ENOLA: Linear-Space and Low-Overhead Control-Flow Attestation for Microcontroller-based Systems

Md Armanuzzaman
marmanuzzaman@utep.edu
CactiLab, University of Texas at El Paso

Engin Kirda
ek@ccs.neu.edu
Northeastern University

Ziming Zhao
z.zhao@northeastern.edu
CactiLab, Northeastern University

***Abstract*—Control-Flow Attestation (CFA) aims to precisely verify execution paths to a remote verifier. However, existing solutions are fundamentally incompatible with resource-constrained, microcontroller-based systems due to the following reasons: (1) the overhead of transmitting measurement and trace data scales poorly (i.e., often exponentially) with the number of basic blocks or linearly with the length of execution traces, making such approaches impractical even on high-end microprocessor-based systems and entirely infeasible on microcontrollers; (2) cryptographic keys and measurement data are typically stored in memory, rendering them vulnerable to cold boot and memory corruption attacks, which is a serious threat for field-deployed devices; and (3) reliance on software-based measurements, combined with frequent context switches between the Rich Execution Environment (REE) and the Trusted Execution Environment (TEE), introduces significant performance overhead.**

In this paper, we present ENOLA, a linear-space and low-overhead control-flow attestation solution for microcontroller-based systems. ENOLA achieves linear transmission complexity with basic blocks, guaranteeing its scalability for larger programs. Moreover, ENOLA uses hardware-assisted measurement computation present in off-the-shelf devices, and avoids key storage in memory. ENOLA also allocates general-purpose registers for measurements to thwart memory corruption attacks. ENOLA significantly reduces context switching overhead by eliminating transitions from REE to TEE for backward-edge measurements. We developed the ENOLA compiler using LLVM passes and a custom attestation engine targeting the ARMv8.1-M architecture. Our evaluation shows that ENOLA reduces data transmission overhead by an average of 52x on the Embench, while maintaining performance comparable to (or exceeding) that of existing approaches. Furthermore, ENOLA is validated on large, real-world applications such as wolfSSL, enabling scalable control-flow attestation on microcontrollers for the first time.

I. INTRODUCTION

While microcontroller (MCU)-powered embedded systems are ubiquitous, they are vulnerable to cyberattacks, with control-flow hijacking being especially dangerous as it enables arbitrary code execution and full system compromise [1]. Numerous efforts have been made to prevent or detect control-flow hijacking in MCU-based systems. These efforts include forward-edge control-flow integrity [2], [3], shadow stacks [4], [5], return address integrity [6], [7], control-flow violation detection [8], [9], and software-fault isolation. However, since embedded systems are often deployed in the field, it is crucial not only to detect control-flow hijacking, but also to demonstrate to a remote verifier the runtime control-flow transfers of the system.

Control-Flow Attestation (CFA) was introduced to precisely attest a program's execution path [10]. However, existing approaches, such as C-FLAT [11], LO-FAT [12], OAT [13], and Blast [14], have been designed and evaluated for high-performance microprocessors, limiting their applicability to resource-constrained microcontroller-based systems. In particular, they face a significant challenge in transmitting large measurement and/or trace data. The size of these transmissions grows either exponentially with the number of basic blocks in the program, or linearly with the runtime execution trace. As a result, these methods are limited to handling small programs or code snippets. For instance, C-FLAT generates 1,527 bytes of measurement data during a 1.2-second execution of a *syringe pump* program, while Blast produces 257,756 bytes of trace data for a 38-second execution of the *tarfind* Embench program on Cortex-A53 microprocessors.

Additionally, these approaches store cryptographic keys and measurement results in memory without encryption or integrity protection, rendering them vulnerable to cold boot [15] or memory corruption attacks. This vulnerability persists even within TEEs like ARM TrustZone, which provide memory isolation, but do not encrypt memory. Moreover, many existing CFA solutions rely on software-based keyed hash functions for measurement computation, incurring substantial latency on resource-constrained MCUs. This overhead is compounded by frequent context switches from REE to TEE for both forward- (call and jump) and backward-edge (return) control-flow events in the attested program.

In this paper, we present ENOLA, a linear-space and low-overhead control-flow attestation solution for microcontroller-based embedded systems. The primary objectives and challenges of ENOLA are: (1) to algorithmically minimize the footprint of trace and measurement data, which is architecture-agnostic, (2) to safeguard the measurement and attestation keys and measurement results from memory corruption attacks, and (3) to enable efficient measurement computation on off-the-shelf microcontroller-based devices. Additionally, ENOLA must remain secure, resist tampering by privileged software, and preserve trace and measurement integrity.

ENOLA addresses the first challenge by introducing a novel authenticator that combines a *basic block occurrence trace* with two distinct measurements for the forward and backward execution paths. This design achieves linear space complexity with respect to the number of basic blocks in the attested program, ensuring scalability to larger programs. ENOLA leverages a TEE, such as the Cortex-M TrustZone, to securely acquire the basic block occurrence trace. Specifically, the ENOLA compiler instruments the attested programs to report basic block occurrence information to the TEE.

The ENOLA compiler dedicates two general-purpose registers exclusively for storing forward and backward path measurement chains, preventing memory corruption attacks by ensuring they are not spilled onto memory. ENOLA leverages novel hardware-assisted Message Authentication Code (MAC) capabilities, such as the Pointer Authentication (PA) extension in ARMv8.1-M [16] for measurement computations and securely storing keys in PA registers. Additionally, the cryptographic key registers for PA instructions are initialized within the TEE, and the ENOLA code scanner verifies that the REE binary is free of instructions, or Return-Oriented Programming (ROP) gadgets capable of altering them.

To address the performance overhead challenge, ENOLA leverages the MAC hardware capabilities, offering a hundred times speedup over the software-based mechanisms employed by prior solutions. Although ENOLA utilizes a TEE, it minimizes context switches from the attested program by eliminating backward path context switches and computing measurements directly in the REE using MAC instructions.

Lastly, compromised Interrupt Service Routines (ISRs) can interfere with authenticator generation by corrupting the attested program state or the reserved measurement registers. To address this threat, ENOLA integrates an ISC-FLAT-inspired secure ISR dispatcher with its authenticator realization [17]. The dispatcher interposes on REE interrupts through the TEE-protected interrupt controller, securely preserves the attested program context and measurement state, restricts ISR access to the program stack, and ensures a controlled return to the interrupted attested program. Thus, the dispatcher is a supporting protection mechanism for secure authenticator generation rather than the central technical novelty of ENOLA.

We implemented and evaluated ENOLA on the Cortex-M85 MCU, using a syringe pump application, Embench, and wolfSSL [18], [19], [20] with various optimizations. Unlike prior solutions, ENOLA scales to large programs such as wolfSSL while maintaining a competitive performance. Our evaluations show that ENOLA drastically reduces the authenticator size: e.g., Blast produces an average of 21,009 bytes of attestation data on Embench, whereas ENOLA requires only 397 bytes. The paper contributions are summarized as follows:

- We note that existing CFA approaches lack a comprehensive analysis of the space complexity associated with trace and measurement data. To address this gap, we formalize the concept of control-flow attestation, and identify the limitations present in prior studies. Specifically, we characterize the transmitted-authenticator scalability of representative CFA schemes.
- We present ENOLA, which introduces a novel authenticator achieving linear space complexity with respect to the number of basic blocks in the attested program. On the Embench, ENOLA reduces transmitted data by 52 times on average compared to Blast. Additionally, to safeguard the authenticator against memory corruption, ENOLA retains the forward- and backward-edge measurement chains in two compiler-reserved general-purpose registers rather than REE memory, while using TrustZone-protected memory for occurrence trace storage.
- We implement ENOLA on an ARMv8.1-M Cortex-M85 microcontroller system leveraging compiler instrumentation, TrustZone, MPU, and PA-based measurement computation. These mechanisms enable efficient authenticator generation, protect measurement state and measurement keys under the stated threat model, and reduce TEE context switches by computing backward-edge measurements directly in the REE. Our evaluations demonstrate ENOLA's effectiveness in substantially reducing transmitted attestation data while achieving performance overhead that is either lower or comparable to existing approaches. We open-sourced the implementation and evaluation[1].

II. FORMALIZING THE COMPLEXITY OF CFA

Modeling control-flow attestation. In control-flow attestation, a remote verifier $\mathbb{V}$ requests a prover $\mathbb{P}$ to execute a program $\mathcal{P}$ by sending a challenge c to ensure the freshness of the request. $\mathbb{P}$ executes $\mathcal{P}$, and a trusted measurement engine $\mathbb{E}$ within $\mathbb{P}$ generates an attestation report $\mathcal{R}$, which is then transmitted to $\mathbb{V}$. The attestation report $\mathcal{R} = (Auth, Sig_{K_a}(Auth, c))$ is composed of a cumulative authenticator $Auth$ of the control-flow path and a signature over $Auth$ and c using a key K_a. The authenticator $Auth = (T, M)$ may include a full or partial trace T of the control-flow path and/or a measurement M.

We model $\mathcal{P}$'s interprocedural Control Flow Graph (CFG) as $G_{\mathcal{P}} = (V, E)$, where V represents the set of basic blocks, and E represents the set of control-flow transfers among the basic blocks defined by $\mathcal{P}$. Each basic block $v_i \in V$ is characterized by an entry point $v_i.s$, and an exit point $v_i.e$, while each edge $e_i \in E$ is defined by a source address $e_i.s$ and a destination address $e_i.d$. The full execution trace $T_{\mathcal{P}}$ of program $\mathcal{P}$ is represented as a sequence of all non-sequential control-flow transfer destinations $(e_1.d, \ldots, e_l.d)$. We use n to denote the number of forward branch instances in $T_{\mathcal{P}}$, which includes conditional jumps and indirect calls/jumps, and m to denote the number of backward branch instances (i.e., returns) in $T_{\mathcal{P}}$. Thus, the total number of branches is $l = n + m$. In most cases, the number of forward branch instances exceeds the number of backward ones in $T_{\mathcal{P}}$ (i.e., $n > m$), and there are significantly more forward branch instances in $T_{\mathcal{P}}$ than there are basic blocks in $G_{\mathcal{P}}$ (i.e., $n \gg |V|$).

[1]https://github.com/CactiLab/ENOLA-Efficient-CFA-for-Embedded-Systems

	Complexity Analysis			Context Switch Triggers	Evaluation Setup		
	Trace†	Measurement†	Verification‡		CPU	Apps (optimization levels)	Evaluation Size
Naive trace-based	$O(l)$	$O(1)$	$O(l)$	Forward & Backward	n/a	n/a	n/a
OAT [13]	$O(n)$	$O(1)$	$O(l\|E\|)$	Forward & Backward	Multi-core Cortex-A	5 IoT apps including syringe pump [18] (`O0`)	$n < 1,000$
C-FLAT [11]	n/a	$O(2^{\|V\|})$	$O(2^{\|V\|})$	Forward & Backward	Multi-core Cortex-A	Syringe pump (`O0`)	$\|E\| = 322$
LO-FAT [12]	n/a	$O(2^{\|V\|})$	$O(2^{\|V\|})$	n/a	Customized multi-core RISC-V	Syringe pump (`O0`)	$\|E\| = 322$
Blast [14]	$O(2^{\|V\|})$	$O(1)$	$O(2^{\|V\|})$	Forward & Backward	Multi-core Cortex-A	Syringe pump and Embench [19] (`O0`)	$\|V\| < 987$
ENOLA	$O(\|V\|)$	$O(1)$	$O(2^{\|V\|})$	Forward	Single-core Cortex-M	Syringe pump, Embench, and wolfSSL [20](`O0`, `O2`, `O3`, and `Oz`)	$\|V\| < 5,480$

TABLE I: Space†/time‡ complexity, context switch triggers, and evaluation setup in C-FLAT, LO-FAT, OAT, Blast, and ENOLA.

Full-trace based approach. In a naive trace-based attestation scheme, the authenticator $Auth$ consists of the full trace $T_{\mathcal{P}}$, and the measurement is simply the cryptographic hash of the full trace. The combined space complexity of the trace and the measurement is $O(l)$. Subsequently, the verifier $\mathbb{V}$ performs an abstract execution of $\mathcal{P}$. During abstract execution, whenever a non-sequential control-flow transfer site is encountered, $\mathbb{V}$ verifies whether the next record in $T_{\mathcal{P}}$ belongs to the destination set of the site i, denoted as $\cup e_i.d$. The time complexity for verification is $O(l)$ as well. However, due to the theoretically unbounded nature of l (e.g., in the presence of an infinite loop) and its practically substantial size, this approach is impractical, even for small programs running on resource-constrained embedded systems.

OAT. To reduce the trace size, OAT [13] employs three techniques: (1) a single bit to represent for each conditional branch outcome. Compared to recording the entire address, this approach reduces the size while maintaining the same asymptotic space complexity; (2) for forward indirect branches (i.e., indirect jumps and indirect calls), OAT records the destination address; and (3) for backward edges maintains a single chained hash for return addresses, denoted as $H = hash(H \oplus RetAddr)$. For the hash function, OAT uses a software implementation of BLAKE2s [21]. During abstract execution, $\mathbb{V}$ tracks the branch direction using the bit to determine the conditional branch path, validates destination addresses for indirect branches, and calculates and verifies hashes for return instructions. OAT reduces the trace and measurement size complexity to $O(n)$, while the verification time complexity for $\mathbb{V}$ in OAT is $O(l|E|)$.

C-FLAT and LO-FAT. In C-FLAT [11] and LO-FAT [12], the $Auth$ consists solely of measurements, excluding any trace information entirely (i.e., $T = \varnothing$). An exemplary $Auth$ takes the form of $(H_1, \langle H_2, 5\rangle, \langle H_3, 4\rangle, ..., ...)$, where H_1 represents the cumulative hash of a path without loops, $\langle H_2, 5\rangle$ signifies the cumulative hash of a path inside a loop executed five times, and $\langle H_3, 4\rangle$ represents the cumulative hash of a different path inside the same loop executed four times. With this approach, if $\mathcal{P}$ has no loops, the measurement consists of only a single value, resulting in a space complexity of $O(1)$. However, when loops are present, the measurement space complexity in C-FLAT and LO-FAT grows to $O(2^{|V|})$, as it is bounded by the number of possible paths within loops, which is itself bounded by $O(2^{|V|})$. Additionally, because no trace information is available, the verifier $\mathbb{V}$ must explore all possible paths to verify $Auth$, leading to a time complexity of $O(2^{|V|})$. For the hash function, C-FLAT employs a software implementation of BLAKE2, while LO-FAT incorporates a hardware SHA-3 engine directly into a RISC-V MCU.

Blast. To further reduce the trace size, Blast [14] does not generate traces at the basic block level; instead, it operates at the function level. Each entry in the trace takes the form $(Func, Path)$, where $Func$ represents the name of a function, and $Path$ denotes the acyclic Ball-Larus path number [22] within that function. To handle loops, Blast employs Ball-Larus' technique of resetting the path number to a non-zero value at the back-edge of the loop. This approach effectively divides the Control Flow Graph (CFG) into a series of acyclic components, with each component independently computing path numbers. Consequently, for a CFG with loops, the size of a function's trace is represented as a set of path numbers, which in the worst case corresponds to the number of paths in the function, bounded by $O(2^{|V|})$. The measurement is computed as a hash of the trace. Blast utilizes a software implementation of BLAKE2s as the hash function.

Limitations of existing schemes. As shown in Table I, due to the substantial size of the trace or measurement data—e.g., $O(n)$ trace in OAT, $O(2^{|V|})$ measurement in C-FLAT and LO-FAT, and the $O(2^{|V|})$ trace in Blast, existing approaches can only handle small programs or even snippets. Note that our complexity analysis provides an upper bound. In practice, real-world programs typically do not exhibit the worst-case complexity. Nonetheless, the space complexity of authenticators remains a fundamental limitation, affecting the scalability of prior approaches. For instance, OAT was evaluated on operations containing fewer than 1,000 forward branches (n), C-FLAT was tested on programs with up to 322 edges ($|E|$), and Blast was evaluated on programs with at most 987 basic blocks ($|V|$). Moreover, LO-FAT requires modifications to the CPU, making it incompatible with off-the-shelf devices.

III. HARDWARE PRIMITIVES ON MCU

In this section, we discuss the ARMv8-M MCU architecture, on which we implemented and evaluated ENOLA. Note that ENOLA's design principles generalize to other architectures with comparable security extensions. Furthermore, its linear-space transmission property for the attested program is architecture-agnostic.

ARMv8-M Architecture and TrustZone. The ARMv8-M architecture provides a 32-bit physical address space and features 16 general-purpose registers, namely `r0` to `r15`. Among these, `r13`/`sp` serves as the stack pointer, `r14`/`lr` (link register) holds the return address during subroutine calls, and `r15`/`pc` is the program counter. ARMv8-M includes a trusted execution environment called TrustZone. The TrustZone divides the memory into secure and non-secure states, where a memory region can be designated as secure, non-secure callable (NSC), or non-secure. Secure state components

can directly access non-secure resources, while the NSC region acts as a bridge for transitioning from the non-secure to secure state, facilitated by the `sg` (Secure Gateway) instruction.

Memory Protection Unit. The Memory Protection Unit (MPU) in ARMv8-M architecture enables privileged software to partition memory regions and assign region-specific access and execution permissions. For example, it can enforce the (W⊕X) property by marking writable data as non-executable and code regions as read-only. Any violation of these permissions triggers a memory management fault.

ARMv8.1-M Pointer Authentication. The Pointer Authentication (PA) extension includes several instructions, such as `pacg`. These instructions generate a keyed Pointer Authentication Code (PAC) for a pointer or data using the QARMA block cipher [23]. The MCU provides four key registers for different use cases. For example, when the `pacg` instruction is executed in the unprivileged level and non-secure state, the `pac_key_u_ns` register is implicitly used as the key to compute the PAC. These key registers can only be modified using the privileged `msr` (move-to-system-register) instruction. Additionally, the secure state privileged code can modify both keys associated with the non-secure state.

IV. ENOLA

In this section, we first outline the system and threat model, followed by the functionality and workflow of ENOLA.

A. System and Threat Model

System model. ENOLA operates under the assumption that the embedded system processor provides a TEE, an MPU, and hardware capabilities for keyed message authentication code computations within the REE. The attested program can execute at unprivileged or privileged level within the REE, while the ENOLA attestation engine operates within the TEE. The verifier in ENOLA can reside on any system or device, such as x86, Cortex-A, etc., but is assumed to be on a powerful machine or cloud-based system to ensure fast verification.

ENOLA's hardware assumptions are practical for modern security-oriented MCUs. TrustZone was introduced to the M-profile through the ARMv8-M Security Extension and is available in Cortex-M23, Cortex-M33, and subsequent security-enabled implementations [24]. MPU support predates ARMv8-M and is available in Cortex-M3 and later Cortex-M implementations [24]. Key MAC was introduced first in the A-profile with ARMv8.3-A and later extended to the M-profile through the ARMv8.1-M PACBTI extension, implemented in recent processors such as Cortex-M85 [24]. In ENOLA, the secure attestation engine configures and manages TrustZone, MPU permissions, and PA keys at runtime; this engine is developed as part of the framework and constitutes its TCB.

Threat model. We assume the presence of secure boot to ensure both: (1) the ENOLA attestation engine's code and data, and (2) the REE software, are securely loaded at boot time. Not every piece of code within the REE requires attestation; the portion that undergoes attestation is referred to as the attested program. We implement code immutability, e.g., W⊕X, for the attested program. While the ENOLA attestation engine is trusted during runtime, the control flow of the REE software, including the attested program, could be compromised at runtime. Interrupt Service Routines (ISRs) in the REE are also outside the Trusted Computing Base (TCB) and may be compromised, containing control flow hijacking vulnerabilities (e.g., ROP gadgets). Although REE ISRs are untrusted, ENOLA detects malicious ISR executions by monitoring ISR transitions and reflecting ISR-induced deviations in the attestation report. The prover and verifier share both the measurement key (K_m) and attestation key (K_a), which can be practically established through manufacturing-time provisioning, device enrollment, or secure bootstrapping. The measurement key is used for calculating measurements, while the attestation key is employed to sign the attestation report. We assume secure storage is available to protect both the measurement key and attestation key at rest. However, attackers could attempt to compromise memory content within the REE, use ROP gadgets to manipulate general-purpose registers, and modify key registers. TOCTOU attacks [25] and physical attacks such as power analysis or electromagnetic analysis, timing attacks, and data-only attacks considered out of scope for ENOLA.

B. ENOLA Trace and Measurement Schemes

In ENOLA, the authenticator $Auth = (T, M)$ includes a basic block occurrence trace (T_O) and two measurements $M = \langle M_f, M_b \rangle$, representing the forward path (M_f) and backward path (M_b) taken by $\mathbb{P}$. Here, the backward path specifically refers to the sequence of function returns. T_O is not a traditional trace of sequential operations; instead, it is structured as follows:

$$T_O = (\{\langle v_i.s, \#v_i \rangle | i = 0, ..., |V| - 1 \wedge \#v_i \neq 0\}, \{t_i | t_i \notin \bigcup_{i=0}^{|V|-1} v_i.s\})$$

Here, $v_i.s$ represents the start address of the basic block v_i, and $\#v_i$ denotes the occurrence count of v_i during an execution. When $\#v_i$ equals zero, the pair $\langle v_i.s, \#v_i \rangle$ is omitted from T_O. Legitimate indirect branch or call targets correspond to the start addresses of all basic blocks. However, in cases involving non-legitimate targets, such as the middle of an instruction in ROP attacks, T_O can include a list of these target addresses in $t_i \mid t_i \notin \bigcup_{i=0}^{|V|-1} v_i.s$. Essentially, any appearance of t_i in T_O represents a control-flow violation. In practice, $|t_i \mid t_i \notin \bigcup_{i=0}^{|V|-1} v_i.s|$ is a small number, and ENOLA sets a maximum threshold to ensure the constant size of this component. Therefore, the trace space complexity for $\mathbb{P}$ is linear with respect to the number of basic blocks, i.e., $O(|V|)$.

The measurement M is composed of two cumulative measurements for the forward path, i.e., M_f and the backward path, i.e., M_b, respectively. We calculate both measurements using the same hash chain approach: H_{K_m} represents the measurement function with the key of K_m. For M_f, the destination address is denoted by d_i, while for M_b, d_i represents the return address.

$$M_i = \begin{cases} H_{K_m}(0, d_i) & \text{if } i = 0 \\ H_{K_m}(M_{i-1}, d_i) & \text{if } i > 0 \end{cases}$$

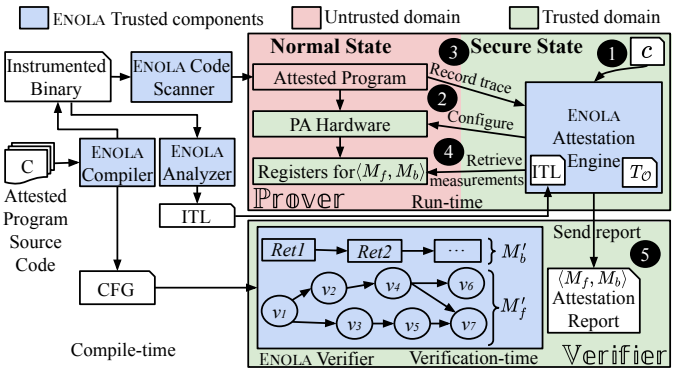


Fig. 1: The workflow of ENOLA

Attestation granularity. The authenticator design of ENOLA is independent of the selected instrumentation granularity. In this work, we employ basic-block-level occurrence reporting and measurement generation to provide fine-grained control-flow attestation. Alternatively, the same representation may be instantiated at a coarser granularity, such as function-level attestation similar to Blast [14], to trade verification granularity for lower instrumentation and runtime overhead while retaining bounded transmitted-authenticator size with respect to the selected attestation units. Evaluating such granularity variants is outside the scope of this work.

C. ENOLA *Components and Workflow*

The workflow (shown in Figure 1) of ENOLA consists of three distinct stages: compile-time, run-time, and verification-time.

Compile-time. ENOLA *Compiler*: The ENOLA compiler instrumentation serves two primary purposes: (1) Reporting Control-Flow Events: The instrumentation reports control-flow transfer events in the program to the attestation engine, which operates in a secure state. This enables the attestation engine to construct T_O, a trace of the program's execution. The details of T_O construction are discussed in §IV-D. (2) Calculating and Storing Measurements: The PA instruction instrumentations compute measurements and store them in designated general-purpose registers. These measurements are further elaborated in §IV-E. Since the measurements are never stored in memory, they are inherently protected from memory corruption attacks. This approach ensures secure and reliable operation of the attested program.

Furthermore, the ENOLA compiler generates an overapproximated control-flow graph (CFG) that is utilized by the ENOLA verifier for abstract execution. While generating a comprehensive CFG encompassing all possible control-flow transfers remains an open research challenge [26], [27], microcontroller programs ($\mathcal{P}$) are typically less complex than desktop applications. This simplicity, combined with advancements in control-flow analysis techniques [28], [29], supports the widely accepted assumption that the CFG for such systems can be treated as complete [11], [30], [31], [32], [33], [34], [35]. This assumption forms the foundation for leveraging the CFG in the attestation and verification processes.

ENOLA *Code Scanner*: Since the compiled program may execute at the privilege level of the normal world, it presents a potential vulnerability where attackers could manipulate PA key registers or reserved general-purpose registers used for measurements. To mitigate this risk, ENOLA incorporates a code scanner to ensure the integrity of the execution environment. The ENOLA code scanner performs a comprehensive analysis to verify that no programs operating in the normal state – including the attested program, libraries, and the kernel – contain any instructions or gadgets capable of tampering with these critical registers. This proactive approach ensures the security of the measurement process by preventing unauthorized modifications at the software level.

Specifically: (1) The code scanner ensures no `msr` instructions write to the PA key registers. (2) It detects and flags `mov` or `pop` instructions that could modify the two designated general-purpose registers used for measurements. The scanner also flags handwritten assembly that contain assembly-level instructions that directly modify the reserved measurement registers or PA key registers. If the scanner identifies such instructions, it issues a warning to the developer, indicating that their presence could compromise the security guarantees of ENOLA. This allows the developer to revise the source code and eliminate the problematic instructions. This methodology has been validated and adopted as a best practice in previous research studies [4], [5], further reinforcing its effectiveness in maintaining a secure execution environment.

ENOLA *Analyzer*: The ENOLA analyzer facilitates efficient attestation by enabling the ENOLA attestation engine to determine whether the destination of an indirect branch corresponds to a valid basic block. This determination is crucial for deciding whether to update $\{\langle v_i.s, \#v_i \rangle\}$ or $\{t_i\}$ during execution. To achieve this, the ENOLA analyzer performs both static and dynamic analysis on the instrumented binary to construct an Indirect Target List (ITL), which contains the valid destination addresses for all potential indirect branches or calls. The presence of the ITL eliminates the need for the attestation engine to dereference and validate the destination address of an indirect branch at run-time to determine whether it corresponds to the starting address of a legitimate basic block. This optimization significantly improves the performance and reliability of the attestation process.

Run-time. ENOLA *Attestation Engine*: At run-time, the attested program $\mathcal{P}$ executes in the normal state, while the ENOLA attestation engine runs in the secure state. At system boot, the attestation engine executes first, and receives a nonce c from the remote verifier (❶). It then retrieves keys from secure storage and configures the PA key registers (❷). During the configuration step, the attestation engine also enables the PA hardware to operate in the normal state and configures the Memory Protection Unit (MPU) to enforce W⊕X. As previously mentioned, during the execution of $\mathcal{P}$, instrumented instructions trigger the attestation engine to record the occurrence trace T_O (❸). Additionally, these instrumented instructions invoke the PA hardware to generate measurements and store the cumulative values $\langle M_f, M_b \rangle$ in reserved registers. On the completion of $\mathcal{P}$, the attestation engine retrieves these values from the reserved registers (❹).

Subsequently, along with c, it generates a signature over $Auth = (T_O, \langle M_f, M_b \rangle)$, constructing the attestation report. This report is then transmitted to $\mathbb{V}$ for verification (❺).

Verification-time. ENOLA *Verifier*: Upon receiving the report from $\mathbb{P}$, $\mathbb{V}$ authenticates the signature using c and K_a. The ENOLA verifier then abstractly executes $\mathcal{P}$, guided by the trace T_O, while comparing the recalculated measurements with the received values. The core component of $\mathbb{V}$ is a backtracking algorithm, which is discussed in detail in §IV-G.

```
1       comparator:
2       cmp  r0, #0x2
3       bne  <branch_2>
4       branch_1:
5       push {r0-r3, lr}
6       mov  r0, pc
7       add.w  r0, r0, #0xb
8       pacg  r10, r0, r10 ;update measurement M_f
9         bl <report_direct> ;report occurrence trace
10      pop {r0-r3, lr}
11      ldr  r0, [sp, #0x4];branch-1 instructions
12      ...
13      branch_2:
14      push {r0-r3, lr}
15      mov  r0, pc
16      add.w  r0, r0, #0xb
17      pacg  r10, r0, r10 ;update measurement M_f
18        bl <report_direct> ;report occurrence trace
19      pop {r0-r3, lr}
20      ldr  r0, [sp]      ;branch-2 instructions
```

Listing 1: Instrumentation example: direct branches. Light green: trace reporting, light blue: measurement calculation

D. Secure Generation of T_O

The ENOLA secure generation of T_O involves three steps: the attestation engine provides runtime branch destination reporting interfaces, the ENOLA compiler instruments interface calls, and the attestation engine logs the resulting trace.

Non-secure callable branch destination reporting interface. The ENOLA attestation engine introduces non-secure callable functions: `report_direct` and `report_indirect`, to allow the attested program to report its branch destinations. The `report_direct` function does not require any parameters, as the attestation engine can directly infer the destination address of branch sites. By contrast, the `report_indirect` function requires the destination address to be provided as a parameter in `r0`. Both are annotated with the ARM Clang compiler `__attribute__((cmse_nonsecure_entry))` directive of the Cortex-M Security Extensions (CMSE). This directive instructs ARM Clang to produce a trampoline within the non-secure callable memory region and position the actual function within the secure memory region. Although both the trampoline and the actual function share the same name, for the direct branch reporting case, they operate within distinct symbol scopes. To report direct branches, the ENOLA compiler instruments the attested program to initiate calls (using the `bl` instruction) to the `report_direct` trampoline, ensuring that the destination is preserved in the `lr` register. The trampoline contains only two instructions: a secure gate and a direct branch. Neither instruction modifies the `lr` register. ENOLA attestation engine extracts the direct branch destination ($v_i.s$) of the attested program from the `lr` register, leveraging a `mov` instruction. For indirect branches, the destination is available in `r0` for the attestation engine.

Instrumenting Reporting: Direct Branches and Loops. For direct branches (e.g., from `if-else` or `switch` statements), ENOLA inserts "`bl <report_direct>`" instructions. These calls are placed at the beginning of each branch target basic block, such as `branch_1` and `branch_2`, as shown in Listing 1. To reduce instrumentation overhead, ENOLA avoids instrumenting basic blocks that end with branch instructions, such as the `comparator` block and the `bne` instruction in Listing 1. Since these blocks immediately dominate their branch target basic blocks, additional tracing is unnecessary. Listing 1 illustrates an example where instrumented calls are inserted on lines 9 and 18 to report the taken path. To ensure correctness and avoid interference, the ENOLA instrumentation surrounds these calls with instructions to store and restore the caller-saved registers, along with the `lr` register, to or from the stack. This precaution is particularly important because these registers may be used as general-purpose registers, especially under `O2` or `Oz` compiler optimizations (as shown in lines 5, 10, 14, and 19).

For loops, ENOLA instruments the loop body and exit basic blocks as shown in the example in Listing 2. The loop condition block is not instrumented because it is the immediate dominator of both the body and exit blocks. This method inherently accounts for the `break` statement, as the block it is in is immediately post-dominated by the exit block.

```
1     ldr  r0, [sp, #0x4]  ;loop counter
2     ldr  r1, [sp, #0x10] ;loop limit
3   loop_condition:
4     cmp  r0, r1
5     bge  <loop_exit>
6   loop_body:
7     push {r0-r3, lr}
8     mov  r0, pc
9     add.w  r0, r0, #0xb
10    pacg  r10, r0, r10 ;update measurement M_f
11    bl <report_direct> ;report occurrence trace
12    pop {r0-r3, lr}
13    ...                ;loop-body instructions
14    adds  r0, #0x1
15    b  <loop_condition>
16  loop_exit:
17    push {r0-r3, lr}
18    mov  r0, pc
19    add.w  r0, r0, #0xb
20    pacg  r10, r0, r10 ;update measurement M_f
21    bl <report_direct> ;report occurrence trace
22    pop {r0-r3, lr}
23    add  sp, #0x18     ;loop-exit instructions
```

Listing 2: Instrumentation example: loops

Instrumenting Reporting: Indirect Branches. Unlike direct branches, which have statically determined targets, indirect branches can be manipulated to jump to arbitrary locations, including potentially the middle of instructions. As a result, in addition to instrumenting the start of target basic blocks, as is done for direct branches, the ENOLA compiler must also instrument indirect calls or jumps to capture the destination address. The destination addresses of indirect branches cannot be derived from the `lr` register. Instead, they may involve any general-purpose register used by the `blx` or `bx` instructions. Listing 3 provides an example of instrumenting an indirect

call site (Line 8), where the trampoline call receives the target address via the `r0` parameter. The process involves saving caller-saved registers (`r0 - r3`) to the stack, then transferring the target address into `r0` (Line 4) to set up for the `report_indirect` trampoline invocation (Line 6). The instrumented sequence concludes with restoring the caller-saved registers from the stack (Line 8).

```
1 movw  r3, r8
2 movt  r3, r9
3 push  {r0-r3}          ;store caller-saved registers
4 mov  r0, r3            ;copy destination to r0
5 pacg  r10, r0, r10     ;update measurement M_f
6 bl <report_indirect>;report occurrence trace
7 pop  {r0-r3}           ;restore caller-saved registers
8 blx  r3                ;indirect call site
```

Listing 3: Instrumentation example: indirect calls

E. Secure and Efficient Measurement Calculation

Measurement key initialization. The ENOLA attestation engine retrieves the measurement key K_m from the secure storage and loads it into the PA key registers `pac_key_u_ns` and `pac_key_p_ns` with the privileged `msr` instruction. This component should be implemented in assembly, leveraging general-purpose registers to temporarily transfer the keys from secure storage to the PA key registers. As a result, the measurement key K_m is never spilled to memory.

Reserving general-purpose registers. To prevent measurements from being spilled to memory, the ENOLA compiler reserves the general-purpose registers `r10` and `r11` to securely store M_f and M_b, respectively. Both registers are initialized to zero before the attestation engine transfers control to the non-secure state. These registers are specifically chosen because they are the callee-saved registers with the highest numerical identifiers in the ARM Procedure Call Standard. Consequently, apart from the instrumented measurement instructions, the compiled REE program-including the attested program-does not utilize these registers, ensuring that the cumulative measurements are not spilled to memory.

Robustness of register reservation. The ENOLA compiler removes `r10` and `r11` from the register allocation pool, making the reservation persistent across compiler optimizations or other flags, including aggressive optimization settings. The implementation details of this compiler reservation are discussed in VII. Furthermore, as `r10` and `r11` are callee-saved registers, the REE program produced by this method is compatible with programs not compiled by the ENOLA compiler, including pre-compiled or vendor libraries. If any uninstrumented program uses these registers, the calling convention will restore their original values upon return. Thus those uninstrumented programs cannot directly influence the measurements. However, such usage could cause the measurements to be spilled into memory, exposing them to potential memory corruption attacks.

Instrumenting measurement calculation: forward path. ENOLA utilizes the `pacg` instruction to compute the measurements. Similar to the instrumentation used for direct branch reporting, the ENOLA compiler inserts PA instructions at the start of all destination basic blocks for forward branches. This is illustrated by the light blue instructions in Listings 1 and 2. Specifically, the instrumentation retrieves the program counter value into a free general-purpose register (e.g., `r4`) and subsequently increments it by a predetermined value to obtain the target basic block address ($v_i.s$) (Lines 6-7 in Listing 1). Then, a `pacg` instruction is instrumented to sign the value in the available general-purpose register, using the previous measurement in `r10` as the modifier (Line 8). Similar to the instrumentation for indirect branch reporting, the ENOLA compiler instruments the indirect branch sites with PA instructions. Listing 3 shows an example `pacg` instrumentation (Line 5) for an indirect call site, where the destination is already copied to the `r0` register for trace reporting.

```
1 prologue:
2   push  {r7, lr}
3   sub  sp, #0x28
4   ...
5 epilogue:
6   add  sp, #0x28
7   ldr  r4, [sp, #0x4]
8   pacg  r11, r4, r11   ;update measurement M_b
9   pop  {r7, pc}
```

Listing 4: Instrumentation example: non-leaf return

Instrumenting measurement calculation: backward path. The ENOLA compiler instruments `pacg` instruction before all function returns to construct the M_b measurements. In the Cortex-M architecture, non-leaf functions preserve return addresses on the stack by pushing `lr`, as shown in Line 2 of Listing 4. Subsequently, function returns are executed by directly popping the saved return address into `pc`. For non-leaf functions, ENOLA instrumentation first loads the return address from the stack into a free general-purpose register. It then inserts a `pacg` instruction to compute M_b, storing the result in the `r11` register (Lines 7 and 8).

Conversely, leaf functions retain the return address in `lr` without spilling it to the stack. They return via the "`bx lr`" or "`mov pc, lr`" instructions. Leaf function returns are directly instrumented using "`pacg r11, lr, r11`". Importantly, ENOLA does not report the occurrence trace for the backward path, thereby **eliminating the need for a context switch** to the attestation engine in the secure state.

F. Secure ISR Dispatcher

Given the frequent and unpredictable nature of interrupts in embedded systems and the fact that they do not conform to a predetermined CFG, ENOLA employs a runtime enforcement, a modified version of ISC-FLAT [17], to ensure ISRs always return benignly to the program $\mathcal{P}$. ENOLA incorporates a secure state interrupt dispatcher that intercepts all normal state interrupts. Leveraging the TrustZone-protected Nested Vectored Interrupt Controller (NVIC), it interposes itself between the interrupt trigger and ISR execution. Upon interception, it saves the context of $\mathcal{P}$: including the stack pointer, measurements M_f and M_b, and normal state system registers onto the secure stack. Before transferring control to the untrusted ISR, the dispatcher configures the normal state MPU to mark $\mathcal{P}$'s stack region as read-only, thereby preventing any unauthorized writes by the ISR. Crucially, this allows the system to maintain control-flow integrity without including the ISRs in the TCB.

On ISR completion, the dispatcher restores the saved context to ensure the correct resumption of $\mathcal{P}$. It also reconfigures the MPU to mark $\mathcal{P}$'s stack region as both readable and writable.

On interrupt entry, ENOLA dispatcher records a flag I_x in the occurrence trace T_O and it's cleared upon ISR's return to the dispatcher. Thus, presence of I_x in T_O signals a deviation from expected ISR behavior, i.e., indicating malicious execution in the normal state interrupt handler or the program $\mathcal{P}$. Notably, I_x preserves $Auth$'s linear space complexity, as it maintains constant space in T_O when a compromised ISR executes.

G. Backtracking Algorithm for Verification

The ENOLA verifier employs the backtracking algorithm to verify the legitimacy of the attested control path. The algorithm takes as inputs the attestation report $\mathcal{R}$ and the control-flow graph $G_{\mathcal{P}}$ of the attested program, which includes the entry basic block ($\mathcal{P}_{entry}$) and the potential exit points ($\mathcal{P}_{exits}$).

The algorithm begins by verifying the attestation report signature and checking for any illegal indirect branch targets. Upon successful verification, it abstractly executes the program using the $G_{\mathcal{P}}$ and validates both the forward and backward measurements. The execution starts at $\mathcal{P}_{entry}$, initializes an empty simulated call stack to track function returns, and recursively executes branches as guided by $Auth$.

Based on the last instruction ($v_c.e$) within the current basic block (v_c), the algorithm executes one of the following: (1) Program Exit: For exits at $v_c.e$, the verifier concludes execution and compares the computed measurements with the received values in $Auth$. (2) Function Call: For a function call at $v_c.e$, the verifier pushes the next instruction address onto the simulated call stack and continues execution at the call target. (3) Conditional Branch: If $v_c.e$ is a conditional branch, the algorithm explores all potential paths emanating from the current basic block after verifying against T_O. A non-zero value in the occurrence count signifies a valid transition to that target. The algorithm decrements count, updates the forward measurement, and proceeds along the path. If no valid targets exist in T_O, the path is deemed invalid, and the algorithm backtracks. (4) Return Instruction: For returns, the backward measurement is updated, and execution continues at the return target, obtained from the simulated function call stack.

Complexity Analysis. The verificaiton algorithm has an exponential worst-case verification bound, $(O(2^{|V|}))$, because the verifier may need to backtrack when multiple paths are consistent with the same basic-block occurrence counts. The largest benchmark, wolfSSL/RSA, has 5,480 basic blocks. In practice, this bound is overly conservative: the verifier usually follows only a few count-compatible successors, while (M_f), (M_b), and the simulated call stack prune inconsistent paths. Hence, practical verification is better viewed as $(O(2^k \cdot l))$, where (l) is the realized abstract execution length and (k) is the number of unresolved regions where multiple paths are consistent with T_O. Significant backtracking is possible in synthetic or highly symmetric CFGs, but it does not dominate realistic embedded workloads such as wolfSSL/RSA.

V. SECURITY ANALYSIS

To successfully hijack the control flow and bypass ENOLA's monitoring, an attacker must meet six key attack prerequisites: (P1): Disable or bypass the instrumented code. (P2): Tamper with ENOLA measurements M_f and M_b, or the measurement key K_m. (P3): Execute malicious control flow to generate hash collisions. (P4): Exploit non-secure callable trampoline interfaces in the attestation engine. (P5): Replay or corrupt $Auth$, T_O, or the attestation key K_a to manipulate verification at $\mathbb{V}$. (P6): Exploit vulnerable ISRs in REE to tamper with $\mathcal{P}$'s stack, return address, or M_f and M_b, thereby disrupting control-flow integrity or trace correctness.

P1 is mitigated through the enforcement of code immutability via the MPU [36], [37]; and any attempt to bypass the instrumentation or interface calls would be reflected in measurements M_f and M_b. For P2, the measurements M_f and M_b or PA key K_m are never spilled into memory and stored in reserved registers accessible only by the instrumented `pacg` instructions. Since the program source code is compiled with the ENOLA compiler, these measurement registers are excluded from the program binary, even under aggressive optimizations (e.g., `O3`, `Oz`). The compiled binary is vetted by the ENOLA code scanner, which flags unauthorized instructions, handwritten assembly, or gadgets that could tamper with the reserved or PA key registers.

Although an attacker can exploit application vulnerabilities for control-flow violations, these are detected through trace and measurement verification. Evading detection would require to forge hash collisions (P3), which is highly infeasible as QARMA block cipher has a collision probability of 2^{-60} [23]. ENOLA's design further makes valid collisions harder by chaining control-flow measurements as modifiers, given *N* transitions, resulting in a cumulative bound of $N * 2^{-60}$.

ENOLA prevents direct manipulation of $Auth$, T_O, or K_a by storing and signing them in the secure state or PA register, while the random nonce c thwarts replay attacks. With M_f and M_b confined to REE registers, the failure to achieve P2, combined with the trusted ARM PA hardware for measurement computation, ensures the integrity of measurements in $Auth$. Thus, exploiting trampoline interface calls becomes the only viable option to manipulate $Auth$ or T_O. However, ENOLA compiler prohibits any direct world-switch calls targeting the attestation engine interfaces, allowing such calls only at instrumented locations. As a result, attackers are prevented from achieving P4, and when combined with TEE and ARM PA security guarantees, P5 is also effectively mitigated.

Normal state ISRs may contain memory corruption bugs exploitable for control flow hijacking (e.g., ROP) or tampering with attested program execution by invalid returns or manipulating measurements. ENOLA mitigates this with a secure ISR dispatcher that intercepts all interrupts, saves context and measurements in secure memory, and marks $\mathcal{P}$'s stack as read-only during ISR execution to block unauthorized writes. If the ISR fails to return to the dispatcher, a flag is recorded in T_O, signaling malicious behavior and mitigating P6.

VI. Residual Risks

ENOLA's guarantees hold under the threat model in Section IV-A: secure boot for the attestation engine and REE software, correct TrustZone/MPU configuration, secure storage for K_m and K_a, and an uncompromised attestation engine at runtime. Compromise of the TrustZone TCB, secure-world software, MPU-enforced W⊕X or $\mathcal{P}$'s stack , unsafe DMA/peripheral configuration, microarchitectural attacks, or TOCTOU attacks is outside scope of ENOLA.

ENOLA's correctness also depends on the quality of the recovered CFG. An incomplete CFG may reject legitimate executions, while an over-approximated CFG may include infeasible paths. Our implementation conservatively favors over-approximation to preserve MCU functionality while relying on CFG analysis to minimize infeasible paths.

ENOLA detects abnormal ISR behavior, including invalid returns and attempts to corrupt $\mathcal{P}$'s stack or measurement state, but does not reconstruct the exact malicious ISR path; such behavior is reflected in T_O. ENOLA supports mixed binaries, including vendor libraries, through memory-based restoration of reserved measurement registers, improving deployability at the cost of potential memory-corruption risks and weaker register-based isolation. These residual risks define ENOLA 's trust boundary: it provides control-flow attestation for the attested program under a securely booted and correctly configured TrustZone/MPU-based TCB, secure key storage, uncompromised measurement logic, and a sound CFG representation.

VII. Implementation

We developed a prototype of ENOLA for the ARMv8.1-M architecture. The ENOLA LLVM compiler modules comprise 4,675 lines of C++ code. It incorporates front- and back-end passes to annotate, instrument, and adjust code for accurate, optimization-resilient measurements. The ENOLA attestation engine consists of 307 lines of C and inline assembly code. The ENOLA analyzer consists of 138 lines of Python code. Additionally, the code scanner and verifier are implemented with 93 and 694 lines of Python code, respectively, leveraging the angr and pyelftools libraries.

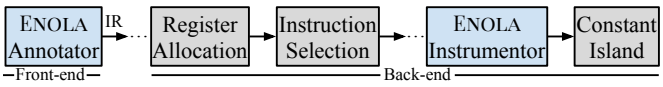


Fig. 2: ENOLA passes in the LLVM pass pipeline

The ENOLA compiler extends the LLVM 16.0.0 ARM embedded toolchain with the front-end ENOLA Annotator and back-end ENOLA Instrumentor passes. We also modify `ARMBaseRegisterInfo` to reserve `r10` and `r11` for measurement by excluding them from LLVM's physical register allocator. Consequently, LLVM cannot use these registers for variables, temporaries, or spill/reload operations, and machine-code-level enforcement prevents `O2`, `O3`, and `Oz` optimizations from overriding this constraint.

The ENOLA Annotator marks functions and valid direct-branch target basic blocks with LLVM IR metadata.

Fig. 3: Syringe pump: Execution time & $Auth$ comparison

The ENOLA Instrumentor then uses these annotations to insert forward-path measurement instructions and `report_direct` trampoline calls. It also instruments indirect branches with `report_indirect` calls and inserts PA instructions before function returns to compute backward-path measurements.

To support `O2`, `O3`, and `Oz` optimizations, all instrumentation is performed by the back-end ENOLA Instrumentor, avoiding LLVM IR phi-node constraints. Because `lr` may be used as a general-purpose register, we modify `ARMFrameLowering` to save and restore it for annotated non-leaf functions, as shown in Listing 1. The Instrumentor also integrates with the `ARMConstantIsland` pass to adjust instrumentation addresses when immediate offsets are limited, as illustrated in Figure 2. Running after instruction selection and register allocation enables precise and efficient instrumentation.

The ENOLA analyzer utilizes angr to identify valid target addresses for instructions inducing indirect control-flow changes. Additionally, it employs dynamic training in a controlled environment with benign inputs to uncover control-flow transfers that may have been overlooked by angr. The analyzer ultimately produces an Indirect Target List (ITL) containing all destination addresses for all indirect calls and branches. The ENOLA code scanner disassembles the generated binary using angr and scans for privileged `msr` instructions or Return-Oriented Programming (ROP) gadgets that could overwrite the ARM PA key or measurement registers.

VIII. Evaluation

We present five key evaluations: (1) a case study of a vulnerable embedded application demonstrating ENOLA 's security guarantees; (2) an overhead analysis across 19 applications from Embench and wolfSSL; (3) a comparative evaluation against existing solutions; (4) an analysis of linearity in $Auth$ size; and (5) a performance evaluation of the ENOLA secure ISR dispatcher. Applications were evaluated with optimization `O0`, `O2`, `O3`, and `Oz`; we report results for `O2` and `Oz`.

A. Evaluation Environment

We evaluated ENOLA on the ARM Versatile Express Cortex-M prototyping FPGA system (V2M-MPS3) [38]. The system was configured as a Cortex-M85 microcontroller running at 25 MHz using the AN555 FPGA image (BSP v1.3.0). ENOLA is the first solution (Table I) to be evaluated on single-core, low-end embedded CPUs, whereas previous solutions were all evaluated on multi-core, high-end, or mid-range CPUs.

B. Evaluations on Syringe Pump Application

We evaluated a version adapted from C-FLAT [11], with minor source-code adjustments to support our compilation process for the Cortex-M85 microcontroller. The program has two main control-flow paths: the `move-syringe` path, which dispenses or withdraws the specified bolus amount (+/−); and the `set-quantity` path, which defines the bolus amount in milliliters from user input. Figure 3 compares the execution overhead and $Auth$ sizes for both control-flow paths.

Execution time overhead and $Auth$ size. Our evaluation attested the entire program, unlike OAT and C-FLAT's partial attestation or Blast's single-path evaluation. For the `move-syringe path` (bolus sizes 0.10, 0.50, 1, and 2 ml), ENOLA achieved the lowest average execution time overhead of **1.3**% and compared to Blast's 6.6% at function-level granularity (Figure 3). ENOLA 's maximum $Auth$ size was **88** bytes, far smaller than C-FLAT's 1,179 bytes for partial attestation; OAT's size was smaller (69 bytes) but excluded loops. In the `set-quantity path`, ENOLA incurred a 67.9% overhead with a **48** bytes $Auth$ due to the small loop-based character-to-integer conversion, whereas C-FLAT's exponential growth produced up to 1,527 bytes for larger inputs.

Case study: anomalous behavior in the move-syringe path. We analyzed anomalous control-flow behavior in the *move-syringe* path, similar to Blast, to demonstrate detection capability of ENOLA. The anomalous path, shown in Listing 5, involves a loop iteration that dispenses or withdraws a specified bolus amount. The total motor steps are governed by the `steps` variable, which depends on `mLBolus` and `ustepsPerML` (Line 1). Here, `mLBolus` is the user input known to the ENOLA verifier, and the loop iteration count (`steps`) can be statically determined since `ustepsPerML` is constant. For example, bolus amounts of `0.010` and `0.011` correspond to 68 and 75 iterations, respectively. As ENOLA instruments the beginning of loop bodies, the verifier - using offline program analysis and user input (`mLBolus`), can detect deviations from expected counts and flag anomalies.

```
1   steps = mLBolus * ustepsPerML;
2   for (i = 0; i < steps ; i++){
3       if(serialStr[0] == '+')
4         dispense();
5       else if(serialStr[0] == '-')
6         withdraw(); }
```

Listing 5: Syringe pump: light blue - loop instrumentation

C. Evaluations on Embench Applications

We evaluate Lines of Code (LoC) and CFG statistics of the Embench [19] applications, along with the ENOLA instrumentation sites. We also evaluated Embench applications with loop unrolling by using the `-mllvm-unroll-count` compiler flag. Figure 4 details the percentage increase with and without loop unrolling in code size due to ENOLA, execution time overhead, the resulting $Auth$ size in bytes, and comparisons of each with Blast. We selected Blast for comparison because it demonstrates the best performance among previous works, and has also been evaluated using Embench.

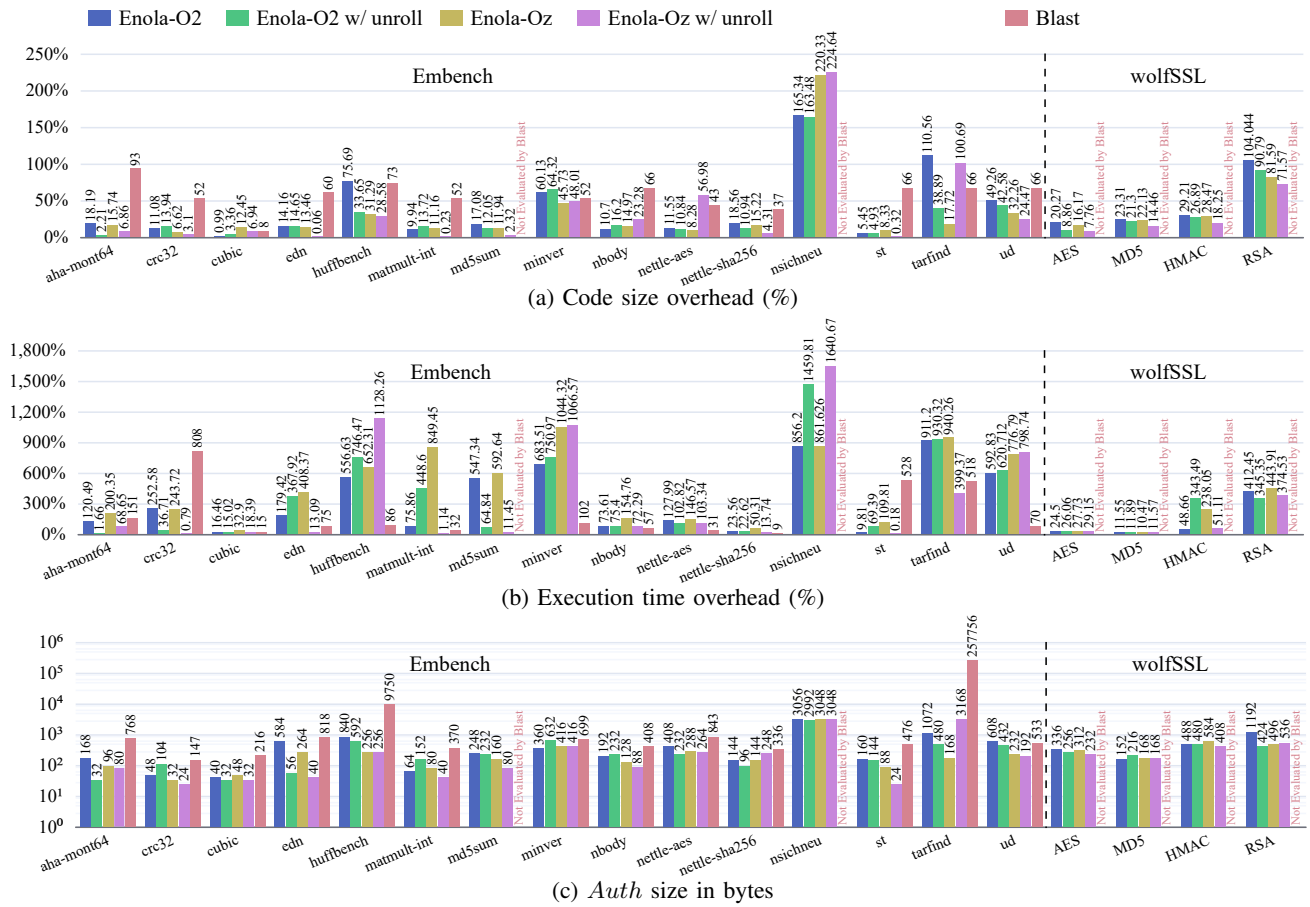
Code size overhead. With `O2`, ENOLA incurs **29.71**% and **38.57**% overhead with unrolling; under `Oz`, **35.3**% and **32.62**%, respectively. Code size overhead depends on the number of instrumented CFG nodes by ENOLA. Our results show that `Oz` induces fewer instrumentation sites in the CFG compared to `O2`, resulting in reduced code size overhead. An exception is `nsichneu`, with similar instrumentation sites on both optimizations, but a smaller original code base amplifies the same instrumentation cost in `Oz`. ENOLA overhead is significantly lower than Blast (64%) on Embench, even without including complex applications like `nsichneu`. This is due to fewer instrumentation sites for direct branches, function calls, address masking, and runtime path ID assignments. OAT reported a 13% size overhead, but it's evaluated on partial programs without loops or function calls.

We also assessed the impact on code size caused by reserving two measurement registers for ENOLA. Figure 5 shows the effect of register reservations on application code size for both optimization levels. On average, `O2` have a higher overhead, as it applies more aggressive optimizations compared to `Oz`.

Execution time overhead. ENOLA incurs lower average execution overhead under `O2` than `Oz`. Although `Oz` produces fewer instrumented basic blocks, code shrinking increases repeated executions of those blocks, resulting in higher runtime cost. For example, `st` shows 9.81% overhead with `O2` (29 instrumentation sites), while `Oz` incurs 11× higher overhead despite only 13 sites, due to increased trampoline or instrumented code invocations (18,852 vs. 230,101). In addition, compiler optimizations such as loop unrolling reduces the execution overhead of ENOLA as shown in Figure 4b.

Compared with prior work, ENOLA demonstrates more realistic overheads, as earlier systems were evaluated on multicore Cortex-A CPUs with limited or coarse-grained attestation. Blast reports 185% overhead with parallel log commit and 175% in single-threaded (requiring function inlining), while providing only function-level verification; ENOLA supports fine-grained basic block-level verification. Even then, Figure 4b illustrates that for six applications: `aha-mont64`, `crc32`, `edn`, `matmult-int`, `st`, and `tarfind`, ENOLA outperforms Blast on at least one optimization level. Across common benchmarks, ENOLA 's average overhead is 322.20% and 278.64% for `O2` (with/without loop unrolling), and 284.54% and 431.53% for `Oz`. Other schemes report higher overheads: OAT achieves 2.7% only on partial code (excluding loops or function calls) but 546% on full programs, while C-FLAT reaches 1004%. Performance could be further improved by adopting function-level attestation similar to Blast, at the cost of reduced security.

ENOLA 's overhead arises from instrumentation (instrumented instructions and trampolines) and context switches to/from secure-state . As shown in Figure 6, context switching accounts for about 26% of total overhead under `O2`. This cost could be reduced by storing traces in normal state memory using SFI techniques as in Blast or utilizing unprivileged load/store instructions as in Silhouette [4].



(a) Code size overhead (%)

(b) Execution time overhead (%)

(c) *Auth* size in bytes

Fig. 4: Code size, *Auth*, and execution time overhead comparison between ENOLA and Blast

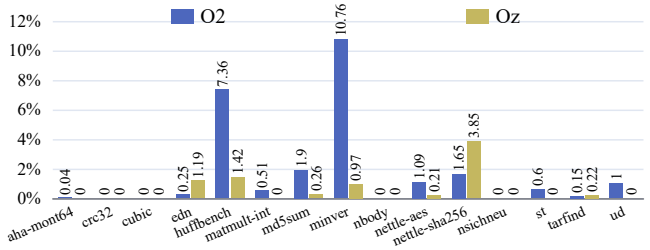


Fig. 5: Code size overhead due to register reservation

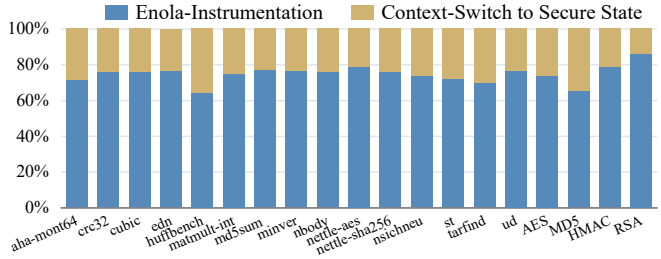


Fig. 6: ENOLA runtime overhead breakdown for instrumentation (`O2`) and context-switch to secure state

Auth **size**. Figure 4c compares the *Auth* sizes of ENOLA with four optimizations and Blast using Embench. ENOLA with `Oz` produces fewer conditional branches compared to `O2`, leading to a reduced number of entries. ENOLA *Auth* depends on the number of unique $v_i.s$ encountered during program execution, while Blast requires separate entries for repeated

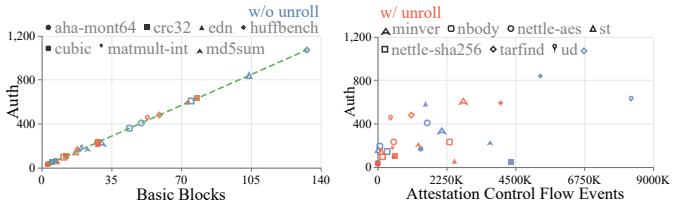


Fig. 7: ENOLA *Auth* size linear to number of basic blocks instead of attestation control flow events

execution of function calls or loops, inflating *Auth* sizes. For instance, `aha-mont64`, `huffbench`, and `tarfind` exhibit repeated execution of identical basic blocks within large or nested loops, leading to 768, 9,750, and 257,756 bytes *Auth* size in Blast, compared to significantly smaller **168**, **840**, and **1,072** bytes in ENOLA under `O2`. An exception to this general trend is `ud` (`O2`), where Blast yields a smaller *Auth* due to a higher frequency of conditional branches than function calls. On average, ENOLA *Auth* size under `O2` and `Oz` is **397** bytes and **224** bytes, respectively, while Blast produces an average of 21,009 bytes for the same set of applications, resulting in a **52** times or greater reduction in *Auth* size.

To demonstrate that ENOLA generates *Auth* with linear space complexity relative to the number of basic blocks, rather than control flow events as in existing works, we plot *Auth* sizes for Embench in Figure 7. The left side shows that the trace

and measurement schemes scale linearly with basic blocks as discussed in Table I and Section IV-B. While, the right side illustrates $Auth$ does not correlate with the number control flow events like existing approaches. For instance, `aha-mont64` and `edn` exhibit a similar amount of control flow events (4,000K) but have 3 and 73 unique instrumented basic blocks, resulting in $Auth$ sizes of 32 and 592 bytes.

D. Evaluations on wolfSSL Applications

We evaluated four larger applications from wolfSSL [20], a library commonly used in embedded systems. Specifically, we examined a 128-bit AES application processing 1KB of data, the MD5 message-digest, an HMAC application generating SHA256 hashes on 1KB of data, and a 2048-bit RSA asymmetric encryption application on 16 bytes of data. Among them, RSA has the most complex CFG with 5,480 basic blocks and 3,309 of them were instrumented by ENOLA.

Code size and execution time overhead. Figures 4a and 4b also include the code size and execution time overheads on the wolfSSL applications. Although the average code size overhead closely resembles that of Embench, the average execution time is notably lower for both optimization levels. This discrepancy can be attributed to the minimal overhead of the MD5 message digest application. The loop unrolling optimization positively affects the code size and execution time overhead for large applications like RSA by reducing the conditional and loop condition basic blocks.

$Auth$ **size**. Figure 4c contains the $Auth$ sizes generated by ENOLA for the four wolfSSL applications. They execute significantly more instrumented unique basic blocks compared to Embench, resulting in larger $Auth$ sizes. Loop unrolling optimization for RSA (`O2`) reduced the unique basic blocks, thereby shrinking the $Auth$ size from **1,192** bytes to **424** bytes. Overall, ENOLA $Auth$ scales linearly with the number of basic blocks even for larger applications.

Although ENOLA substantially reduces transmitted authenticator data, its runtime overhead is not uniformly low across all benchmarks. The primary scalability benefit is communication-side: $Auth$ grows linearly with the number of basic blocks rather than the exponential trace length. Runtime overhead depends on the frequency of instrumented branch targets, trace updates, and compiler optimizations; thus, tight loops or frequently executed instrumentions blocks can still incur noticeable overhead. Therefore, ENOLA is best characterized as a communication-efficient and hardware-assisted CFA design. Further runtime reductions are possible through coarser attestation granularity, as discussed in Section IV-B, fewer secure-state transitions, or hardware trace support.

E. Evaluations on Secure ISR Dispatcher

We evaluated the performance of ENOLA interrupt dispatcher using a toggle-LED application. The NVIC was configured to route the SysTick interrupt to the secure state, allowing the dispatcher to securely invoke the untrusted ISR in the normal state. The SysTick timer generated an interrupt every 10ms, resulting in over 870.24 million interrupts during the application's lifetime. In the baseline configuration, the program finishes in 6.73 million clock cycles, whereas with the ENOLA dispatcher, it takes 7.53 million clock cycles, reflecting a modest overhead of **13.33**% incurred by the runtime enforcement, MPU configuration, and context management.

IX. RELATED WORK

Attestation of embedded software integrity. Pioneer [39] addresses the challenge of achieving verifiable code execution on untrusted legacy systems. The VIPER [40] approach extends these efforts by detecting peripheral firmware proxy attacks in remote attestation frameworks. PUFatt [41] combines physically unclonable functions with remote attestation to counter impersonation attacks. To systematize these advances, Armknecht et al. [42] proposed a mechanism for analyzing and designing software attestation schemes. In parallel, several works explored hardware-based remote attestation to establish a dynamic root of trust on untrusted platforms [43]. While these static approaches can verify code integrity, they are limited in detecting dynamic control-flow hijacking attacks.

Attestation of control-flow. Beyond C-FLAT, OAT, and Blast, several other control-flow attestation (CFA) approaches have been proposed. DIAT [44] and ARI [45] employ software modularization based on criticality, combining monitoring and attestation of control and data flows. While those improve performance, they fail to detect sophisticated control-flow attacks. ReCFA [34], SpecCFA [46], and ScaRR [47] introduce coarse-grained program analysis to reduce the number of recorded entries through call-site filtering, control-flow event folding, and subpath checkpoint separation. ReCFA further leverages hardware memory protection for critical data structures and userspace kernel trapping, although these features are rarely available on embedded systems. ZEKRA [48] proposes a cryptographic method to delegate execution-path attestation to a dedicated third party, while subsequent works [49], [50] analyze the capabilities of both provers and verifiers in CFA. CFA+ [51] leverages the ARMv8.5-A BTI feature to implement a hybrid mechanism enforcing local CFI and supports lightweight attestation, though without interrupt handling.

Attestation via specialized hardware. VRASED, PEARTS, and TRACES employ modified/existing hardware for monitoring alongside software measurements to enable verification of program execution integrity [52], [53]. SANCUS utilizes customized hardware for static attestation [54]. ACFA [55], LiteHAX [56], Tiny-CFA [57], and LO-FAT [12] are other approaches to CFA through specialized hardware, thus unsuitable for commodity devices. IDA [58] leverages specialized hardware to monitor program counters, IRQs, memory addresses, and DMA writes, facilitating interrupt-aware attestation. To tackle the TOCTOU challenge, RATA developed a specialized hardware aimed at reducing the time gap between the attestation state and reporting [25]. ENOLA is complementary to the TOCTOU defense approaches and focuses on full program attestation without hardware modification.

ARM PA usage beyond authentication. Several recent works leveraged ARM PA for return address protection with measurement chains, providing spatial and temporal memory protection, pointer integrity, etc. As a trusted measurement module on the normal state of TrustZone and digest storage on the higher bits of the pointer itself in Cortex-A, has led to efficient temporal and spatial memory protection in recent works of PTAuth [59], PAC it up [16], and PACMem [60]. PACStack overcomes the limitation of hash collision by constructing a measurement chain for all return addresses in the program [61]. ENOLA is the first work to utilize PA for control-flow attestation on embedded systems.

X. Limitations and Future Work

Minimizing runtime. To reduce runtime overhead and TEE switches, ENOLA can be realised on top of Blast or leveraging hardware trace components (Micro Trace Buffer to record instruction traces instead of software-based instrumentation. Furthermore, to eliminate reliance on TEE, unprivileged load-/store instructions – similar to those used in Silhouette [4] and Kage [5] – can be employed to save and access occurrence traces directly in the normal state. This approach may broaden ENOLA's applicability while reducing TEE switches, thereby improving overall performance.

TOCTOU defense and interrupt. Defenses like RATA [25] with hardware modifications can be enabled on top of ENOLA. ENOLA currently lacks multitask attestation support for environments like real-time operating systems. Similarly, its secure interrupt dispatcher is designed for bare-metal applications and does not support nested interrupts, both of which represent promising directions for future enhancement.

XI. Conclusion

Existing CFA solutions face significant challenges in handling the transmission of large amounts of measurement and/or trace data and often suffer from the slow performance of software-based measurement calculations. These limitations restrict their applicability to small programs or code snippets on high-performance devices. In this paper, we presented ENOLA, a novel CFA solution designed specifically for embedded systems. ENOLA incorporates a novel authenticator with linear space complexity relative to the number of basic blocks and leverages hardware-assisted message authentication code capabilities for efficient measurement computation. It also employs a trusted execution environment and reserves general-purpose registers to mitigate memory corruption attacks.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation (NSF) under grants 2508320, 2453496, and 2512972. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or any of its agencies.

References

[1] I. Stellios, P. Kotzanikolaou, M. Psarakis, C. Alcaraz, and J. Lopez, "A survey of iot-enabled cyberattacks:assessing attack paths to critical infrastructures and services," *IEEE Communications Surveys & Tutorials*, 2018.

[2] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2017.

[3] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-flow integrity for real-time embedded systems," in *Euromicro Conference on Real-Time Systems*, 2019.

[4] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *USENIX Security Symposium*, 2020.

[5] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell, "Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage," in *USENIX Security Symposium*, 2022.

[6] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing Embedded Systems with Return Address Integrity," in *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[7] B. Kim, K. Lee, W. Park, J. Cho, and B. Lee, "Rio: Return instruction obfuscation for bare-metal iot devices," *IEEE Access*, 2023.

[8] X. Tan and Z. Zhao, "SHERLOC: Secure and Holistic Control-Flow Violation Detection on Embedded Systems," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[9] Y. Wang, C. Lemieux Mack, X. Tan, N. Zhang, Z. Zhao, S. Baruah, and B. C. Ward, "InsectACIDE: Debugger-Based Holistic Asynchronous CFI for Embedded System," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2024.

[10] M. Ammar, A. Caulfield, and I. D. O. Nunes, "Sok: Integrity, attestation, and auditing of program execution," in *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024.

[11] 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 Conference on Computer and Communications Security (CCS)*, 2016.

[12] 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 *Annual Design Automation Conference (DAC)*, 2017.

[13] Z. Sun, B. Feng, L. Lu, and S. Jha, "OAT: Attesting operation integrity of embedded devices," in *IEEE Symposium on Security and Privacy*, 2020.

[14] N. Yadav and V. Ganapathy, "Whole-program control-flow path attestation," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[15] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, 2009.

[16] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *USENIX Security Symposium*, 2019.

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

[18] Github, "Open Syringe Pump," https://github.com/manimino/OpenSyringePump.

[19] Github, "Embench-IoT," https://github.com/embench/embench-iot.

[20] M. T. Contributors, "Embedded TLS Library," https://www.wolfssl.com.

[21] B. Contributors, "," https://www.blake2.net/.

[22] T. Ball and J. R. Larus, "Efficient path profiling," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1996.

[23] Avanzi, Roberto, "The QARMA block cipher family. Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes," *Transactions on Symmetric Cryptology*, 2017.

[24] ARM, "ARM TrustZone," https://developer.arm.com/documentation/dd i0553/latest.

[25] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the toctou problem in remote attestation," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[26] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX Security Symposium*, 2015.

[27] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.
[28] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *ACM Conference on Computer and Communications Security (CCS)*, 2019.
[29] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level." in *Network and Distributed System Security Symposium (NDSS)*, 2021.
[30] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, "In-kernel control-flow integrity on commodity oses using arm pointer authentication," in *USENIX Security Symposium*, 2022.
[31] K. Lu, "Practical program modularization with type-based dependence analysis," in *IEEE Symposium on Security and Privacy (S&P)*, 2023.
[32] J.-J. Bai, T. Li, K. Lu, and S.-M. Hu, "Static detection of unsafe dma accesses in device drivers," in *USENIX Security Symposium*, 2021.
[33] N. Yang, W. Shen, J. Li, Y. Yang, K. Lu, J. Xiao, T. Zhou, C. Qin, W. Yu, J. Ma *et al.*, "Demons in the shared kernel: Abstract resource attacks against os-level virtualization," in *ACM Conference on Computer and Communications Security*, 2021.
[34] Y. Zhang, X. Liu, C. Sun, D. Zeng, G. Tan, X. Kan, and S. Ma, "ReCFA: resilient control-flow attestation," in *Annual Computer Security Applications Conference (ACSAC)*, 2021.
[35] T. Cloosters, D. Paaßen, J. Wang, O. Draissi, P. Jauernig, E. Stapf, L. Davi, and A.-R. Sadeghi, "Riscyrop: Automated return-oriented programming attacks on risc-v and arm64," in *International Symposium on Research in Attacks, Intrusions and Defenses(RAID)*, 2022.
[36] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting bare-metal embedded systems with privilege overlays," in *IEEE Symposium on Security and Privacy*, 2017.
[37] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing real-time microcontroller systems through customized memory view switching." in *NDSS*, 2018.
[38] ARM, "ARM MPS3," https://www.arm.com/products/development-tools/development-boards/mps3.
[39] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *ACM Symposium on Operating Systems Principles*, 2005.
[40] Y. Li, J. M. McCune, and A. Perrig, "VIPER: Verifying the integrity of peripherals' firmware," in *ACM conference on Computer and Communications Security (CCS)*, 2011.
[41] J. Kong, F. Koushanfar, P. K. Pendyala, A.-R. Sadeghi, and C. Wachsmann, "PUFatt: Embedded platform attestation based on novel processor-based PUFs," in *Annual Design Automation Conference (DAC)*, 2014.
[42] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "A security framework for the analysis and design of software attestation," in *ACM Conference on Computer and Communications security (CCS)*, 2013.
[43] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "Smart: secure and minimal architecture for (establishing dynamic) root of trust." in *Network and Distributed System Security Symposium (NDSS)*, 2012.
[44] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A.-R. Sadeghi, and M. Schunter, "Diat: Data integrity attestation for resilient collaboration of autonomous systems." in *NDSS*, 2019.
[45] 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," 2023.
[46] A. Caulfield, L. Tyler, and I. D. O. Nunes, "Speccfa: Enhancing control flow attestation/auditing via application-aware sub-path speculation," in *Annual Computer Security Applications Conference (ACSAC)*, 2024.
[47] F. Toffalini, E. Losiouk, A. Biondo, J. Zhou, and M. Conti, "Scarr: Scalable runtime remote attestation for complex systems," in *Research in Attacks, Intrusions and Defenses (RAID)*, 2019.
[48] H. B. Debes, E. Dushku, T. Giannetsos, and A. Marandi, "ZEKRA: Zero-Knowledge Control-Flow Attestation," in *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2023.
[49] M. Ammar, A. Caulfield, and I. D. O. Nunes, "Sok: Integrity, attestation, and auditing of program execution," in *IEEE Symposium on Security and Privacy (SP)*, 2025.
[50] A. I. Caulfield, N. Rattanavipanon, and I. D. O. Nunes, "Run-time attestation and auditing: The verifier's perspective," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2025.
[51] M. Ammar, A. Abdelraoof, and S. Vlasceanu, "On bridging the gap between control flow integrity and attestation schemes," in *USENIX Security Symposium*, 2024.
[52] 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.
[53] A. Joia Neto, N. Rattanavipanon, and I. De Oliveira Nunes, "Provable execution in real-time embedded systems," 2025.
[54] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *USENIX Security Symposium*, 2013.
[55] 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.
[56] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: lightweight hardware-assisted attestation of program execution," in *IEEE/ACM International Conference on Computer-Aided Design*, 2018.
[57] I. D. O. Nunes, S. Jakkamsetti, and G. Tsudik, "Tiny-cfa: Minimalistic control-flow attestation using verified proofs of execution," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021.
[58] F. Arkannezhad, J. Feng, and N. Sehatbakhsh, "Ida: Hybrid attestation with support for interrupts and toctou," 2024.
[59] R. M. Farkhani, M. Ahmadi, and L. Lu, "PTAuth: Temporal Memory Safety via Robust Points-to Authentication," in *USENIX Security Symposium*, 2021.
[60] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang, "Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.
[61] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "PACStack: an Authenticated Call Stack," in *USENIX Security Symposium*, 2021.

Md Armanuzzaman is an Assistant Professor in the Department of Computer Science at The University of Texas at El Paso, where he directs the Trusted Systems and Software Security Laboratory (T3SLab). His research interests include systems and software security for embedded systems, confidential computing, agentic systems, and program analysis. His work has appeared in venues such as ACM CCS, NDSS, ACM AsiaCCS, ACM EMSOFT, IEEE TIFS, IEEE SEED, and ACM SAC.

Engin Kirda is a professor of computer science at Northeastern University in Boston. He previously held faculty positions at Institute Eurecom and the Technical University of Vienna, where he co-founded the Secure Systems Lab. His research focuses on malware analysis and detection, web application security, and social networking security, with more than 175 peer-reviewed publications. He has chaired top-tier security conferences, including NDSS 2015, USENIX Security 2017, AsiaCCS 2019, and ACM CCS 2023 and 2024. He co-founded Lastline, Inc., a malware-detection company acquired by VMware in 2020 and now part of Broadcom.

Ziming Zhao is an Associate Professor at the Khoury College of Computer Sciences and the director of the CyberspACe securiTy and forensIcs Lab (CactiLab), Northeastern University. His research has been supported by the U.S. National Science Foundation (NSF), the U.S. Department of Defense, the U.S. Air Force Office of Scientific Research, and the U.S. National Centers of Academic Excellence in Cybersecurity. He is a recipient of an NSF CAREER award and an NSF CRII award. He is also a recipient of best paper awards from USENIX Security 2019, ACM AsiaCCS 2022, ACM CODASPY 2014, and ITU Kaleidoscope 2016.