed to the CM-UART, which accounts for 4.1% and 7.8% of LUTs and FFs, in an *CARAMEL*-equipped MCU. Compared to a Tiny-CFA-equipped MCU, *CARAMEL* adds 1.9% LUTs and 14.2% FFs. We note, however, that Tiny-CFA implements best effort delivery without guaranteeing communication between $\mathcal{V}$rf and $\mathcal{P}$rv.

## 5.2 Security Analysis

Per $\mathcal{A}$dv model in Sec. 4.1, $\mathcal{A}$dv can exploit software vulnerabilities on $\mathcal{P}$rv, including modifying unprotected code or data (e.g., control flow hijack through the stack modification) and attempting to disrupt *CARAMEL* protocol at any phase.

*CARAMEL* control flow logging is hardware-based, hence, it cannot be altered by a software $\mathcal{A}$dv. Thus, any control flow diversions caused by $\mathcal{A}$dv exploits will be reflected in $CF_{Log}$. Instead, $\mathcal{A}$dv may attempt to corrupt reports generated by *CARAMEL* (e.g. differ from the actual content of $CF_{Log}$). To achieve this, $\mathcal{A}$dv would need to (1) alter TCB's execution, (2) replace $CF_{Slice}$ or the authentication token ($Prv_{Auth}$ from Fig. 2), or (3) learn the secret key to forge reports. These options are prevented by *CARAMEL*'s inclusion of *ACFA* as a sub-module, inheriting hardware protection for TCB code, $CF_{Log}$, and keys [12]. *ACFA* triggers a reset should these regions be illegally accessed. It also upholds immutability and controlled invocation of its TCB, ensuring it can only be entered or exited through dedicated points. Additionally, data used by *CARAMEL* (from Fig. 2) is protected by its hardware from illegal modifications, making attacks that exploit new functionality infeasible.

$\mathcal{A}$dv may also attempt to forge/modify $\mathcal{V}$rf response to avoid remediation. Since all incoming messages are authenticated within TCB, this is infeasible. Alternatively, $\mathcal{A}$dv may block $\mathcal{V}$rf's communication from reaching $\mathcal{P}$rv. In this case, *CARAMEL* will eventually trigger its TCB through **[T4]**, entering a secure software state while trying to retransmit $CF_{Slices}$ until an authenticated response from $\mathcal{V}$rf is received. This ensures that reports are not lost as long as there is eventual communication. Lastly, $\mathcal{A}$dv attempts to disable/misconfigure the CM-UART (e.g., tampering the transmission rate) are also prevented by *CARAMEL* hardware write protection.

## 6 Related Work

***Runtime Attestation.*** *CF-Att*, combining TEE hardware support (from ARM TrustZone) with code instrumentation, was first introduced in C-FLAT [1]. In C-FLAT, branch instructions are instrumented with a *trampolines* to the TrustZone-protected "Secure World" to build a hash-chain of branch destinations uniquely representing the path followed during execution. Several techniques [8, 14, 42, 43, 46, 47, 49] build upon C-FLAT, combining some form of instrumentation and TEE support. They trade-off instrumentation for added hardware cost.

***Runtime Auditing.*** *ACFA* [12] originally proposed *CF-Aud* by augmenting *CF-Att* with "active root of trust" [3] to eventually deliver runtime evidence to $\mathcal{V}$rf (assuming the network is not blocked indefinitely). *ACFA* uses custom hardware to build $CF_{Log}$ and actively trigger execution of its trusted software. The trusted software generates and transmits the report. TRACES [14] demonstrated that *CF-Att* is achievable without requiring custom hardware. It uses instrumentation and TEE support to instantiate the same guarantees of *ACFA*. After sending reports, *ACFA* and TRACES require $\mathcal{P}$rv to wait in a secure state before continuing execution, decreasing $\mathcal{P}$rv utilization. *CARAMEL* aims to address this problem.

***Reliable Data Transfer.*** Network-level reliable data transfer [2, 16, 33, 41, 45] is well-studied in the context where both endpoints are honest. Within our context, *CARAMEL* creates a root of trust with a self-contained communication capability. Different from prior work in this space, this ensures reliable delivery despite software compromise of the endpoint where this RoT resides.

## 7 Conclusion

We present *CARAMEL*, a *CF-Aud* architecture that boosts $\mathcal{P}$rv utilization by mitigating busy-wait in its evidence delivery protocol. *CARAMEL* introduces a novel RoT embedded with a self-contained, reliable communication interface. The evaluation of *CARAMEL*'s open-source prototype [24] shows significant utilization gains compared to the state-of-the-art with modest hardware overhead.

## Acknowledgments

We thank the reviewers for their constructive comments.

## References


[1] Tigist Abera et al. 2016. C-FLAT: control-flow attestation for embedded systems software. In *CCS*. ACM, 743–754.

[2] Ozgür B Akan et al. 2005. Event-to-sink reliable transport in wireless sensor networks. *IEEE/ACM transactions on networking* 13, 5 (2005), 1003–1016.

[3] Esmerald Aliaj et al. 2022. GAROTA: Generalized Active Root-Of-Trust Architecture (for Tiny Embedded Devices). In *31st USENIX Security Symposium (USENIX Security 22)*. 2243–2260.

[4] Omar Alrawi et al. 2019. Sok: Security evaluation of home-based iot deployments. In *2019 IEEE symposium on security and privacy (sp)*. IEEE, 1362–1380.

[5] Mahmoud Ammar et al. 2021. Delegated attestation: scalable remote attestation of commodity cps by blending proofs of execution with software attestation. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 37–47.

[6] Mahmoud Ammar, Adam Caulfield, and Ivan De Oliveira Nunes. 2025. Sok: Integrity, attestation, and auditing of program execution. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3255–3272.

[7] M.S.V. Appaji et al. 2014. An 8051 Microcontroller based Syringe Pump Control System for Surface Micromachining. *Procedia Materials Science* 5 (2014), 1791–1800. https://doi.org/10.1016/j.mspro.2014.07.391 ICAMME.

[8] Md Armanuzzaman, Engin Kirda, and Ziming Zhao. 2025. ENOLA: Efficient Control-Flow Attestation for Embedded Systems. *arXiv preprint arXiv:2501.11207* (2025).

[9] Tyler Bletsch et al. 2011. Jump-oriented programming: a new class of code-reuse attack. In *ACM CCS*. 30–40.

[10] Nathan Burow et al. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 1–33.

[11] Adam Caulfield et al. 2022. ASAP: reconciling asynchronous real-time operations and proofs of execution in simple embedded systems. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 721–726.

[12] Adam Caulfield et al. 2023. ACFA: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5827–5844.

[13] Adam Caulfield et al. 2024. SpecCFA: Enhancing Control Flow Attestation/Auditing via Application-Aware Sub-Path Speculation. In *Annual Computer Security Applications Conference, ACSAC 2024*. IEEE, 563–578.

[14] Adam Caulfield et al. 2024. TRACES: TEE-based Runtime Auditing for Commodity Embedded Systems. In *Annual Computer Security Applications Conference, ACSAC 2024*. IEEE, 257–270.

[15] Ivan De Oliveira Nunes et al. 2022. Casu: Compromise avoidance via secure update for low-end embedded systems. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[16] Budhaditya Deb et al. 2003. ReInForM: Reliable information forwarding using multiple paths in sensor networks. In *28th Annual IEEE International Conference on Local Computer Networks, 2003. LCN'03. Proceedings*. IEEE, 406–415.

[17] Ghada Dessouky et al. 2017. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[18] Ghada Dessouky et al. 2018. Litehax: lightweight hardware-assisted attestation of program execution. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[19] Aiguo Fei, Guangyu Pei, Roy Liu, and Lixia Zhang. 1998. Measurements on delay and hop-count of the internet. In *IEEE GLOBECOM'98-Internet Mini-Conference*.

[20] Aurélien Francillon et al. 2008. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*. 15–26.

[21] Olivier Girard. 2009. openMSP430. https://opencores.org/projects/openmsp430.

[22] Microchip Technology Inc. 2025. ATmega Microcontrollers (MCUs). https://www.microchip.com/en-us/about/corporate-overview/acquisitions/atmel/atmega

[23] Texas Instruments Inc. 2025. MSP430-Microcontroller. https://www.ti.com/product-category/microcontrollers-processors/mcus/overview.html

[24] Alexandra Lengert et al. 2026. Repository for *CARAMEL*: Boosting Device Utilization in Control Flow Auditing. https://github.com/SPINS-RG/CARAMEL.git

[25] Zheyuan Ma et al. 2023. Return-to-Non-Secure Vulnerabilities on ARM Cortex-M TrustZone: Attack and Defense. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[26] Antonio Joia Neto and Ivan De Oliveira Nunes. 2023. 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, 133–146.

[27] Ivan De Oliveira Nunes et al. 2019. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *28th USENIX Security Symposium (USENIX Security 19)*. 1429–1446.

[28] Ivan De Oliveira Nunes et al. 2020. APEX: A verified architecture for proofs of execution on remote devices under full software compromise. In *29th USENIX Security Symposium (USENIX Security 20)*. 771–788.

[29] Ivan De Oliveira Nunes et al. 2021. Tiny-CFA: Minimalistic control-flow attestation using verified proofs of execution. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 641–646.

[30] Ivan De Oliveira Nunes et al. 2022. Privacy-from-birth: Protecting sensed data from malicious sensors with VERSA. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2413–2429.

[31] Ivan De Oliveira Nunes et al. 2024. Toward Remotely Verifiable Software Integrity in Resource-Constrained IoT Devices. *IEEE Communications Magazine* 62, 7 (2024), 58–64.

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

[33] Rabin Patra et al. 2003. Dtnlite: A reliable data transfer architecture for sensor networks. *CS294-1: Deeply Embedded Networks (Fall 2003)* (2003).

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

[35] RiS3-Lab. 2020. Rover. https://github.com/RiS3-Lab/OAT-Project/blob/master/oat-evaluation/roverpi-orig/rovertcp.c

[36] Ryan Roemer et al. 2012. Return-oriented programming: Systems, languages, and applications. *TISSEC* 15, 1 (2012), 1–34.

[37] Seeed-Studio. 2015. Temperature Sensor. https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/temp_humi_sensor

[38] Seeed-Studio. 2015. Ultrasonic Ranger. https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/ultrasonic_ranger

[39] Tohid Shekari et al. 2021. MaMIoT: Manipulation of energy market leveraging high wattage IoT botnets. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1338–1356.

[40] Saleh Soltan et al. 2018. BlackIoT: IoT botnet of high wattage devices can disrupt the power grid. In *27th USENIX security symposium (USENIX security 18)*. 15–32.

[41] Fred Stann et al. 2003. RMST: Reliable data transport in sensor networks. In *Proceedings of the First IEEE International Workshop on Sensor Network Protocols and Applications, 2003.* IEEE, 102–112.

[42] Zhichuang Sun et al. 2020. OAT: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1433–1449.

[43] Flavio Toffalini et al. 2019. ScaRR: Scalable Runtime Remote Attestation for Complex Systems. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 121–134.

[44] Theo Walker. 2022. OpenSyringePump. https://github.com/manimino/OpenSyringePump

[45] Chieh-Yih Wan et al. 2002. PSFQ: a reliable transport protocol for wireless sensor networks. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. 1–11.

[46] Jinwen Wang et al. 2023. ARI: Attestation of Real-time Mission Execution Integrity. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2761–2778.

[47] Nikita Yadav and Vinod Ganapathy. 2023. Whole-Program Control-Flow Path Attestation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2680–2694.

[48] Shaza Zeitouni et al. 2017. Atrium: Runtime attestation resilient under memory attacks. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 384–391.

[49] Yumei Zhang et al. 2021. ReCFA: resilient control-flow attestation. In *Annual Computer Security Applications Conference*. 311–322.

[50] Jean-Karim Zinzindohoué et al. 2017. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1789–1806.