

ENOLA: Efficient Control-Flow Attestation for Embedded Systems

Md Armanuzzaman
m.armanuzzaman@northeastern.edu
 CactiLab, Northeastern University

Engin Kirda
ek@ccs.neu.edu
 Northeastern University

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

Abstract

Microcontroller-based embedded systems are vital in daily life, but are especially vulnerable to control-flow hijacking attacks due to hardware and software constraints. Control-Flow Attestation (CFA) aims to precisely attest the execution path of a program to a remote verifier. However, existing CFA solutions face challenges with large measurement and/or trace data, limiting these solutions to small programs. In addition, slow software-based measurement calculations limit their feasibility for microcontroller systems. In this paper, we present ENOLA, an efficient control-flow attestation solution for low-end embedded systems. ENOLA introduces a novel authenticator that achieves linear space complexity. Moreover, ENOLA capitalizes on the latest hardware-assisted message authentication code computation capabilities found in commercially-available devices for measurement computation. ENOLA employs a trusted execution environment, and allocates general-purpose registers to thwart memory corruption attacks. We have developed the ENOLA compiler through LLVM passes and attestation engine on the ARMv8.1-M architecture. Our evaluations demonstrate ENOLA’s effectiveness in minimizing data transmission, while achieving lower or comparable performance to the existing works.

1 Introduction

Embedded systems powered by microcontrollers (MCUs) play a vital role in everyday life by controlling and enabling a wide range of systems. Unfortunately, despite the unparalleled benefits these systems offer, they remain susceptible to cyberattacks [33, 56]. Among these threats, control-flow hijacking stands out as particularly dangerous, as it allows arbitrary code execution, and grants attackers full control of the system. Numerous efforts have been made to prevent or detect control-flow hijacking in MCU-based systems. These efforts include techniques such as forward-edge control-flow integrity [37, 52, 61], shadow stack implementations [30, 69], return address integrity mechanisms [9, 38], control-flow violation detection [58, 63], and software-fault isolation [60].

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.

To address this need, Control-Flow Attestation (CFA) was introduced as a technique for precise attestation of a program’s execution path [10]. However, existing CFA approaches, such as C-FLAT [7], LO-FAT [29], OAT [57], and Blast [65], face a significant challenge in transmitting large volumes of 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 only small programs or code snippets.

Additionally, many existing CFA solutions depend on software-implemented keyed hash functions to compute measurements, which introduces several issues: 1) they store cryptographic keys in memory, making them vulnerable to memory corruption or cold-boot attacks [36], even when stored within a Trusted Execution Environment (TEE), 2) the resource constraints of MCUs significantly hinder the performance of software-based implementations, leading to considerable delays in measurement calculations. Notably, existing solutions have predominantly been implemented and evaluated on high-performance microprocessors rather than resource-constrained MCUs.

In this paper, we introduce ENOLA, an efficient control-flow attestation solution designed specifically for low-end embedded systems. The primary objectives and challenges of ENOLA are: 1) to algorithmically minimize the footprint of trace and measurement data, 2) to enable efficient measurement computation on off-the-shelf low-end embedded devices. Additionally, ENOLA must ensure its own security, preventing tampering, or disabling by privileged software, and guaranteeing the integrity of trace and measurement data.

Our work addresses the first challenge by introducing a novel authenticator that combines a *basic block occurrence trace*

with two distinct measurements representing both the forward and backward execution paths. This authenticator achieves linear space complexity with respect to the number of basic blocks in the attested program, ensuring scalability and applicability to larger programs. ENOLA leverages a Trusted Execution Environment (TEE), such as the Cortex-M TrustZone, to securely acquire the basic block occurrence trace. Specifically, the ENOLA compiler instruments the attested program, enabling it to report basic block occurrence information to the TEE.

To address the second challenge, ENOLA leverages novel hardware-assisted Message Authentication Code (MAC) capabilities, such as the Pointer Authentication (PA) extension in ARMv8.1-M [6, 45], achieving a speed improvement of over two orders of magnitude. Specifically, the ENOLA compiler instruments instructions to utilize these features during forward and backward control-flow transfers, enabling the calculation of two chained measurements. Unlike existing approaches that incur significant overhead due to context switches to the TEE for measurement, ENOLA minimizes this overhead by utilizing the MAC extension within the Rich Execution Environment (REE).

To protect these measurements from memory corruption attacks, the ENOLA compiler dedicates two general-purpose registers exclusively for storing them, ensuring they are not spilled onto memory. The cryptographic key registers for these instructions are initialized within the TEE, and the ENOLA code scanner verifies that the compiled REE binary is free of instructions, or Return-Oriented Programming (ROP) gadgets capable of altering the key registers.

We implemented ENOLA on the Cortex-M85 MCU, and evaluated its effectiveness and performance using a syringe pump application [34], Embench [5], and wolfSSL [64] with $\mathcal{O}2$ and $\mathcal{O}z$ optimization levels. Notably, prior solutions have not been tested on large programs such as wolfSSL, and have been limited to evaluations using the less practical $\mathcal{O}0$ optimization level. Our evaluation results show that ENOLA significantly reduces the size of authenticators while maintaining competitive performance compared to existing approaches. The contributions of this paper 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.
- We present ENOLA, a secure and efficient approach to CFA. ENOLA introduces an innovative authenticator that achieves linear space complexity with respect to the number of basic blocks in the attested program. Additionally, it leverages novel security extensions for measurement generation, significantly reducing the need for context switches to the TEE for measurements.

- We implement ENOLA and evaluate its security and performance on the Cortex-M85 microcontroller system. Our evaluations demonstrate ENOLA’s effectiveness in minimizing data transmission while achieving performance that is either lower or comparable to existing approaches. To promote transparency and facilitate artifact evaluation, we have made the source code available online¹.

2 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 $(e_1.d, \dots, e_l.d)$. 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., $n \gg |V|$).

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

¹<https://github.com/CactiLab/ENOLA-Efficient-CFA-for-Embedded-Systems>

	Complexity Analysis			Evaluation Environment		
	Trace [†]	Measurement [‡]	Verification [‡]	CPU	Apps (optimization levels)	Evaluation Size
Naive trace-based	$O(l)$	$O(1)$	$O(l)$	n/a	n/a	n/a
OAT [57]	$O(n)$	$O(1)$	$O(E)$	Multi-core Cortex-A	5 IoT apps including syringe pump [34] (00)	$n < 1,000$
C-FLAT [7]	n/a	$O(2^{ V })$	$O(2^{ V })$	Multi-core Cortex-A	Syringe pump (00)	$ E = 322$
LO-FAT [29]	n/a	$O(2^{ V })$	$O(2^{ V })$	Customized multi-core RISC-V	Syringe pump (00)	$ E = 322$
Blast [65]	$O(2^{ V })$	$O(1)$	$O(2^{ V })$	Multi-core Cortex-A	Syringe pump and Embench [5] (00)	$ V < 987$
ENOLA	$O(V)$	$O(1)$	$O(2^{ V })$	Single-core Cortex-M	Syringe pump, Embench, and wolfSSL [64] (02 and 0z)	$ V < 5,480$

Table 1: Comparing the space[†]/time[‡] complexity and evaluation environments in C-FLAT, LO-FAT, OAT, Blast, and ENOLA.

approach is impractical, even for small programs running on resource-constrained embedded systems.

OAT. To reduce the trace size, OAT [57] employs three techniques: 1) For each conditional branch, OAT uses a single bit to denote whether the branch is taken. Compared to recording the entire address, this approach reduces the size while maintaining the same asymptotic space complexity. 2) For each forward indirect branch (i.e., indirect jumps and indirect calls), OAT records the destination address. 3) For backward edges, OAT maintains a single chained hash value of return addresses, denoted as $H = \text{hash}(H \oplus \text{RetAddr})$. For the hash function, OAT uses a software implementation of BLAKE-2s [16]. During abstract execution, \forall tracks the branch direction using the bit to determine whether a conditional branch is taken or not, 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 \forall in OAT is $O(|E|)$.

C-FLAT and LO-FAT. In C-FLAT [7] and LO-FAT [29], the authenticator *Auth* consists solely of measurements, excluding any trace information entirely (i.e., $T = \emptyset$). An exemplary *Auth* takes the form of $(H_1, \langle H_2, 5 \rangle, \langle H_3, 4 \rangle, \dots, \dots)$, 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 \forall 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 BLAKE-2 [16], while LO-FAT incorporates a hardware SHA-3 engine directly into a RISC-V MCU.

Blast. To further reduce the trace size, Blast [65] does not generate traces at the basic block level; instead, it operates at the function level. Each entry in the trace takes the form $(\text{Func}, \text{Path})$, where *Func* represents the name of a function, and *Path* denotes the acyclic Ball-Larus path number [20] 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 in Blast is computed as a hash of the trace. Blast utilizes a software implementation of BLAKE-2s [16] as the hash function.

Limitations of existing schemes. As shown in Table 1, due to the substantial size of the trace or measurement data—such as the $O(n)$ trace in OAT, the $O(2^{|V|})$ measurement in C-FLAT and LO-FAT, and the $O(2^{|V|})$ trace in Blast, existing approaches can only handle whole but small programs or even snippets, often referred to as operations (i.e., self-contained tasks or logic). 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 previous 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.

3 Hardware Primitives on MCU

In this section, we discuss the ARMv8-M MCU architecture, on which we implemented and evaluated ENOLA.

ARMv8-M Architecture and TrustZone. The ARMv8-M architecture is designed with a 32-bit physical address space. It features 16 general-purpose registers, namely r_0 to r_{15} . Among these, r_{13}/sp serves as the stack pointer, r_{14}/lr (link register) holds the return address during subroutine calls, and r_{15}/pc is the program counter. ARMv8-M includes a trusted execution environment called TrustZone. The division of secure and non-secure states in TrustZone is based on a memory map, 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 the secure state, facilitated by the *sg* (Secure

Gateway) instruction.

ARMv8.1-M Pointer Authentication. The Pointer Authentication (PA) extension [4] includes several instructions, such as `pacg`, as detailed in Table 2 (Appendix). These instructions generate a keyed Pointer Authentication Code (PAC) for a pointer or data using the QARMA block cipher [18]. The MCU provides four key registers for different use cases as shown in Table 3 (Appendix). 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.

4 ENOLA

In this section, we first outline the system and threat model, and then describe the functionality of each ENOLA module. As depicted in Figure 1, ENOLA consists of compile-time modules, a run-time attestation engine, and a remote verification module.

4.1 System and Threat Model

System model. ENOLA operates under the assumption that the processor in the embedded system provides a Trusted Execution Environment (TEE) and hardware capabilities for keyed message authentication code computations within the Rich Execution Environment (REE). The attested program can execute at either the 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.

Threat model. We assume the presence of a secure boot mechanism in the embedded system 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 assume code immutability, e.g., $W \oplus X$, 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. A control-flow hijacking attack might lead to the execution of existing functions or unintended ROP gadgets. The prover and verifier share both the measurement key (K_m) and attestation key (K_a). 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 [26], interrupt service attestation, and physical attacks such as power analysis, timing attacks, and electromagnetic analysis are considered out of scope for ENOLA.

4.2 ENOLA Trace and Measurement Schemes

In ENOLA, the authenticator $Auth = (T, M)$ is designed to include a basic block occurrence trace (T_O) and measurements $M = \langle M_f, M_b \rangle$, which represent the forward path (M_f) and the backward path (M_b) taken by \mathbb{P} . Within the scope of this paper, the backward path is specifically defined as the sequence of function returns. T_O is not a traditional trace of sequential operations; instead, it is structured as follows:

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

Here, $v_i.s$ represents the start address of the basic block v_i , and $\#v_i$ indicates 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 maintain the constant size of this component. Therefore, the trace space complexity 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 utilize the same hash chain approach to calculate both measurements as follows. 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.

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

4.3 ENOLA Components and Workflow

The workflow of ENOLA consists of three distinct stages: compile-time, run-time, and verification-time.

Compile-time. ENOLA *Compiler*: The instrumentation by the ENOLA compiler 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 will be discussed in §4.4. 2)

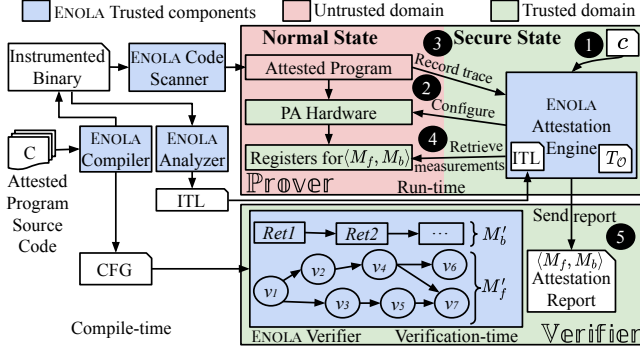


Figure 1: The workflow of ENOLA

Calculating and Storing Measurements: The instrumentation uses PA instructions to compute measurements and stores them in designated general-purpose registers. These measurements are further elaborated in §4.5. 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 binary program.

Furthermore, the ENOLA compiler generates an over-approximated control-flow graph (CFG) that is utilized by the ENOLA verifier for abstract execution. Note that generating a comprehensive CFG encompassing all possible control-flow transfers remains an open research challenge [22, 25]. However, in the context of microcontroller-based systems, the program \mathcal{P} is typically less complex than desktop applications. This simplicity, combined with advancements in control-flow analysis techniques [40, 47], supports the widely accepted assumption that the CFG for such systems can be treated as complete [7, 19, 24, 46, 66–68].

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.

This goal is accomplished by verifying that all programs operating in the normal state lack any `msr` instructions for writing to the PA key registers and `mov` or `pop` instructions altering the two designated general-purpose registers. Should the code scanner identify any such instructions, it alerts a warning message to the developer, indicating that employing these instruc-

tions could compromise the security guarantees of ENOLA. Therefore, the developer can revise the source code to eliminate the use of such instructions. Through this process, it is ensured that no part of the normal world can make unauthorized modifications to these critical security components. This approach has also been recognized and employed in previous studies as an accepted and adopted practice [30, 69].

This goal is achieved by verifying that all programs operating in the normal state do not contain any unauthorized instructions capable of altering critical security components. Specifically: 1) The code scanner ensures there are no `msr` instructions used for writing 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. 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. By enforcing these constraints, the code scanner ensures that no part of the normal world, including the attested program, libraries, or kernel, can make unauthorized modifications to these critical security components. This methodology has been validated and adopted as a best practice in previous research studies [30, 69], 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 quickly determine whether the destination of an indirect branch corresponds to a valid basic block. This determination is crucial for deciding whether to update $\{v_i.s, \#v_i\}$ or $\{t_i\}$ during execution. To achieve this, the ENOLA analyzer provides the capability to perform both static analysis and dynamic analysis on the instrumented binary. These analyses are used 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. When the system boots, the ENOLA attestation engine executes first, and receives a nonce c from the remote verifier (1). It then retrieves keys from secure storage and configures the PA key registers (2). During the configuration step, the attestation engine also enables the PA hardware to operate in the normal state. As previously mentioned, during the execution of the program, instrumented instructions trigger the attesta-

tion engine to record the occurrence trace T_O (④). 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. Once the execution of \mathcal{P} concludes, the attestation engine retrieves these values from the reserved registers (④). Subsequently, it generates a signature over $Auth = (T_O, \langle M_f, M_b \rangle)$ along with c , constructing the attestation report. This report is then transmitted to \mathbb{V} for verification (⑤).

Verification-time. *ENOLA Verifier:* Upon receiving the report from \mathbb{P} , \mathbb{V} initiates the verification process by authenticating the signature using c and K_a . The ENOLA verifier then abstractly executes \mathcal{P} , guided by the occurrence trace T_O , while comparing the recalculated measurements with the received values. The core component of the ENOLA verifier is a backtracking algorithm, which will be discussed in detail in §4.6.

4.4 Secure Generation of T_O

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

Non-secure callable branch destination reporting interface. The ENOLA attestation engine introduces two non-secure callable functions, denoted as `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 ENOLA attestation engine can directly infer the destination address of each branch site. In contrast, the `report_indirect` function requires the destination address to be provided as a parameter in `r0`. Both functions are annotated with the ARM Clang compiler `__attribute__((cmse_nonsecure_entry))` directive of the Cortex-M Security Extensions (CMSE) [3]. 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, as shown in Listings 6 and 7 (Appendix) 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 branch destination is preserved in the `lr` register. As shown in Listing 6, the trampoline contains only two instructions: a secure gate and a direct branch. Neither instruction modifies the `lr` register. Listing 7 further demonstrates that the ENOLA attestation engine extracts the direct branch destination ($v_i.s$) of the attested program from the `lr` register. For indirect branches, the destination is available in `r0` for the attestation engine.

```

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. Trace reporting in light green, measurement calculation in light blue

Instrumenting Reporting: Direct Branches and Loops.

For direct branches, such as those generated from `if-else` or `switch` statements in the C programming language, ENOLA’s instrumentation involves inserting “`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).

In the case of loops, the ENOLA compiler 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.

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

```

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

instrument indirect calls or jumps to capture the destination address. Unlike direct branches, 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

4.5 Secure and Efficient Calculation of Measurements

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.

```

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 function return

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 [3]. Consequently, apart from the instrumented measurement calculation instructions, the compiled REE program—including the attested program—does not utilize these two registers, ensuring that the cumulative measurements are not spilled to memory. Furthermore, because `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 libraries. In cases where any uninstrumented program uses these registers, it will restore their original values upon return. 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 stored 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.

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, as demonstrated in 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. Returns from leaf functions are directly instrumented using the `"pacg r11, lr, r11"` instruction, as shown in Line 3 of Listing 8 (Appendix). It is important to note that 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.

4.6 Backtracking Algorithm for Verification

The ENOLA verifier employs the backtracking algorithm presented in Algorithm 1 (Appendix) 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_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_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 various 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: If the last instruction in v_c is a program exit, 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 address of the next instruction 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 them against T_O . A non-zero value in the occurrence count signifies a valid transition to that target. The algorithm decrements the 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: When $v_c.e$ is a return instruction, the backward measurement is updated, and execution continues at the return target, obtained from the simulated function call stack.

5 Security Analysis

To successfully hijack the control flow and bypass ENOLA’s monitoring, an attacker must meet five key attack prerequisites: (P1): Disable or bypass the instrumented code. (P2): Influence ENOLA measurements M_f and M_b , or tamper with the measurement key K_m in the PA key registers. (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} .

P1 is mitigated under the realistic assumption of code immutability, as demonstrated by various prior works [23, 39]. Furthermore, any attempt to bypass the instrumentation or interface calls would be reflected in the measurements M_f and M_b . For P2, the measurements M_f and M_b are stored in reserved general-purpose registers, and only the instrumented `pacg` instructions are permitted to access these registers. The compiled binary is rigorously vetted by the ENOLA code scanner, which detects any unauthorized instructions that manipulate the reserved registers or the PA key registers.

An attacker may attempt to launch control-flow violation attacks by exploiting vulnerabilities in the embedded application. However, these attacks will be detected during verification through the trace and measurements. To avoid detection, the attacker would need to construct a sophisticated attack that results in hash collisions (P3). Such an attack is highly infeasible due to the low collision probability of the QARMA block cipher with a 64-bit modifier, which is approximately 2^{-60} [17]. ENOLA further complicates such attacks by chaining measurements, using the previous control-flow measurement as a modifier for the subsequent one, thereby increasing the difficulty of generating valid hash collisions.

TEE security guarantees prevent the direct manipulation of $Auth$, T_O , or K_a , as these elements are stored and signed in the secure state. Replay attacks are also thwarted by the use of a random nonce. Although M_f and M_b are stored in REE registers, the failure to achieve P2, combined with the trusted ARM PA hardware for measurement computation, ensures the integrity of the measurement chains in $Auth$. Consequently, exploiting trampoline interface calls becomes the only viable option to manipulate $Auth$ or T_O . To counter this, the ENOLA compiler prohibits any direct world-switch calls targeting the attestation engine interfaces, allowing such calls only at instrumented locations. Furthermore, to safeguard the interfaces from exploitation through indirect calls or jumps, the ENOLA compiler can adopt commonly used Software Fault Isolation (SFI) techniques [60], such as address masking, for each indirect transfer event. As a result, attackers are prevented from achieving P4, and when combined with TEE and ARM PA security guarantees, P5 is also effectively mitigated.

6 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. 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` [11] and `pyelftools` [35] libraries.

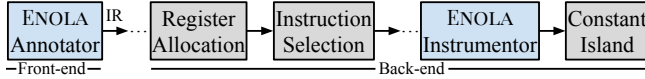


Figure 2: ENOLA passes in the LLVM pass pipeline

The ENOLA compiler is implemented by adding a front-end pass, named ENOLA Annotator, and a back-end pass, called ENOLA Instrumentor, to the LLVM embedded toolchain for ARM version 16.0.0 [14]. Additionally, we modified the class `ARMBaseRegisterInfo` to update register allocation constraints and reserve two measurement registers.

The ENOLA Annotator annotates functions and all valid direct branch destination basic blocks with LLVM metadata in the intermediate representation (IR) to enable back-end instrumentation. The ENOLA Instrumentor uses these IR annotations to instrument the basic blocks, inserting instructions to calculate the forward path measurement and the `report_direct` trampoline calls. Additionally, the back-end pass identifies indirect branches and inserts the necessary preceding instructions for `report_indirect` trampoline calls. It also inserts PA instructions for calculating the backward path measurement before function returns.

The instrumentation process addresses additional challenges when dealing with `O2` and `Oz` optimizations. Due to constraints imposed by LLVM IR phi nodes, all instrumentation is routed through the back-end ENOLA Instrumentor. Additionally, the use of `lr` as a general-purpose register requires stacking and unstacking operations, as depicted in Listing 1. Achieving non-leaf returns (via the stack) is accomplished by modifying the class `ARMFrameLowering`, where `lr` is added to the saved register list for functions annotated by the ENOLA Annotator. The LLVM compiler addresses limitations in immediate offset instructions by employing the `ARMConstantIsland` pass, which introduces nearby constant pool islands. The ENOLA Instrumentor integrates with this pass to make additional adjustments to instrumentation addresses, as illustrated in Figure 2. Moreover, the ENOLA Instrumentor leverages the completion of instruction selection and register allocation passes in the LLVM back-end pass pipeline to ensure precise and efficient instrumentation.

The ENOLA analyzer utilizes `angr` [11] to identify valid target addresses for instructions that induce 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.

7 Evaluation

7.1 Evaluation Environment

We evaluated ENOLA on the ARM Versatile Express Cortex-M prototyping FPGA system (V2M-MPS3) [2]. We configured this system as a Cortex-M85 microcontroller running at 25MHz using the AN555 FPGA image (BSP version 1.3.0) [1]. As indicated in Table 1, ENOLA is the first solution 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.

7.2 Micro-level Performance Evaluations

pacg versus software-implemented hash functions. Figure 3 presents a micro-performance analysis comparing a single execution of the `pacg` instruction for measurement with software implementations such as SHA-256 and BLAKE2s. We evaluated the software measurement implementations at both `Oz` (optimized for size) and `O2` (optimized for speed) optimization levels. As shown in the table, a `pacg` instruction consumes 12 CPU cycles, with `Oz`-optimized SHA-256 and BLAKE2s needing 5,083 and 7,939 cycles, respectively. In contrast, `O2`-optimized SHA-256 and BLAKE2s consume 4,054 and 5,768 cycles, respectively. Given BLAKE2s’ prevalence as a hash function in prior CFA solutions, a measurement calculation in ENOLA is at least 480 times faster than software-based measurements in earlier approaches.

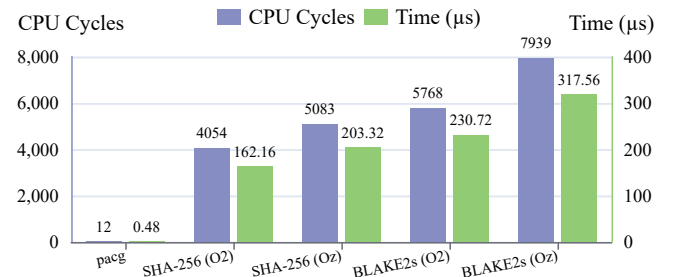


Figure 3: Execution time comparison: a single execution of the `pacg` instruction versus single runs of software-implemented hashing functions on a 25MHz Cortex-M85

ENOLA micro-level runtime overhead. Initializing PA key

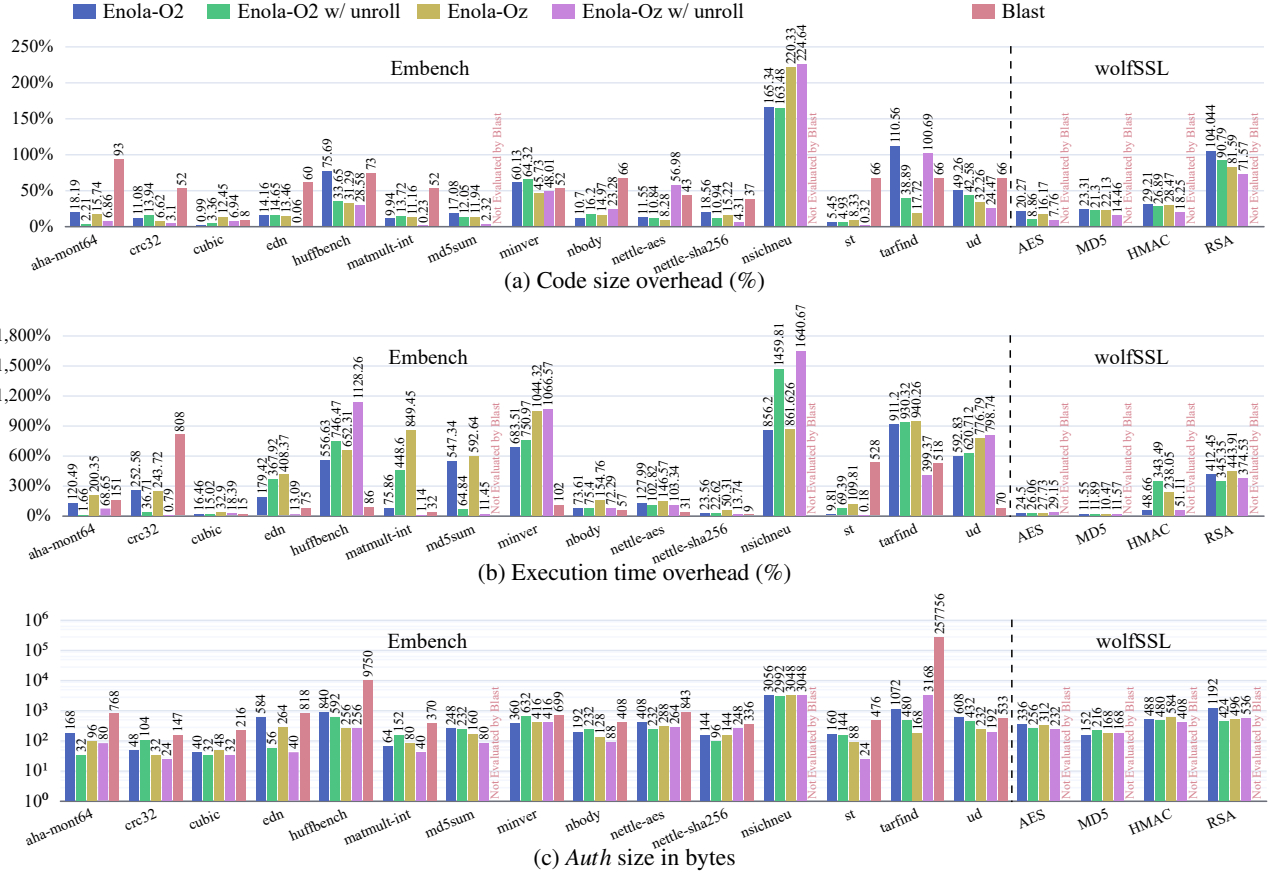


Figure 4: Code size, *Auth*, and execution time overhead comparison between ENOLA and Blast

registers consumes approximately 112 CPU cycles, whereas a single instrumented direct branch, loop body, loop exit, and indirect call takes around 67, 71, 69, and 64 CPU cycles respectively, as shown in Table 4 (Appendix). The ENOLA attestation engine utilizes a table indexed by the starting address of each basic block ($v_i.s$) to store each basic block’s occurrence trace. This structure allows for $O(1)$ time complexity when accessing and updating the occurrence trace.

7.3 Evaluations on Syringe Pump Application

Our evaluation used a version adapted by the C-FLAT [7], with minor source code adjustments to accommodate our compilation process for the Cortex-M85 microcontroller. The program consists of two main control-flow paths: the *move-syringe* path, which either dispenses or withdraws the specified bolus amount (+/-), and the *set-quantity* path, which defines the bolus amount in milliliters from user input. Figure 5 illustrates the comparison of execution overhead and *Auth* sizes for both control-flow paths.

Execution time overhead and *Auth* size. Our evaluation attested the entire program, in contrast to OAT and C-FLAT’s partial attestation and Blast’s single path evaluation. For the

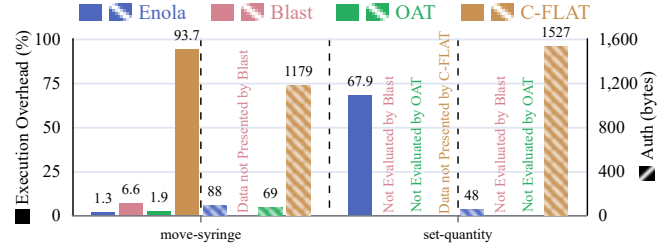


Figure 5: Execution overhead and *Auth* size comparison on syringe pump application

move-syringe path, we used bolus sizes of 0.10, 0.50, 1, and 2 ml, where ENOLA has the lowest average execution time overhead of 1.3% among the existing approaches. Note that Blast’s attestation, which operates at the function level granularity for the entire application (in contrast to ENOLA’s basic block level granularity), incurs a significantly higher average overhead of 6.6%. ENOLA generated a maximum *Auth* size of 88 bytes, a substantial reduction compared to C-FLAT’s 1,179 bytes for partial attestation. OAT has a smaller *Auth* size of 69 bytes, however it did not execute loops in the program. In the *set-quantity* path, ENOLA produces a 67.9% execution time overhead and an *Auth* size of 48 bytes. The higher execution overhead is due to the small operation of character-

```

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 code snippet: light blue loop body instrumentation

to-integer conversion within a loop. Conversely, C-FLAT’s exponential *Auth* generation leads to a maximum of 1,527 bytes for the highest input length.

Case study: anomalous behavior in the move-syringe path.

We conducted a case study on anomalous behaviors in the *move-syringe* path, similar to Blast, demonstrating ENOLA’s ability to detect such control flow violations. The anomalous path under consideration is shown in Listing 5, and involves a loop iteration for dispensing or withdrawing a specified bolus amount. The total motor steps are governed by the `steps` variable, which depends on the values `mLBolus` and `ustepsPerML` (Line 1). Here, `mLBolus` is the user input known to the ENOLA verifier and the `steps` value or loop iteration count can be statically determined since `ustepsPerML` remains constant. For instance, 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 analysis of the program binary and user input (`mLBolus`), can determine the loop body’s execution count and detect such anomalous behaviors.

7.4 Evaluations on Embench Applications

Table 5 (Appendix) presents the Lines of Code (LoC) and CFG statistics of the Embench [5] applications, along with the ENOLA instrumentation sites for each application. 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.

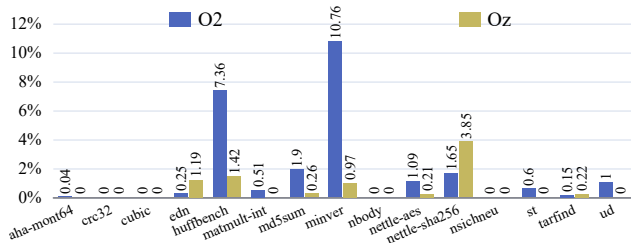


Figure 6: Code size overhead due to register reservation

Code size overhead. For O2, ENOLA with and without loop

unrolling incurs an average code size overhead of 29.71% and 38.57%, whereas Oz exhibits an overhead of 35.3% and 32.62%, respectively. This impact on code size is contingent upon the number of nodes meeting the instrumentation criteria specified by the ENOLA design. Comparing the CFG, instrumentation sites data without loop unrolling from Table 5, it’s evident that there is less instrumentation in Oz compared to O2, resulting in reduced code size overhead. However, there is an exception in *nsichneu*, where instrumentation sites are similar in both optimizations and generate a higher overhead in Oz with the same amount of ENOLA instrumented code. The ENOLA code size overhead is significantly lower compared to Blast (64%) on the Embench benchmark applications, even without including complex applications like *nsichneu*. OAT reported a 13% size overhead but it’s evaluated on small parts of the programs without loops or function calls.

We also assessed the impact on code size caused by reserving two measurement registers for ENOLA. Figure 6 shows the affect of register reservations on application code size for both optimization levels. On average, O2 results in higher overhead, as it applies more aggressive optimizations compared to Oz.

Execution time overhead. The ENOLA execution time overhead on average is comparatively lower with O2 optimization level. Even though from Table 5, we observe Oz has fewer instrumented basic blocks compared to O2, code shrinking to optimize the binary size leads to extra repetitive executions of those basic blocks and instrumented instructions. As a result, throughout all applications, we get a higher runtime overhead with Oz. For example, *st*’s O2 overhead is 9.81% with 29 direct instrumentation sites, while Oz overhead is 11 times higher even with 13 direct instrumentation sites. Further analysis indicated that the increase in trampoline invocations or instrumented code executions from 18,852 for O2 to 230,101 for Oz is the cause. Adapting loop unrolling optimization, for most cases, especially with Oz, ENOLA’s overhead decreases compared to execution without loop unrolling. For some applications such as *huffbench*, the total number of attestation control flow events significantly increases, resulting in higher overhead than without loop unrolling.

The ENOLA evaluation demonstrates a more realistic execution time overhead compared to previous works. Those solutions were evaluated on multicore Cortex-A CPUs using O0 optimizations, involved only partial attestation, and offered coarse-grained attestation capabilities. Blast reports execution time overhead of 185% with parallel execution of log commit and even though single thread execution of \mathcal{P} with log commit has an overhead of 175%, it requires function lining. Besides that Blast attestation produces verification capability at function-level granularity while ENOLA enables 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. On the common applications with Blast (excluding `md5sum` and `nsichneu`), ENOLA observes average execution time overheads of 322.20% and 278.64% for `O2` with and without loop unrolling, whereas for `Oz` the overhead is 284.54% and 431.53%, respectively. ENOLA’s average overhead is lower compared to other works of OAT and C-FLAT. OAT attests only to specific operations of a program without loops or function calls with a 2.7% overhead or 546% overhead when applied to a full program. And C-FLAT when applied to a whole program generates an overhead of 1004%. Given the above reasons and the usage of ARM PA hardware for measurement, the ENOLA execution time overhead is lower or comparable to those of prior work.

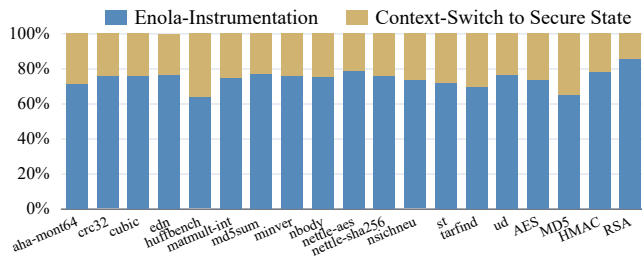
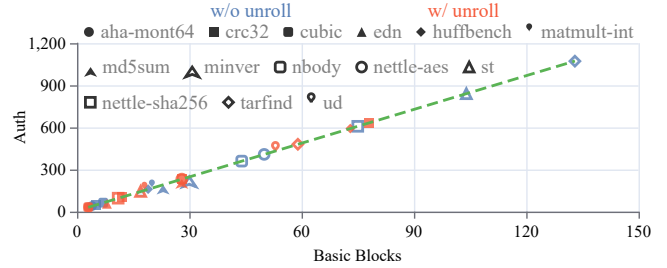


Figure 7: ENOLA runtime overhead breakdown for instrumentation (`O2`) and context-switch to secure state

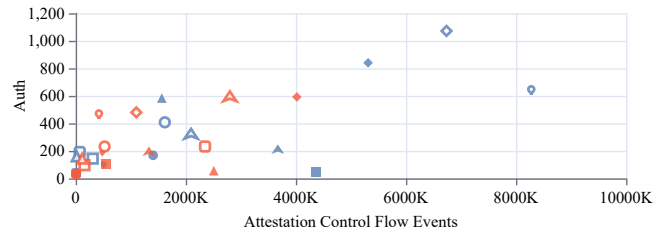
The total execution time overhead of ENOLA can be divided into two categories: the ENOLA-instrumentation overhead involving the instrumented instructions along with attestation engine trampoline execution and the context-switching overhead when transitioning to the secure state. Figure 7 shows the breakdown of these two categories for `O2` optimization level, with context-switching accounting for about 26% of the total overhead. This overhead could be eliminated by leveraging secure storage for occurrence traces in the normal state, such as through SFI techniques utilized in Blast or leveraging unprivileged load-/store instructions used in Silhouette [69].

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. In ENOLA, the *Auth* size depends on the number of unique $v_i.s$ encountered during program execution, while in Blast, repeated executions of the same basic blocks result in separate entries for each direct function call and higher *Auth* sizes. An exception to this general trend is the `ud` (`O2`) application, which exhibits a smaller *Auth* size with Blast compared to ENOLA. Further analysis revealed that `ud` contains a lot more conditional branches than function calls, leading to a smaller *Auth* size in Blast.

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 applications in Figure 8a and Figure 8b. Figure 8a illustrates the trace and measurement schemes linearly



(a) *Auth* linear to number of unique basic blocks



(b) *Auth* not affected by attestation control flow events

Figure 8: ENOLA *Auth* linear to number of basic blocks instead of attestation control flow events

aligns with basic blocks as discussed in Table 1 and Section 4.2. In Figure 8b, we observe that *Auth* does not correlate with the number attestation control flow events like existing approaches. For instance, `aha-mont64` and `edn` exhibit a similar count of control flow events (4,000K) but have 3 and 73 unique instrumented basic blocks, resulting in *Auth* sizes of 32 and 592 bytes, respectively.

7.5 Evaluations on wolfSSL Applications

We evaluated four larger applications from wolfSSL [64], a library commonly used in embedded systems.

Specifically, we examined a 128-bit AES application processing 1KB of data, the MD5 message-digest algorithm, an HMAC application generating SHA256 hashes on 1KB of data, and a 2048-bit RSA asymmetric encryption application on 16 bytes of data. Among the four applications, 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 respectively 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. These applications execute significantly more instrumented unique basic blocks compared to Embench, resulting in larger *Auth* sizes. Adapting loop unrolling optimization for applications like RSA (02) reduced the unique basic blocks, thereby shrinking the *Auth* size from 1,192 bytes to 424 bytes. However, it is evident that the *Auth* size even for larger applications in ENOLA scales linearly with the number of basic blocks.

8 Related Work

Attestation of embedded software integrity. SWATT [55] enables a remote verifier to detect instances where an attacker alters the code to perform unauthorized activities. VIPER [42] follows a similar approach to detect proxy attacks based on peripheral firmware for remote attestation frameworks. Pioneer [54] addresses the issue of verifiable code execution on untrusted legacy systems. PUFatt [41] uses physically unclonable functions combined with remote attestation to detect impersonation attacks. Armknecht et al. proposed a security framework for the analysis and design of software attestation schemes [15]. Several works also explored hardware-based remote attestation approaches for establishing a dynamic root of trust for an untrusted platform [21, 31, 53]. While the above-mentioned static approaches provide the ability to verify the code integrity of an untrusted platform, they can not detect dynamic control-flow hijacking attacks.

Attestation of control-flow. In addition to C-FLAT, OAT, and Blast, several other control-flow attestation approaches have been proposed. DIAT [8] and ARI [62] attempt to enhance the control flow attestation through modularization of the software components based on criticality, monitoring, and attesting to the module’s control or data flow. While they enhance performance, they leave sophisticated control-flow attacks undetected. ReCFA [68] and ScaRR [59] propose coarse-grained methods to reduce the number of recorded control events through program analysis, including call-site filtering, control-flow event folding, and checkpoint separation based on subpaths. ReCFA utilizes hardware memory protection a feature for critical data structures and userspace kernel trapping, which is rarely present on embedded systems. ZEKRA [27] proposed a cryptographical method to securely delegate the execution path attestation computation to a dedicated third party.

Attestation through specialized hardware. VRASED employs modified hardware for monitoring alongside software measurements to enable verification of program execution integrity [50]. SANCUS utilizes customized hardware for static attestation [49]. LiteHAX [28], Tiny-CFA [51], and LO-FAT [29] are other approaches to control-flow attestation through specialized hardware, thus unsuitable for commodity devices. IDA [12] and ISC-FLAT [48] leverage specialized hardware components to monitor various aspects such as pro-

gram counters, IRQs, memory addresses, and DMA writes, facilitating interrupt-aware attestation. In response to the TOCTOU challenge inherent in these frameworks, RATA has developed specialized hardware aimed at reducing the time gap between the attestation state and reporting [26]. ENOLA is complementary to the interrupt support or TOCTOU defense approaches and focuses on full program attestation without hardware modification.

ARM PA hardware usage beyond authentication. Several recent works leveraged ARM PA for return address protection with measurement chain, 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 PTAAuth [32], PAC it up [45], and PACMem [43]. PACStack overcomes the limitation of hash collision by constructing a measurement chain for all return addresses in the program [44]. ENOLA is the first work to utilize the PA feature for control-flow attestation on embedded systems.

9 Limitations and Future Work

To reduce runtime overhead and minimize the number of TEE switches in ENOLA, leveraging hardware trace components, such as the micro trace buffer [13], to record instruction traces could serve as a viable alternative to software-based instrumentation. Furthermore, to eliminate reliance on TEE, unprivileged load/store instructions – similar to those used in Silhouette [69] and Kage [30] – 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. Defense against TOCTOU attacks and interrupt support can be enabled on top of ENOLA utilizing prior works like RATA [26] and ISC-FLAT [48] with hardware modifications. Additionally, ENOLA currently lacks multitask attestation capabilities, such as for real-time operating system (RTOS) tasks, which remains an area for future development.

10 Conclusion

Existing control-flow attestation (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 introduced 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 mea-

surement computation. It also employs a trusted execution environment and reserves general-purpose registers to mitigate memory corruption attacks.

Acknowledgment

This material is based upon work supported in part by National Science Foundation (NSF) grants (2237238, 2329704, 2512972, 2508320, and 2422242). Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government or any agency thereof.

References

- [1] ARM FPGA Images for MPS3 and MPS2+ development boards. <https://developer.arm.com/downloads/-/download-fpga-images>.
- [2] Arm MPS3 FPGA Prototyping Board. <https://www.arm.com/products/development-tools/development-boards/mps3>.
- [3] ARMv8-M Security Extensions: Requirements on Development Tools - Engineering Specification. <https://developer.arm.com/documentation/eca0359818/latest/>.
- [4] Armv8.1-M Pointer Authentication and Branch Target Identification Extension. <https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension>.
- [5] Embench: A Modern Embedded Benchmark Suite. <https://www.embench.org/>.
- [6] Qualcomm. Pointer authentication on ARMv8.3. <https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf>, 2017.
- [7] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: control-flow attestation for embedded systems software. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.
- [8] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. Diat: Data integrity attestation for resilient collaboration of autonomous systems. In *NDSS*, 2019.
- [9] Naif Saleh Almahdhub, Abraham A Clements, Saurabh Bagchi, and Mathias Payer. μ RAI: Securing Embedded Systems with Return Address Integrity. In *Network and Distributed Systems Security (NDSS) Symposium*, 2020.
- [10] Mahmoud Ammar, Adam Caulfield, and Ivan De Oliveira Nunes. Sok: Integrity, attestation, and auditing of program execution. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 77–77. IEEE Computer Society, 2024.
- [11] Angr. <https://angr.io/>.
- [12] Fatemeh Arkanzezhad, Justin Feng, and Nader Sehabkhsh. Ida: Hybrid attestation with support for interrupts and toctou. 2024.
- [13] ARM. Armv8-m architecture reference manual. <https://developer.arm.com/documentation/ddi0553/bm/>. Online; accessed 25 March 2024.
- [14] ARM. LLVM-embedded-toolchain-for-Arm. <https://github.com/ARM-software/LLVM-embedded-toolchain-for-Arm>.
- [15] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security framework for the analysis and design of software attestation. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.
- [16] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O’Hearn, and Christian Winnerlein. BLAKE2 - fast secure hashing. <https://www.blake2.net/>, 2017.
- [17] Roberto Avanzi, Subhadeep Banik, Andrey Bogdanov, Orr Dunkelman, Senyang Huang, and Francesco Regazzoni. Qameleon v. 1.0. *A Submission to the NIST Lightweight Cryptography Standardization Process*, 2019.
- [18] 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 (ToSC)*, 2017.
- [19] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. Static detection of unsafe dma accesses in device drivers. In *USENIX Security Symposium*, 2021.
- [20] Thomas Ball and James R Larus. Efficient path profiling. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1996.
- [21] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koerberl. TyTAN: Tiny trust anchor for tiny devices. In *Annual Design Automation Conference (DAC)*, 2015.
- [22] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, 2015.
- [23] Abraham A Clements, Naif Saleh Almahdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.
- [24] Tobias Cloosters, David Paaßen, Jianqiang Wang, Ousama Draissi, Patrick Jauernig, Emmanuel Stapf, Lucas Davi, and Ahmad-Reza Sadeghi. Riscyrop: Automated return-oriented programming attacks on risc-v

- and arm64. In *International Symposium on Research in Attacks, Intrusions and Defenses(RAID)*, 2022.
- [25] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.
- [26] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. On the toctou problem in remote attestation. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.
- [27] Heini Bergsson Debes, Edlira Dushku, Thanassis Giannetos, and Ali Marandi. ZEKRA: Zero-Knowledge Control-Flow Attestation. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2023.
- [28] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. Litehax: lightweight hardware-assisted attestation of program execution. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.
- [29] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Annual Design Automation Conference (DAC)*, 2017.
- [30] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J Walls, and John Criswell. Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage. In *USENIX Security Symposium*. USENIX Association, 2022.
- [31] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. Smart: secure and minimal architecture for (establishing dynamic) root of trust. In *Network and Distributed System Security Symposium (NDSS)*, 2012.
- [32] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. PTAAuth: Temporal Memory Safety via Robust Points-to Authentication. In *USENIX Security Symposium*, 2021.
- [33] FDA. Certain Medtronic MiniMed Insulin Pumps Have Potential Cybersecurity Risks: FDA Safety Communication. <https://www.fda.gov/medical-devices/safety-communications/certain-medtronic-minimed-insulin-pumps-have-potential-cybersecurity-risks-fda-safety-communication>, 2019.
- [34] Github. Open Syringe Pump. <https://github.com/manimino/OpenSyringePump>.
- [35] Github. pyelftools. <https://github.com/eliben/pyelftools>.
- [36] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 2009.
- [37] Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. Tzmcfi: Rtos-aware control-flow integrity using trustzone for armv8-m. *International Journal of Parallel Programming*, 2020.
- [38] Beomseok Kim, Kiyong Lee, Woojin Park, Jinsung Cho, and Ben Lee. Rio: Return instruction obfuscation for bare-metal iot devices. *IEEE Access*, 2023.
- [39] Chung Hwan Kim, Taegyung Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *NDSS*, 2018.
- [40] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. Refining indirect call targets at the binary level. In *Network and Distributed System Security Symposium (NDSS)*, 2021.
- [41] Joonho Kong, Farinaz Koushanfar, Praveen K Pendyala, Ahmad-Reza Sadeghi, and Christian Wachsmann. PU-Fatt: Embedded platform attestation based on novel processor-based PUFs. In *Annual Design Automation Conference (DAC)*, 2014.
- [42] Yanlin Li, Jonathan M McCune, and Adrian Perrig. VIPER: Verifying the integrity of peripherals’ firmware. In *ACM conference on Computer and Communications Security (CCS)*, 2011.
- [43] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.
- [44] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. PACStack: an Authenticated Call Stack. In *USENIX Security Symposium*, 2021.
- [45] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *USENIX Security Symposium*, 2019.
- [46] Kangjie Lu. Practical program modularization with type-based dependence analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

- [47] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.
- [48] Antonio Joia Neto and Ivan De Oliveira Nunes. Iscflat: On the conflict between control flow attestation and real-time operations. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023.
- [49] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-cost software trusted computing base. In *USENIX Security Symposium*, 2013.
- [50] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *USENIX Security Symposium*, 2019.
- [51] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Tiny-cfa: Minimalistic control-flow attestation using verified proofs of execution. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021.
- [52] Thomas Nyman, Jan-Erik Ekberg, Lucas 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.
- [53] Bryan Parno, Jonathan M McCune, and Adrian Perrig. Bootstrapping trust in commodity computers. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.
- [54] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM Symposium on Operating Systems Principles*, 2005.
- [55] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Using software-based attestation for verifying embedded systems in cars. In *Embedded security in cars workshop (ESCCAR)*, 2004.
- [56] Ioannis Stelios, Panayiotis Kotzanikolaou, Mihalis Psarakis, Cristina Alcaraz, and Javier Lopez. A survey of iot-enabled cyberattacks: Assessing attack paths to critical infrastructures and services. *IEEE Communications Surveys & Tutorials*, 2018.
- [57] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. Oat: Attesting operation integrity of embedded devices. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.
- [58] Xi Tan and Ziming Zhao. SHERLOC: Secure and Holistic Control-Flow Violation Detection on Embedded Systems. In *ACM Conference on Computer and Communications Security (CCS)*, 2023.
- [59] Flavio Toffalini, Eleonora Losiouk, Andrea Biondo, Jianying Zhou, and Mauro Conti. Scarr: Scalable runtime remote attestation for complex systems. In *Research in Attacks, Intrusions and Defenses (RAID)*, 2019.
- [60] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1993.
- [61] Robert J Walls, Nicholas F Brown, Thomas Le Baron, Craig A Shue, Hamed Okhravi, and Bryan C Ward. Control-flow integrity for real-time embedded systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- [62] Jinwen Wang, Yujie Wang, Ao Li, Yang Xiao, Ruide Zhang, Wenjing Lou, Y Thomas Hou, and Ning Zhang. ARI: Attestation of Real-time Mission Execution Integrity. 2023.
- [63] Yujie Wang, Cailani Lemieux Mack, Xi Tan, Ning Zhang, Ziming Zhao, Sanjoy Baruah, and Bryan C. Ward. InsectACIDE: Debugger-Based Holistic Asynchronous CFI for Embedded System. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2024.
- [64] wolfSSL. wolfSSL. <https://www.wolfssl.com/>.
- [65] Nikita Yadav and Vinod Ganapathy. Whole-program control-flow path attestation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.
- [66] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, et al. Demons in the shared kernel: Abstract resource attacks against os-level virtualization. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.
- [67] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. In-kernel control-flow integrity on commodity oses using arm pointer authentication. In *USENIX Security Symposium*, 2022.
- [68] Yumei Zhang, Xinzhi Liu, Cong Sun, Dongrui Zeng, Gang Tan, Xiao Kan, and Siqi Ma. ReCFA: resilient

control-flow attestation. In *Annual Computer Security Applications Conference (ACSAC)*, 2021.

- [69] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J Walls. Silhouette: Efficient protected shadow stacks for embedded systems. In *USENIX Security Symposium*, 2020.

A ARMv8.1-M Pointer Authentication

Table 2 represents the new instructions for ARMv8.1 pointer authentication (PA) security extension. These PA instructions can generate and verify a keyed tweakable Pointer Authentication Code (PAC) for a pointer or data with the QARMA block cipher [18]. The resulting 32-bit PAC is stored in a general-purpose register. For example, the `pac` instruction signs the value in `lr` using `sp` as the tweak/- modifier and the key of the current state and privilege level and stores the result in `r12`. With the `pacg` instruction, the software can specify which registers to use. In the case of authentication failure, the authentication instructions, e.g., `aut`, generate an `INVSTATE UsageFault`.

PA Instructions	Usage
<code>pac</code>	Sign <code>lr</code> using <code>sp</code> as the modifier and store the resulting PAC in <code>r12</code>
<code>aut</code>	Authenticate <code>lr</code> using <code>sp</code> as the modifier and validate with the PAC in <code>r12</code>
<code>pacg rd, rn, rm</code>	Sign a general-purpose register <code>rn</code> using <code>rm</code> as the modifier and store the PAC in <code>rd</code>
<code>autg rd, rn, rm</code>	Authenticate a general-purpose register <code>rn</code> using <code>rm</code> as modifier and validate with the PAC in <code>rd</code>
<code>bxaut rd, rn, rm</code>	Authenticate <code>rn</code> using <code>rm</code> as the modifier and <code>rd</code> as the PAC. If validated, branch to <code>rn</code> .

Table 2: The ARMv8.1-M PA extension instructions. ENOLA utilizes `pacg` for measurement calculations.

Table 3 displays the four 128-bit PA key registers, indicating their usage and configurable settings. Each 128-bit PA key consists of four 32-bit registers, which are not memory mapped. For example, `pac_key_u_ns` register is made up of `pac_key_u_ns_0`, `pac_key_u_ns_1`, `pac_key_u_ns_2`, and `pac_key_u_ns_3` registers. When a `pacg` instruction is executed at the unprivileged level and non-secure state, the `pac_key_u_ns` register is implicitly used as the key to compute the PAC. As denoted in the table, this PA key register can only be configured by software in the secure state and set individually for each 32-bit internal register using privileged `msr` (move-to-system-register) instructions.

PA Key Registers	Used at	Configurable at
<code>pac_key_u_ns</code>	U-NS	P-NS and P-S
<code>pac_key_p_ns</code>	P-NS	P-NS and P-S
<code>pac_key_u_s</code>	U-S	P-S
<code>pac_key_p_s</code>	P-S	P-S

U-NS: unprivileged non-secure, P-NS: privileged non-secure, U-S: unprivileged secure, P-S: privileged secure.

Table 3: The 128-bit PA key registers (not memory mapped) and their usage and configuration settings.

B Trampoline and ENOLA Attestation Engine functions

Listing 6 illustrates an ENOLA trampoline cross-state call that requires two additional instructions in the non-secure callable state compared to a normal function call. While Listing 7 presents first instruction in the secure state trampoline of attestation engine to retentive the branch destination $v_i.s$ to `r0` general purpose register.

```

;Non-secure callable memory region
report_direct:
0x101FFE00 sg
0x101FFE04 b report_direct <0x100031D4>

```

Listing 6: Non-secure callable region trampoline for reporting direct branches

```

;Secure memory region
report_direct:
0x100031D4 mov r0, lr
0x100031D8 ...

```

Listing 7: ENOLA attestation engine retrieving attested program’s direct branch destinations

C Instrumentation for Leaf function return

For leaf functions, the return address is stored in the `lr` register, allowing straightforward instrumentation to construct the backward edge measurement chain M_b with a single instruction: "`pacg r11, lr, r11`", as shown in Listing 8. Here, the leaf and non-leaf function returns are chained together to construct M_b on the reserved `r11` register.

D Verification Algorithm Details

The ENOLA verifier uses the backtracking algorithm listed in Algorithm 1 to verify the legitimacy of the attested control path. The algorithm takes the following inputs: the attestation report \mathcal{R} and the control-flow graph G_ϕ of the attested program, incorporating the entry basic block (\mathcal{P}_{entry}) and the potential exit points (\mathcal{P}_{exits}).

```

1  epilogue:
2  add sp, #0x8
3  pacg r11, lr, r11 ;update measurement M_b
4  bx lr

```

Listing 8: Instrumentation example: leaf function return

The algorithm initially confirms the validity of the attestation report signature $Sig_{K_a}(Auth, c)$ (Line 2-3) and checks for any illegal indirect branch targets (Line 4-5). Upon successful verification, the algorithm abstractly executes the program based on its CFG and verifies the forward and backward measurements. The abstract execution starts at \mathcal{P}_{entry} and recursively executes various branches. The abstract execution initializes an empty simulated call stack (S) to keep track of return addresses.

The algorithm executes one of four actions based on the type of the last instruction at address $v_c.e$ within the current basic block (v_c): (1) if the last instruction in v_c signifies a program exit, this marks the end of \mathcal{P} . Here, the computed measurements are compared with the received values in $Auth$ (Line 12); (2) for a function call at $v_c.e$, the verifier pushes the address of the subsequent instruction onto the simulated function call stack S and proceeds to abstractly execute the call target (Line 14-15); (3) at $v_c.e$, in the case of a conditional branch, the algorithm explores all potential paths emanating from the current basic block (Line 17). And the occurrence count for each target in T_O is checked. If the occurrence count is greater than zero, the verifier decreases this count by one, computes a measurement (M'_f), and transitions abstract execution to that target. An invalid path is identified if no such targets exist in T_O , prompting the recursive function to backtrack; (4) when $v_c.e$ is a return instruction, the backward edge measurement (M'_b) is updated, and abstract execution moves to the return target from the simulated function call stack (Line 23-24).

E ENOLA micro-level runtime overhead

	CPU cycles	Time (μ s)
Init. PA key registers	112	4.48
Instrumented direct branch	67	2.79
Instrumented loop body	71	2.84
Instrumented loop exit	69	2.76
Instrumented indirect call	64	2.56

Table 4: ENOLA micro-level runtime overhead

Table 4 provides the micro-level runtime overheads for various ENOLA operations. The ENOLA attestation engine initializes the PA key in approximately 4.8 μ s. Runtime overheads for instrumentation at specific sites are as follows: 2.79 μ s for direct branches, 2.84 μ s for loop entries, 2.76 μ s for loop exits, and 2.56 μ s for indirect calls.

Algorithm 1: ENOLA verifier

Input: $G_{\mathcal{P}} = (V, E)$
 $\mathcal{R} = (Auth, Sig_{K_a}(Auth, c))$, where
 $Auth = (T_O, \langle M'_f, M'_b \rangle)$ and
 $T_O = \{\{v_i.s, \#v_i\} | i = 0, \dots, |V| - 1 \wedge \#v_i \neq 0\}, \{t_i | t_i \notin \bigcup_{i=0}^{|V|-1} v_i.s\}$
Output: The legitimacy of the control-flow path

```

1 Procedure Verify( $\mathcal{R}$ ):
2   if  $Sig_{K_a}(Auth, c)$  is not valid then
3     return False;
4   if  $|\{t_i\}| \neq 0$  then
5     return False;
6   return AbstractExec( $\mathcal{P}_{entry}, T_O, 0, 0, \emptyset$ );
7 Procedure AbstractExec( $v, T, M'_f, M'_b, S$ ):
8    $v_c \leftarrow v; M''_f \leftarrow M'_f; M''_b \leftarrow M'_b;$ 
9    $T' \leftarrow T; S' \leftarrow S;$ 
10  if  $v_c.e \in \mathcal{P}_{exit}$  then
11     $M''_f \leftarrow H_{K_m}(v_c.s, M'_f);$ 
12    return  $M_f == M''_f \wedge M_b == M''_b;$ 
13  else if  $v_c.e$  is a function call then
14     $S'.push(v_c.e + 1);$ 
15    return AbstractExec( $e_{v_c}.d, T', M''_f, M''_b, S'$ );
16  else if  $v_c.e$  is not a function return then
17    foreach  $d_i \in \cup e_{v_c}.d$  do
18      if  $T'.\#d_i > 0$  then
19         $T'.\#d_i \leftarrow T'.\#d_i - 1;$ 
20         $M''_f = H_{K_m}(d_i, M'_f);$ 
21        return AbstractExec( $d_i, T', M''_f, M''_b, S'$ );
22  else if  $v_c.e$  is a function return then
23     $M''_b = H_{K_m}(S'.top(), M'_b);$ 
24    return AbstractExec( $S'.pop(), T', M''_f, M''_b, S'$ );
25  return False;

```

F Evaluated application's CFG and ENOLA instrumentation sites

Table 5 summarizes the Lines of Code (LoC), CFG statistics, and ENOLA instrumentation sites for the Embench and wolfSSL applications at both O2 and Oz optimization levels. Unlike prior control-flow attestation studies, ENOLA was evaluated on comparatively larger applications, notably the wolfSSL applications, using real-world compiler optimization levels.

G Case Study on the crc32 Embench Application

We conducted a case study on the Embench application performing a Cyclic Redundancy Check named `crc32` to illus-

Application	LoC	CFG Statistics				ENOLA Instrumentation Sites					
		Nodes		Edges		Direct		Indirect		Returns	
		Oz	Oz	Oz	Oz	Oz	Oz	Oz	Oz	Oz	Oz
Embench Applications											
aha-mont64	162	875	898	1,399	1,422	27	8	0	10	9	9
crc32	291	385	414	289	303	25	14	0	0	17	13
cubic	254	431	474	516	533	11	21	0	11	6	6
edn	359	401	415	401	331	47	28	0	0	13	13
huffbench	309	653	481	793	455	230	71	0	0	16	15
matmult-int	175	369	406	288	317	23	20	0	0	10	10
md5sum	153	409	415	343	330	41	25	0	0	15	15
minver	187	517	466	580	432	133	58	0	0	7	7
nbody	172	554	444	824	424	35	23	0	0	7	7
nettle-aes	1,147	440	461	412	398	73	50	0	0	14	14
nettle-sha256	422	437	435	395	377	51	30	0	0	10	10
nsichneu	2,676	1,102	1,262	1,765	2,170	769	765	0	0	6	5
st	117	856	438	1,723	404	29	13	0	0	13	12
tarfind	81	590	417	684	337	229	28	0	0	16	13
ud	95	435	403	417	324	82	28	0	0	6	6
wolfSSL Applications											
AES	10,646	1,038	1,087	1,648	1,702	90	87	0	0	23	26
MD5	4,917	1,194	1,241	2,002	1,981	196	169	0	0	88	70
HMAC	6,085	1,374	1,426	2,425	23,83	294	257	0	0	108	83
RSA	27,807	5,480	3,444	9,813	6,110	3,309	1,476	0	18	272	267

Table 5: ENOLA instrumentation sites, Lines of Code (LoC) and application’s CFG statistics.

trate the functionality of the ENOLA framework. The source code of the program, depicted in Listing 9, reveals two main functions: the `benchmark_body` parent function, which contains a loop that iteratively calls `crc32pseudo` based on the `rpt` value and the callee executes a loop 1,024 times to perform CRC operations.

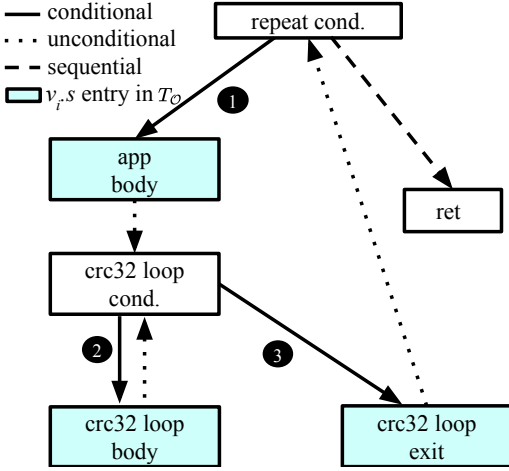


Figure 9: Simplified CFG of `crc32` Embench application (Oz)

For the Oz optimization level, the ENOLA compiler generates and instruments two basic blocks for loop body branches and one for loop exit branch, as highlighted in Listing 9. Figure 9 presents a simplified CFG of the program, where the entry address of highlighted nodes or basic blocks are designated as $v_i.s$. Specifically, edges marked as ❶ and ❷ represent loop

```

1  DWORD  crc32pseudo () {
2      int  i; register DWORD oldcrc32;
3      oldcrc32 = 0xFFFFFFFF;
4      for (i = 0; i < 1024; ++i) { //(2) crc loop-body
5          → branch
6          oldcrc32 = UPDC32 (rand_beebs (), oldcrc32);
7      }
8      return ~oldcrc32; //(3) crc loop-exit branch
9  }
10 static int __attribute__ ((noinline))
11 benchmark_body (int rpt) {
12     int i; DWORD r;
13     for (i = 0; i < rpt; i++) { //(1) App loop-body
14         → branch
15         srand_beebs (0);
16         r = crc32pseudo ();
17     }
18     return (int) (r % 32768); //App sequential
19     → return
20 }

```

Listing 9: Source code of `crc32` Embench application

body control flow changes, while ❸ denotes the loop exit edge in the `crc32pseudo` function. Despite the presence of 303 edges in the CFG, only these three edges are executed at runtime that meet the criteria to be included in the trace. Consequently, only the $v_i.s$ of the three highlighted basic blocks are included in the occurrence trace T_O , resulting in a generated size of 24 bytes. Table 6 illustrates the contents of the generated T_O when the application is executed with the ENOLA framework. The *Auth* consists of T_O combined with M_f and M_b measurements, which provide enough evidence for \forall to validate \mathbb{P} ’s execution control-flow of the program \mathcal{P} using ENOLA verification Algorithm 1.

Address ($v_i.s$)	Occurrence Count ($\#v_i$)
0x10000495	4,250
0x10000441	4,352,000
0x10000465	4,250

Table 6: T_O content of `crc32` with ENOLA

H Instruction Operands in T_O

Table 7 shows the instructions meeting the criteria for ENOLA target basic block instrumentation and their corresponding elements or entries in T_O . The backward edges are enclosed in the measurement only, thus not included in T_O .

Instruction Type	Instruction	T_O entry
Indirect Call	<code>blx rx</code>	<code>rx</code>
Indirect Jump	<code>bx rx</code>	<code>rx</code>
Conditional Jump	<code>b.{condition} #addr</code>	<code>#addr</code>
Conditional Jump	<code>cb.{condition} #addr</code>	<code>#addr</code>

Table 7: Instructions targeting ENOLA’s instrumented basic blocks and corresponding T_O entry