EILID: Execution Integrity for Low-end IoT Devices

Sashidhar Jakkamsetti
Robert Bosch LLC
sashidhar.jakkamsetti@us.bosch.com

Youngil Kim
UC Irvine
youngik2@uci.edu

Andrew Searles
UC Irvine
searlesa@uci.edu

Gene Tsudik
UC Irvine
gene.tsudik@uci.edu

***Abstract*—Prior research yielded many techniques to mitigate software compromise for low-end Internet of Things (IoT) devices. Some of them detect software modifications via remote attestation and similar services, while others preventatively ensure software (static) integrity. However, achieving run-time (dynamic) security, e.g., control-flow integrity (CFI), remains a challenge.**

Control-flow attestation (CFA) is one approach that minimizes the burden on devices. However, CFA is not a real-time countermeasure against run-time attacks since it requires communication with a verifying entity. This poses significant risks if safety- or time-critical tasks have memory vulnerabilities.

To address this issue, we construct EILID – a hybrid architecture that ensures software execution integrity by actively monitoring control-flow violations on low-end devices. EILID is built atop CASU [1], a prevention-based (i.e., active) hybrid Root-of-Trust (RoT) that guarantees software immutability. EILID achieves fine-grained backward-edge and function-level forward-edge CFI via semi-automatic code instrumentation and a secure shadow stack.

I. Introduction

A recent report [2] states that there are now over 15 billion IoT devices worldwide, and this number is predicted to double by 2030. These devices are deployed in a wide range of settings, including homes, offices, farming, factories, public venues, and vehicles. They also often collect sensitive information and/or perform safety-critical tasks. Unlike higher-end computing devices (e.g., laptops, tablets, and smartphones), IoT devices generally lack robust security features due to cost, size, energy, and performance constraints, making them attractive attack targets. Recent surveys demonstrate that IoT devices are subject to numerous attacks through physical, network, software, and cryptographic vulnerabilities [3]–[7].

To that end, various research proposed Roots-of-Trust (RoTs) for low-end devices [8]–[21]. One popular technique underlying these proposals is Remote Attestation ($\mathcal{R}$A) [8]–[14], a well-established security service that **detects** malware presence on an untrusted remote device. $\mathcal{R}$A requires a device (prover or $\mathcal{P}$rv) to securely interact with a remote trusted party (verifier or $\mathcal{V}$rf), conveying the current software state of the former to the latter. An alternative approach, exemplified by CASU [1], **prevents** all software modifications except secure updates. However, all these detection- and prevention-based techniques focus on **static** software integrity and do not provide any protection against, or detection of, run-time attacks.

Since run-time attacks do not modify the software code itself, conventional $\mathcal{R}$A schemes can not handle them. To mitigate such attacks, many Control-Flow Attestation (CFA) [22]–[32] and Data-Flow Attestation (DFA) [33]–[36] schemes have been proposed. They allow $\mathcal{V}$rf to check execution integrity of remote $\mathcal{P}$rv-s at run-time. However, merely detecting run-time attacks is insufficient, particularly in safety-critical environments where real-time response and recovery are imperative. Furthermore, practicality of CFA itself is questionable since it requires storing – and later transferring – potentially voluminous logs.

Another line of research [37]–[53] produced various Control-Flow Integrity (CFI) techniques for embedded systems. However, all such efforts target higher-end devices with ample computing resources (e.g., multiple cores and many MBs of memory) and advanced security features, such as Memory Protection Units (MPUs), Memory Management Units (MMUs), or Trusted Execution Environments (TEEs), making them unsuitable for low-end devices.

To bridge this gap, this work constructs a real-time countermeasure, EILID, against control-flow attacks on low-end devices. EILID uses CASU as a foundational platform to offer user software immutability. It extends CASU by implementing CFI monitor that ensures fine-grained backward-edge and function-level forward-edge execution integrity. EILID has three components: (1) *Code instrumenter* – inserts additional instructions into device software at compile-time to jump to CFI monitor, (2) *CFI monitor* – trusted (minimal) software that maintains a secure shadow stack for validating crucial control-flow metadata, and (3) *Secure hardware* – derived from CASU, it detects control-flow violation and triggers a reset.

Contributions of this work are twofold:

- To the best of our knowledge at the time of this writing, EILID is the first RoT architecture enforcing CFI on low-end devices, given that relevant prior work either focused on detecting such attacks [22]–[36] or targeted higher-end systems [37]–[53].
- A prototype implementation of EILID on openMSP430, which includes code instrumentation at compile-time and trusted software running in secure ROM. We also evaluate EILID performance, showing that it has a very low average run-time overhead. All source code is publicly available at [54].

II. Background & Related Work

A. Scope of Low-end Devices

IoT devices vary greatly in computing power. For example, low-end devices (e.g., smart plugs or light bulbs) feature a single-core MCU and minimal amount of memory (e.g., a few KBs), while high-end devices (e.g., car infotainment systems)

978-3-9826741-0-0/DATE25/© 2025 EDAA

host multi-core MCUs and extensive memory (e.g., a few GBs). In this work, we focus on the former.

A typical low-end MCU has an 8- or 16-bit Von Neumann architecture, running at $\leq$ 48MHz with $\leq$ 64KB of memory, e.g., TI MSP430 or AVR ATMega32. SRAM is used as data memory (DMEM), typically ranging from 4KB to 16KB, while other address space is available for program memory (PMEM). Such devices usually run software atop "bare metal" with no memory management support, such as MMUs or MPUs. Therefore, there is neither memory isolation nor privilege guarantees.

B. Roots-of-Trust (RoTs) & CASU

Related work on RoT-s for low-end devices can be classified into: *passive* and *active* designs. The former [8]–[11] detect malware presence on $\mathcal{P}$rv via $\mathcal{R}$A. This involves $\mathcal{P}$rv generating (upon a challenge from $\mathcal{V}$rf) a cryptographic proof of its PMEM state and returning the result to $\mathcal{V}$rf, which decides/detects whether $\mathcal{P}$rv is compromised. Besides being passive, $\mathcal{R}$A imposes a non-negligible computational effort on $\mathcal{P}$rv. Whereas, **Active RoTs** [1], [16]–[18] continuously monitor $\mathcal{P}$rv behavior to prevent or minimize the impact of compromise.

EILID is built atop an active RoT, ***CASU*** [1] – a hybrid hardware/software architecture that enforces run-time software immutability, while supporting authenticated software updates. It defends against code injection attacks (i.e., attacks that insert executable code by exploiting memory vulnerabilities [55]) by preventing unauthorized modifications of PMEM and any code execution from DMEM. The only means to modify PMEM is through secure updates. To achieve this, CASU monitors several CPU hardware signals and triggers a reset if any violation is detected. CASU obviates the need for $\mathcal{R}$A between software updates, with minimal hardware modifications.

C. Control-Flow Attacks

Due to the lack of support for advanced memory in languages such as C/C++ and assembly (which are widely used in embedded systems software), devices are often vulnerable to control-flow attacks. There are two types of such attacks: code injection and code reuse. The former inserts executable code by exploiting memory vulnerabilities in the user software, e.g., stack overflow [55]. These attacks can be thwarted by a simple memory protection policy, $W \oplus X$, preventing any execution from writable memory. Code reuse attacks, on the other hand, deviate the software control flow to execute arbitrary (malicious) sequences of existing code. Relying on the target they aim to alter, control-flow attacks can be divided into forward- and backward-edge attacks. The former manipulates forward edges in the Control-Flow Graph (CFG), such as indirect function calls and jumps, while backward-edge attacks alter edges going backward, such as a return address.

There is a large body of research [37]–[44], [56]–[59] guaranteeing CFI on various embedded platforms. Some [37], [38] extend an instruction set architecture with new CFI instructions. Another CFI approach [40], [41], [43] relies on advanced features (e.g., MPUs, MMUs, and TEEs) to offer

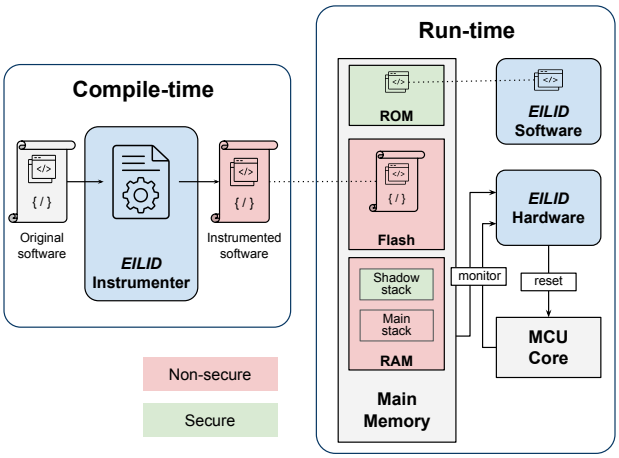


Fig. 1: EILID Design Overview

CFI. Nonetheless, since all prior techniques are geared for higher-end devices, they are not applicable to "bare-metal" devices that we target.

Another line of research [22]–[33] focuses on CFA using $\mathcal{R}$A. CFA records a log of the control-flow path taken by the software (on $\mathcal{P}$rv). Upon receiving CFA request from $\mathcal{V}$rf, $\mathcal{P}$rv generates an unforgeable integrity proof of this log and returns it to $\mathcal{V}$rf. Despite being computationally efficient for $\mathcal{P}$rv (i.e., no validation/real-time protection), CFA only detects (does not prevent) control-flow deviations at run-time. It also incurs significant log storage and transmission costs. Moreover, log verification on $\mathcal{V}$rf is not trivial as it depends on the complexity of the code and the number of $\mathcal{P}$rv-s. As shown in Table I, EILID is the first RoT construction that offers real-time CFI protection for low-end devices.

III. Overview & Assumptions

A. EILID Overview

As shown in Figure 1, EILID has three components (colored blue): instrumenter EILID$_{inst}$, trusted software EILID$_{sw}$, and hardware EILID$_{hw}$. We focus on the first two, since EILID$_{hw}$ is based on CASU hardware and does not require any modification, except for the secure memory extension reserved for the shadow stack. EILID$_{inst}$ instruments device software (which potentially has memory vulnerabilities) at compile-time to produce a CFI-aware instrumented binary. This binary is installed on EILID-enabled device, where EILID$_{hw}$ continuously monitors software behavior and resets the device in case of CFI violations. At run-time, the software control flow is continuously validated by EILID$_{sw}$ with the shadow stack support. Note that EILID$_{sw}$ is immutable since it is housed in ROM. Furthermore, the shadow stack is only accessible by EILID$_{sw}$, ensuring that control-flow metadata remains protected.

B. Threat Model

In line with other state-of-art CFA/CFI schemes [56]–[59], we assume a powerful external adversary ($\mathcal{A}$dv) with comprehensive knowledge of all software running on the device, including all memory vulnerabilities, if any. Since much of

Method	Work	RT	F-edge	B-edge	Interrupt	Platform	Technique Summary
CFI	HAFIX [37]	✓		✓		Intel Siskiyou Peak	Extends Intel ISA with shadow stack
	HCFI [38]	✓	✓	✓		Leon3	Extends Sparc V8 ISA with shadow stack and labels
	FIXER [39]	✓	✓	✓		RocketChip	Extends RISC-V ISA with shadow stack
	Silhouette [40]	✓	✓	✓	✓	ARMv7-M	Uses ARM MPU for hardened shadow-stacks and labels
	CaRE [41]	✓		✓		ARMv8-M	Uses ARM TrustZone for shadow stack & nested interrupts
CFA	Tiny-CFA [23]		✓	✓		openMSP430	Hybrid CFA with shadow stack
	ACFA [24]		✓	✓	✓	openMSP430	Active hybrid CFA with secure auditing of code
	LO-FAT [25]		✓	✓		Pulpino	Hardware-based CFA solution
	CFA+ [26]		✓	✓	✓	ARMv8.5-A	Leverages ARM's Branch Target Identification
EILID		✓	✓	✓	✓	**opemMSP430**	**Uses CASU for shadow stack**

TABLE I: CFA and CFI Techniques from Prior Work (RT: Real-time protection, F: Forward, B: Backward)

embedded software is written in memory-unsafe languages (e.g., C, C++, or Assembly), we assume that it likely has (presumably unintentional) control-flow bugs.

$\mathcal{A}$dv can arbitrarily access any executable memory location at run-time. It can also tamper with any data (e.g., return addresses, function pointers, and indirect function calls) on the stack and heap, thereby manipulating program control flow. Moreover, $\mathcal{A}$dv can exploit vulnerabilities in Interrupt Service Routines (ISRs) to alter the program context (e.g., an ISR return address), thus forcing a deviation from the legal control flow. As common in most related work, physical attacks and side-channels are out of scope. Furthermore, EILID does not address non-control-data attacks. Protection against them is quite difficult and incurs prohibitive overhead for resource-constrained devices. Section VII contains more information about this issue.

C. Security Goals

To enable EILID, the following security properties must hold.
[P1] Return Address Integrity: Function return addresses should be protected. Any attempt to overwrite the return address, leading to a transition to an illegal (unintended) address, must be detected and handled properly.
[P2] Return-from-Interrupt Integrity: When an interrupt is triggered, the current program context is saved on the main stack, and the system branches to the appropriate ISR. After serving the interrupt, the stored context information is retrieved and execution is resumed where it left off. A memory vulnerability in an ISR allows modifications of the main stack where the context is kept, causing it to return to an illegal address. Therefore, the interrupt context stored on the main stack must remain intact while the ISR runs.
[P3] Indirect Call Integrity: Unlike a direct branch which explicitly specifies the next instruction address, an indirect call is invoked with the pointer containing the destination address. Thus, that address can not be resolved at compile-time since it is unknown until the value is loaded from memory at run-time.

Besides these three properties, complete CFI also requires integrity against indirect jumps. However, indirect jumps are generally not critical because in practice they are only used for `switch` statements. Jump-tables, generated to store indirect jump pointers for these statements, can be avoided by using compiler flags such as '`-fno-jump-tables`' and '`-fno-switch-tables`'. More details are in Section VII.

Platform	Instructions			
	Call	Return	Return from Interrupt	Indirect Call
TI MSP430	`CALL`	`RET`	`RETI`	`CALL`
AVR ATMega32	`CALL`	`RET`	`RETI`	`RCALL, ICALL`
Microchip PIC16	`CALL`	`RETURN`	`RETFIE`	`CALL, RCALL`

TABLE II: Instruction Set in Low-end Platforms

IV. EILID Design: EILID$_{inst}$ & EILID$_{sw}$

As stated in Section III-A, we use CASU hardware as EILID$_{hw}$ without significant modifications. Consequently, we avoid introducing any new hardware overhead and preserve CASU's formally verified properties. Figure II shows instruction sets that can be used by EILID$_{inst}$ on popular low-end MCU platforms.

A. EILID$_{inst}$

EILID$_{inst}$ analyzes and instruments assembly code at compile-time. For backward-edge CFI, it discovers every function call statement (e.g., `call` in MSP430 and ATMega32) and adds a few instructions to jump to EILID$_{sw}$ (**P1**). At run-time, before each function call site, the instrumented code resolves its return address (i.e., the next address of the function call site) and invokes EILID$_{sw}$ to store the address on the shadow stack.

EILID$_{inst}$ also detects all function return instructions (e.g., `RET` in MSP430 and `RETURN` in PIC16). It sequentially instruments the code to execute EILID$_{sw}$, verifying that the return address in the current function context matches the one stored on the shadow stack.

Similarly, EILID$_{inst}$ instruments the binary at ISR prologues (entry points) to store interrupt context metadata (e.g., return address and status register) on the shadow stack (**P2**). Meanwhile, at ISR epilogues, (i.e., right before return), instrumented code retrieves and verifies stored context meta against the current context. EILID$_{inst}$ discovers ISR prologues by their reserved names, while ISR epilogues are identified by return-from-interrupt instructions, e.g., `RETI` in MSP430 and `RETFIE` in PIC16.

Moreover, EILID$_{inst}$ enumerates entry points of all functions and puts them into a table for forward-edge CFI (**P3**). EILID$_{inst}$ discovers indirect call instructions (e.g., `CALL` in

MSP430 and `RCALL, ICALL` in ATMega32) and introduces a few extra instructions to verify the legitimacy of function addresses. Note that EILID achieves function-level forward-edge CFI because $\mathcal{A}$dv can redirect the indirect function call to another valid address in the function entry table. However, we believe that the chances of that are low due to the small number of functions in a typical low-end device software.

B. EILID$_{sw}$

At run-time, EILID$_{sw}$ is invoked to store and validate function or ISR return addresses provided by EILID$_{inst}$ as function arguments (**P1** & **P2**). Before executing an indirect function call, EILID$_{sw}$ is triggered to verify the legitimacy of the function by searching for it in the table (**P3**). If any CFI validations fail, EILID$_{hw}$ resets the device, thus thwarting control-flow attacks.

Registers	Description
r4	Used as an argument of `S_EILID_init()`
r5	Used as a pointer to the shadow stack's current index
r6, r7	Used as an argument of other `S_EILID` functions

TABLE III: Reserved Registers for EILID

V. Implementation

EILID is implemented atop openMSP430 [60], an open-sourced 16-bit MCU core. The code is synthesized using Xilinx Vivado 2023.1, and the synthesized design is then deployed on a Basys3 Artix-7 FPGA board for prototyping and evaluation. All EILID source code is open-sourced at [54].

EILID$_{inst}$ is a Python script with $\approx$ 200 lines of code. To facilitate instrumentation, general-purpose user registers, `r4-r7` are reserved for EILID, which are seldom used. Their usage is summarized in Table III. If any of those registers are in use by device software, merely two instructions are additionally needed: (1) push the register value on the main stack before its usage, e.g., `push r4`, and (2) pop the value from the stack after it is used, e.g., `pop r4`.

We allocate 256 bytes of secure DMEM (realized by EILID hardware) for the shadow stack, exclusive to EILID$_{sw}$. It can store $\leq$ 128 return addresses and the interrupt context. Since MSP430 devices usually have tens of KB of memory, reserving 256 bytes for EILID should be acceptable. Also, the memory region that stores the function return address is freed up and ready for re-use once the current function returns. Therefore, we believe it can accommodate control-flow metadata of all tasks running on typical low-end commodity devices. Nevertheless, the shadow stack size is configurable based on memory constraints and software complexity.

A. EILID$_{inst}$

Figure 2 shows the iterated compile process with EILID$_{inst}$. It takes as input two files: (1) `*.lst` to discover function return addresses, and (2) `*.s` to be instrumented. Its output is `*instr.s` file, which is colored red in the iterations. Three iterations are needed due to the input file `*.lst`, generated after the build is completed, and shifted addresses in the second

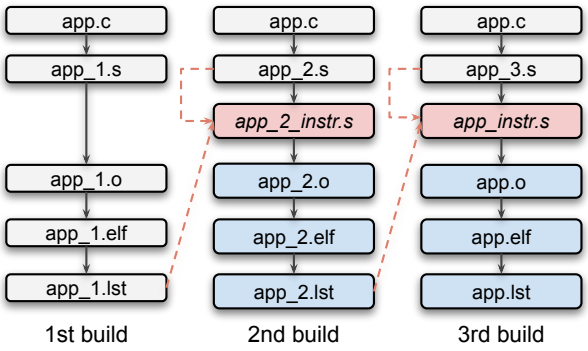


Fig. 2: EILID Instrumented Compilation

```
1
2
3   call #foo
```

(a) Original

```
1   mov #e200, r6
2   call #NS_EILID_store_ra
3   call #foo
```

(b) Instrumented

Fig. 3: Instrumentation before Function Call

```
1   foo:
2       ...
3
4
5       ret
```

(a) Original

```
1   foo:
2       ...
3       mov @(r1), r6
4       call #NS_EILID_check_ra
5       ret
```

(b) Instrumented

Fig. 4: Instrumentation before Function Return

loop after the first-iteration instrumentation. This adjustment is essential because the return address of each function changes from the instructions introduced during the second iteration. For example, if the return address of `foo()` is initially 0x200 (`app_2.s`), it could be modified to 0x210 if 8 instructions are added ahead of it in `app_2_instr.s` file. This also results in increased file sizes (colored red and blue). Because `app_2.lst` already shifted addresses, it is directly used as input of EILID$_{inst}$ at the last loop with no further changes.

Figures 3 and 4 show the instrumentation needed to achieve **P1**. Before jumping to the function, its next instruction address (0xe200, the return address of `foo()`) is loaded to `r6` as an argument of `NS_EILID _store_ra()`. For example, if the function call address is 0x100, its return address would be 0x102 or 0x104, depending on its instruction size. Then, it invokes `NS_EILID_store_ra()` to store this address on the

```
1   ISR:
2
3
4
5       ...
```

(a) Original

```
1   ISR:
2       mov 0(r1), r6
3       mov -2(r1), r7
4       call #NS_EILID_store_rfi
5       ...
```

(b) Instrumented

Fig. 5: Instrumentation at ISR Entry point

```
1   ISR:
2       ...
3
4
5
6       reti
```

(a) Original

```
1   ISR:
2       ...
3       mov 0(r1), r6
4       mov -2(r1), r7
5       call #NS_EILID_check_rfi
6       reti
```

(b) Instrumented

Fig. 6: Instrumentation before ISR Return

```
1   main:
2
3
4
5
6       ...
```

(a) Original

```
1   main:
2       mov #e400, r6
3       call #NS_EILID_store_ind
4       mov #e500, r6
5       call #NS_EILID_store_ind
6       ...
```

(b) Instrumented

Fig. 7: Instrumentation at Main Function Entry point

```
1
2
3   call r13
```

(a) Original

```
1   mov r13, r6
2   call #NS_EILID_check_ind
3   call r13
```

(b) Instrumented

Fig. 8: Instrumentation before Indirect Function Call

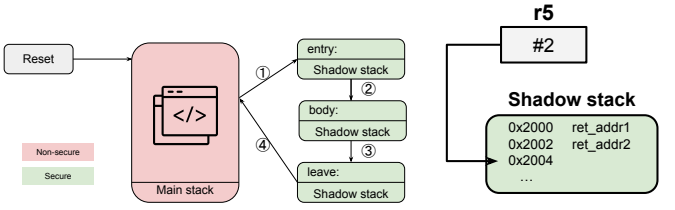
shadow stack. Because the function return address was pushed on the main stack (`r1` in MSP430) at the function prologue, this value is loaded to `r6` and `NS_EILID_check_ra()` is called to check before `ret` instruction.

Figures 5 and 6 show how the code is instrumented to ensure **P2**. In MSP430 architecture, when an interrupt is triggered, the current instruction address and the status register (`r2`) are pushed on the main stack. We consider these two values as the interrupt context. At the ISR entry point, the interrupt context is loaded to `r6` and `r7` as arguments, and in turn, `NS_EILID_store_rai()` is called. Before its return, the system loads the interrupt context to `r6` and `r7`, and validates the context in `NS_EILID_check_rai()`.

As illustrated in Figure 7, at the beginning of `main()` function (once the device boot sequence completes), a few instructions are introduced calling `NS_EILID_store_ind_func()`. The code stores all legitimate function addresses of the device software in a table. Figure 8 shows that the target address of an indirect function is checked via `NS_EILID_check_ind_func()`. It makes sure that the address is present in the table before each indirect function call.

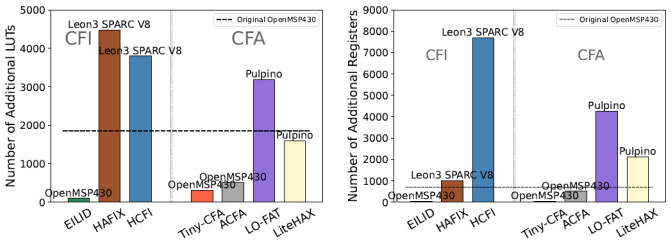
B. EILID$_{sw}$

EILID$_{sw}$ is composed of three sections: entry, body, and leave. The entry section is the only legal entry point that can switch to a secure state, thus minimizing the attack surface. The body section houses all `S_EILID` functions. Before transitioning to execute non-secure software, the exit section must be passed through. EILID$_{hw}$ (i.e., CASU hardware) ensures atomicity of all authorized functions running in the secure state by resetting the device if any violation occurs.



(a) EILID Software Flow (b) Shadow Stack Example

Fig. 9: EILID Software Implementation



(a) EILID LUT overhead (b) EILID Register overhead

Fig. 10: EILID Hardware Overhead Comparison

Figure 9(a) illustrates the software flow on EILID-enabled devices. After a reset, the device runs its normal software. Instrumented code jumps to EILID$_{sw}$, switching to the secure state – ①. Within the entry section, `r4` determines which `S_EILID` function is invoked. For example, if `r4`$==1$, it branches to `S_EILID_store_ra()` in the body section – ②. In the function, the return address is stored on the shadow stack. Finally, it branches to `S_EILID_exit()` in the exit section – ③, and resumes normal device software – ④.

Recall that `r5` is reserved for the shadow stack index. For example, if `r5`$==2$, as shown in Figure 9(b), the next return address is stored at $0x2000 + 2 * (2 - 1) = 0x2002$ (each memory size is 2 bytes in 16-bit architecture). When `S_EILID_store_ra()` is invoked, it stores the address at $0x2000 + 2*$`r5`$=0x2004$ and increments `r5` by 1. This obviates the need for memory access to the shadow stack to maintain its index, thus improving performance. When `S_EILID_check_ra()` is executed, the stored address on the shadow stack is popped off and `r5` is decremented by 1.

VI. Evaluation

EILID Hardware Overhead: Recall that EILID hardware overhead is entirely derived from CASU hardware. Figure 10 compares hardware costs to prior CFI techniques (HAFIX [37] and HCFI [38]) as well as CFA schemes (Tiny-CFA [23], ACFA [24], LO-FAT [25], and LiteHAX [33]). Since most of these techniques are implemented on different architectures and platforms, it is difficult to compare them directly. However, we believe that EILID hardware cost is minimal because: (i) Tiny-CFA and ACFA, implemented on the same platform (openMSP430), have substantially higher hardware overhead, and (ii) other techniques geared for 32-bit architectures have more computing resources, as shown in Figure 10. In absolute numbers, EILID augments Look-Up Tables (LUTs) by 99 (5.3%) and registers – by 34 (4.9%), over the openMSP430 baseline. Whereas, Tiny-CFA and ACFA increase: (1) LUTs by 302 (16.2%) and 501 (26.9%), respectively, and (2) registers by 44 (6.4%) and 946 (136.7%), respectively. Furthermore, LO-FAT and LiteHAX require 216KB and 158KB of RAM, respectively [61]. Such sizes far exceed the entire addressable memory (64KB) of a 16-bit CPU of MSP430. This supports our claim that low-end devices can not use prior techniques designed for higher-end platforms.

Software		Compile-time	Binary size	Running time
Light Sensor	Original	$321ms$	233 byte	$251\mu s$
	EILID	$419ms$	246 byte	$277\mu s$
	diff	$98ms$ **(30.53%)**	13 byte **(5.58%)**	$26\mu s$**(10.36%)**
Ultrasonic Ranger	Original	$334ms$	296 byte	$2,094\mu s$
	EILID	$423ms$	349 byte	$2,303\mu s$
	diff	$89ms$ **(26.65%)**	53 byte **(17.91%)**	$209\mu s$ **(9.98%)**
Fire Sensor	Original	$341ms$	465 byte	$4,105\mu s$
	EILID	$484ms$	565 byte	$4,648\mu s$
	diff	143ms **(41.94%)**	100 byte **(21.51%)**	$543\mu s$ **(13.23%)**
Syringe Pump	Original	$318ms$	274 byte	$2,151\mu s$
	EILID	$458ms$	308 byte	$2,265\mu s$
	diff	140ms **(44.03%)**	34 byte **(12.41%)**	$114\mu s$ **(5.30%)**
Temp Sensor	Original	$351ms$	305 byte	$1,257\mu s$
	EILID	$465ms$	325 byte	$1,327\mu s$
	diff	114ms **(32.48%)**	20 byte **(6.56%)**	$70\mu s$ **(5.57%)**
Charlie-plexing	Original	$360ms$	325 byte	$4,930\mu s$
	EILID	$455ms$	342 byte	$5,146\mu s$
	diff	95ms **(26.39%)**	17 byte **(5.23%)**	$216\mu s$ **(4.38%)**
Lcd Sensor	Original	$370ms$	604 byte	$4,877\mu s$
	EILID	$474ms$	642 byte	$5,005\mu s$
	diff	104ms **(38.11%)**	38 byte **(6.29%)**	$128\mu s$ **(2.62%)**
Average Overhead		**34.30%**	**10.78%**	**7.35%**

TABLE IV: EILID Software overhead

EILID **Software Overhead:** Note that there are no benchmarking tools available for low-end devices, i.e., 8-bit or 16-bit architectures. Thus, we instead used several publicly available practical applications to evaluate EILID software overhead: [`LightSensor`, `FireSensor`, `UltrasonicRanger`][1], `SyringePump`[2], and [`Temp Sensor`, `Charlieplexing`, `LcdSensor`][3]. Some of these applications, ported to run on openMSP430, have been used to motivate and evaluate prior CFA and CFI techniques [22]–[24], [34].

Table IV presents experimental results for these applications at compile- and run-time. Applications are built on an Ubuntu 20.04 LTS desktop with an Intel i5-11400 processor running at 2.6GHz with 16GB of RAM. Compile-time in each scenario is measured on average over 50 iterations. In each case, compile-time increases by $\leq$ 44.03%, which is reasonable considering that EILID requires three compile iterations (see Figure 2). The binary size grows by $\leq$ 21.51% across all applications.

Run-time overhead, measured using Vivado 2022.1 behavioral simulation running at 100MHz, ranges from 2.62% to 13.23%. Given infrequent use of indirect function calls in typical low-end device software, run-time overhead primarily arises from ensuring function return address (**P1**) and return-from-interrupt (**P2**) integrity. The instrumentation overhead per function call or interrupt is $\approx$ $25.2\mu s$. In more detail, storing control-flow values takes $\approx$ $11.8\mu s$, while checking these values takes $\approx$ $13.4\mu s$. The number of instructions introduced for storing and checking is 26 and 29, respectively. We believe these results are reasonable, considering the context switch overhead between normal software and EILID_{sw}.

VII. Discussion

Non-control-data Attacks: Non-control-data attacks [62] do not directly corrupt control-flow metadata. Instead, these attacks manipulate application data, involving environmental data (i.e., data of sensor/actuator), user inputs, and loop/conditional variables. For example, $\mathcal{A}$dv can corrupt the heap memory to modify a loop variable in a `if` statement, thus skipping an integrity check before running a new task. Since such actions do not alter control-flow metadata, they can circumvent various control-flow protections that only monitor that metadata. To ensure integrity of such data, all memory used for non-control data must be monitored and protected.

However, monitoring all non-control data incurs an excessive software overhead. Consider a simple loop, `for (i=0; i<100; i++)`. Since manipulation of `i` influences the execution sequence, it should be protected to maintain data-flow integrity. However, preserving the value of '`i`' would require storing it 100 times and verifying – 99 times, since the initial value of 0 is stored without checking. Also, tracing such data outside the current function scope requires additional processing, when it is passed as function arguments or return values. This results in a prohibitive memory overhead that low-end devices can not afford.

Indirect Jumps: Consistent with prior work [63]–[65] and as mentioned earlier, EILID does not consider indirect jumps. They are primarily used to implement `switch` statements, which are converted into jump-tables at compilation. These jump-tables are read-only and boundary-checked at run-time to ensure that they do not leave the function boundary. EILID_{inst} avoids generating jump instructions by compiling with flags, such as `-fno-jump-tables` and `-fno-switch-tables`. Also, a compile-time warning is raised if there are indirect jumps besides `switch` statements.

Recursion: EILID does not handle recursion because of its excessive memory overhead. Specifically, recursion consumes significant stack space and hence it is rarely used in embedded systems software. A function call in MSP430 requires at least 2 bytes for the return address, plus a few more bytes to store local variables. For example, on a device with only 2KB of RAM, a recursive function can be invoked at most 204 times with 2 integer variables (8 bytes) in recursion. This number is significant if other functions or global variables are present. Thus, it is generally unadvisable to use recursion on low-end devices; instead, it makes more sense to convert them into iterative statements using loops.

VIII. Conclusions

This paper constructed EILID, a prevention-based `RoT` architecture that assures software integrity and prevents run-time attacks on low-end devices. Its evaluation on openMSP430 shows that EILID achieves a low average run-time overhead of $\approx$ 7.5% across seven real-world applications.

Acknowledgment

We thank DATE 2025 reviewers for their constructive feedback. This work was supported in part by funding from NSF Award SATC-1956393, NSA Awards H98230-20-1-0345 and H98230-22-1-0308, as well as a DARPA subcontract from Peraton Labs.

[1] https://github.com/Seeed-Studio/LaunchPad_Kit
[2] https://github.com/manimino/OpenSyringePump
[3] https://github.com/ticepd/msp430-examples

REFERENCES

[1] I. De Oliveira Nunes *et al.*, "CASU: Compromise avoidance via secure update for low-end embedded systems," *ICCAD*, 2022.
[2] F. Duarte, "Number of IoT devices (2023-2030)," Feb 2023.
[3] J. Deogirikar *et al.*, "Security attacks in IoT: A survey," *I-SMAC*, 2017.
[4] M.Nawir *et al.*, "Internet of things (IoT): Taxonomy of security attacks," *ICED*, 2016.
[5] F. Meneghello *et al.*, "IoT: Internet of threats? A survey of practical security vulnerabilities in real IoT devices," *IEEE IoT-J*, 2019.
[6] E. Ronen and A. Shamir, "Extended functionality attacks on IoT devices: The case of smart lights," *EuroS&P*, 2016.
[7] A. K. Sikder *et al.*, "A survey on sensor-based threats and attacks to smart devices and applications," *IEEE Commun. Surv. Tutor.*, 2021.
[8] M. Grisafi *et al.*, "PISTIS: Trusted computing architecture for low-end embedded systems," *USENIX'Sec*, 2022.
[9] J. Noorman *et al.*, "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices," *TOPS*, 2017.
[10] I. De Oliveira Nunes *et al.*, "VRASED: A verified hardware/software co-design for remote attestation," *USENIX'Sec*, 2019.
[11] I. De Oliveira Nunes *et al.*, "On the TOCTOU problem in remote attestation," *CCS*, 2021.
[12] F. Arkannezhad *et al.*, "IDA: Hybrid Attestation with Support for Interrupts and TOCTOU," *NDSS*, 2024.
[13] I. De Oliveira Nunes *et al.*, "PURE: Using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems," *ICCAD*, 2019.
[14] N. Asokan *et al.*, "ASSURED: Architecture for secure software update of realistic embedded devices," *TCAD*, 2018.
[15] S. Surminski *et al.*, "Realswatt: Remote software-based attestation for embedded devices under realtime constraints," *CCS*, 2021.
[16] M. Huber *et al.*, "The Lazarus Effect: Healing compromised devices in the Internet of Small Things," *AsiaCCS*, 2020.
[17] E. Aliaj *et al.*, "GAROTA: generalized active root-of-trust architecture," *USENIX'Sec*, 2022.
[18] M. Xu *et al.*, "Dominance as a New Trusted Computing Primitive for the Internet of Things," *S&P*, 2019.
[19] X. Jin *et al.*, "Understanding IoT security from a market-scale perspective," 2022.
[20] H. Alasmary *et al.*, "Analyzing and detecting emerging Internet of Things malware: A graph-based approach," *IEEE IoT-J*, 2019.
[21] R. Vijayakanthan *et al.*, "SWMAT: Mel-frequency cepstral coefficients-based memory fingerprinting for IoT devices," *Computers & Security*, 2023.
[22] T. Abera *et al.*, "C-FLAT: control-flow attestation for embedded systems software," *CCS*, 2016.
[23] I. De Oliveira Nunes *et al.*, "Tiny-CFA: Minimalistic control-flow attestation using verified proofs of execution," *DATE*, 2021.
[24] A. Caulfield *et al.*, "ACFA: Secure runtime auditing & guaranteed device healing via active control flow attestation," *USENIX'Sec*, 2023.
[25] G. Dessouky *et al.*, "LO-FAT: Low-overhead control flow attestation in hardware," *DAC*, 2017.
[26] M. Ammar *et al.*, "On bridging the gap between control flow integrity and attestation schemes," *USENIX'Sec*, 2024.
[27] N. Yadav and V. Ganapathy, "Whole-program control-flow path attestation," *CCS*, 2023.
[28] Y. Zhang *et al.*, "Recfa: Resilient control-flow attestation," *ACSAC*, 2021.
[29] A. J. Neto and I. D. O. Nunes, "ISC-FLAT: On the conflict between control flow attestation and real-time operations," *RTAS*, 2023.
[30] S. Zeitouni *et al.*, "Atrium: Runtime attestation resilient under memory attacks," *ICCAD*, 2017.
[31] H. B. Debes *et al.*, "ZEKRA: Zero-knowledge control-flow attestation," *AsiaCCS*, 2023.
[32] J. Wang *et al.*, "ARI: Attestation of real-time mission execution integrity," *USENIX'Sec*, 2023.
[33] G. Dessouky *et al.*, "LiteHAX: lightweight hardware-assisted attestation of program execution," *ICCAD*, 2018.
[34] I. De Oliveira Nunes *et al.*, "DIALED: Data integrity attestation for low-end embedded devices," *DAC*, 2021.
[35] T. Abera *et al.*, "DIAT: Data integrity attestation for resilient collaboration of autonomous systems.," *NDSS*, 2019.
[36] B. Kuang *et al.*, "DO-RA: data-oriented runtime attestation for IoT devices," *Computers & Security*, 2020.
[37] L. Davi *et al.*, "HAFIX: Hardware-assisted flow integrity extension," *DAC*, 2015.
[38] N. Christoulakis *et al.*, "HCFI: Hardware-enforced control-flow integrity," *CODASPY*, 2016.
[39] A. De *et al.*, "FIXER: Flow integrity extensions for embedded risc-v," *DATE*, 2019.
[40] J. Zhou *et al.*, "Silhouette: Efficient protected shadow stacks for embedded systems," *USENIX'Sec*, 2020.
[41] T. Nyman *et al.*, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," *RAID*, 2017.
[42] D. Sullivan *et al.*, "Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity," *Proceedings of the 53rd Annual Design Automation Conference*, 2016.
[43] N. Almakhdhub *et al.*, "μRAI: Securing embedded systems with return address integrity," *NDSS*, 2020.
[44] S. Das *et al.*, "A fine-grained control flow integrity approach against runtime memory attacks for embedded systems," *VLSI Systems*, 2016.
[45] H. Liljestrand *et al.*, "PAC it up: Towards pointer integrity using ARM pointer authentication," *USENIX'Sec*, 2019.
[46] R. T. Gollapudi *et al.*, "Control flow and pointer integrity enforcement in a secure tagged architecture," *S&P*, 2023.
[47] S. Park *et al.*, "Bratter: An instruction set extension for forward control-flow integrity in RISC-V," *Sensors*, 2022.
[48] S. Yoo *et al.*, "In-Kernel Control-Flow integrity on commodity OSes using ARM pointer authentication," *USENIX'Sec*, 2022.
[49] T. Murray *et al.*, "seL4: from general purpose to a proof of information flow enforcement," *S&P*, 2013.
[50] R. De Clercq *et al.*, "SOFIA: Software and control flow integrity architecture," *Computers & Security*, 2017.
[51] R. J. Walls *et al.*, "Control-flow integrity for real-time embedded systems," *ECRTS*, 2019.
[52] A. Fu *et al.*, "FH-CFI: Fine-grained hardware-assisted control flow integrity for ARM-based IoT devices," *Computers & Security*, 2022.
[53] A. Peng *et al.*, "CEFI: Command execution flow integrity for embedded devices," *DIMVA*, 2023.
[54] Anonymous, "EILID source code." https://anonymous.4open.science/r/eilid, 2023.
[55] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," *CCS*, 2007.
[56] M. Bauer and other, "Typro: Forward CFI for C-style indirect function calls using type propagation," *ACSAC*, 2022.
[57] X. Ge *et al.*, "Griffin: Guarding control flows using intel processor trace," *ACM SIGPLAN Notices*, 2017.
[58] A. J. Mashtizadeh *et al.*, "CCFI: Cryptographically enforced control flow integrity," *CCS*, 2015.
[59] C. Tice *et al.*, "Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM," *USENIX'Sec*, 2014.
[60] O. Girard, "openMSP430." https://opencores.org/projects/openmsp430, 2009.
[61] I. De Oliveira Nunes *et al.*, "APEX: A verified architecture for proofs of execution on remote devices under full software compromise," *USENIX'Sec*, 2020.
[62] S. Chen *et al.*, "Non-control-data attacks are realistic threats.," *USENIX'Sec*, 2005.
[63] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," *CCS*, 2019.
[64] X. Ge *et al.*, "Fine-grained control-flow integrity for kernel software," *EuroS&P*, 2016.
[65] N. Amit *et al.*, "JumpSwitches: Restoring the performance of indirect branches in the era of spectre," *USENIX ATC*, 2019.