Tiny-CFA: Minimalistic Control-Flow Attestation Using Verified Proofs of Execution

Ivan De Oliveira Nunes
University of California, Irvine
ivanoliv@uci.edu

Sashidhar Jakkamsetti
University of California, Irvine
sjakkams@uci.edu

Gene Tsudik
University of California, Irvine
gene.tsudik@uci.edu

Abstract—The design of tiny trust anchors attracted much attention over the past decade, to secure low-end MCU-s that cannot afford more expensive security mechanisms. In particular, hardware/software (hybrid) co-designs offer low hardware cost, while retaining similar security guarantees as (more expensive) hardware-based techniques. Hybrid trust anchors support security services (such as remote attestation, proofs of software update/erasure/reset, and proofs of remote software execution) in resource-constrained MCU-s, e.g., MSP430 and AVR AtMega32. Despite these advances, detection of control-flow attacks in low-end MCU-s remains a challenge, since hardware requirements for the cheapest mitigations are often more expensive than the MCU-s themselves.

In this work, we tackle this challenge by designing Tiny-CFA – a Control-Flow Attestation (CFA) technique with a single hardware requirement – the ability to generate proofs of remote software execution (PoX). In turn, PoX can be implemented very efficiently and securely in low-end MCU-s. Consequently, our design achieves the lowest hardware overhead of any CFA technique, while relying on a formally verified PoX as its sole hardware requirement. With respect to runtime overhead, Tiny-CFA also achieves better performance than prior CFA techniques based on code instrumentation. We implement and evaluate Tiny-CFA, analyze its security, and demonstrate its practicality using real-world publicly available applications.

I. INTRODUCTION

With the growth of the Internet of Things (IoT) and popularity of Cyber-Physical Systems (CPS), embedded devices have become ubiquitous in modern society. Since they often perform safety-critical tasks and process security- and privacy-sensitive data, they become an attractive attack targets. In this context, Remote Attestation (RA) has been proposed as a means to secure the software state of embedded systems. RA is a challenge-response protocol (see Section II-B for details) whereby a trusted verifier (Vrf) obtains an authentic and timely report about the software state of an untrusted (and potentially infected) remote device, called prover (Prv). This report allows Vrf to learn whether Prv’s current state is trustworthy, i.e., whether it hosts benign software. RA has been implemented efficiently, even on low-end MCU-s [9], [15], [5] to detect malware presence in the form of modified executables. However, conventional (aka static) RA can only ensure integrity of binaries and not of their execution.

Runtime/data-oriented attacks [22] tamper with execution state on the program’s stack or heap to arbitrarily divert the program’s execution flow. Such attacks need not modify the executable itself, but only the order in which its instructions are executed. Thus, they are not detectable by conventional RA. In particular, RA cannot detect runtime software attacks that hijack the program’s control-flow. Control-flow attacks can be launched by a variety of means. For instance, in languages such as C, C++, and Assembly (which are widely used to program MCU-s), buffer overflows [4] can overwrite functions’ return addresses, hijacking the program’s control-flow and launching well-known Return-Oriented Programming (ROP) attacks [17]. These attacks are especially dangerous for low-end MCU-s that can not avail themselves of more sophisticated OS-based mitigations, e.g., canaries, Address Space Layout Randomization (ASLR), and Control-Flow Integrity (CFI) techniques, available in high-end platforms. We discuss a concrete example of such an attack in low-end MCU-s (and how it is detected by Tiny-CFA) in Section IV-A.

Control-Flow Attestation (CFA) [1], [8], [7], [24] augments conventional RA capability to enable detection of control-flow attacks. In a nutshell, CFA techniques provide ∀rf a report that allows it not only learn if the expected code is loaded on Prv, but also which particular instruction path was taken during each execution of this program. In other words, CFA provides ∀rf with an authentic and unforgeable report that allows ∀rf to learn if instructions of a given program were executed in a particular expected/legal order, or a set thereof. This is typically achieved by securely logging information associated with the destination of each control-flow altering instruction, e.g., jumps, branches, returns, during program execution.

CFA techniques have been implemented on medium- to high-end embedded devices (e.g., Raspberry Pi, and RISC-V based processors), by leveraging trusted hardware support, such as ARM TrustZone, hardware branch monitors, and hardware hash engines. However, for resource constrained MCU-s, these requirements are too costly, since their hardware overhead is often higher than that of the MCU’s core itself, in terms of size, energy and monetary cost. To bridge this gap, our work leverages a recently proposed primitive – Proofs of Execution – PoX [6] (see Section II-D for details) – along with automatic code instrumentation, to derive a new CFA technique. Since PoX can be implemented efficiently even on most resource-constrained MCU-s, our CFA technique has considerably lower hardware overhead than that of prior work.

Contribution: we design, implement, and evaluate Tiny-CFA – a CFA technique based on automated software instrumentation where the only hardware requirement is that already provided (at relatively low-cost) by PoX architectures. As a result, Tiny-CFA hardware cost is about 1 to 2 orders of magnitude lower than prior CFA techniques and it is suitable for the low-end and ultra-low-energy MCU-s, such as MSP430 and AVR AtMega32. Furthermore, because our Tiny-CFA implementation relies on a formally verified PoX architecture as the sole architectural component on Prv, it is also the first CFA technique to offer the high-level of assurance provided by a formally verified Trusted Computing Base (TCB).

II. BACKGROUND & RELATED WORK

A. The Scope of Low-End Devices

This paper focuses on tiny CPS/IoT sensors and actuators (or hybrids thereof) with low computing power. These are some of the smallest and weakest devices based on low-power single-core MCU-s with only a few KBytes of program and data memory (such as...
the aforementioned Atmel AVR ATmega and TI MSP430), with 8- and 16-bit CPUs, typically run at 1-16MHz clock frequencies, with ≈ 64 KBytes of addressable memory. SRAM is used as data memory normally ranging in size between 4 and 16KBytes, while the rest of address space is available for program memory. Such devices usually run software atop “bare metal”, execute instructions in place (physically from program memory), and have no memory management unit (MMU) to support virtual memory. Our implementation is based on MSP430. This choice is due to public availability of formally verified RA [5] and PoX [6] architectures implemented on OpenMSP430 [10], which our work relies upon. Nevertheless, our design rationale is applicable to other low-end MCU-s in the same class.

B. Remote Attestation (RA)

As mentioned earlier, RA allows a trusted verifier (Vrf) to detect unauthorized code modifications (e.g., malware infections) on an untrusted remote device, called a prover (Prv) by remotely measuring the latter’s software state. Per Figure 1, RA is typically realized as a challenge-response protocol:
1) Vrf sends an attestation request containing a challenge (Chal) to Prv. This request might also contain a token derived from a secret that allows Prv to authenticate Vrf.
2) Prv receives the attestation request and computes an authenticated integrity check over a pre-defined memory region (e.g., program memory) and Chal.
3) Prv returns the result to Vrf.
4) Vrf receives the result from Prv, and checks whether it corresponds to a valid memory state.

![Fig. 1. RA interaction](image)

The authenticated integrity check is usually realized as a Message Authentication Code (MAC) or a digital signature over Prv’s memory. However, these cryptographic primitives require Prv to have a unique secret key (K) either shared with Vrf (MAC-s), or for which Vrf knows the public key (signatures). This K must reside in secure storage, and not be accessible to any (potentially compromised) software running on Prv, except for trusted attestation code itself. Since most RA threat models assume a fully compromised software state on Prv, secure storage implies some level of hardware support.

RA architectures fall into three categories depending on the level of hardware support: software-based, hardware-based, and hybrid. Security of software-based attestation [13], [19], [20], [21] relies on strong assumptions about precise timing and constant communication delays, which are unrealistic in the IoT/CPS ecosystem. Hardware-based methods [16], [23], [14], [18] rely on dedicated hardware components, e.g., TPMs [23], Intel SGX [11], or ARM TrustZone [2]. However, the cost of such hardware is prohibitive for low-end MCU-s. Hybrid RA [9], [3], [5] aims to achieve security equivalent to hardware-based mechanisms, with lower hardware cost. It implements the authenticated integrity ensuring function in software, while relying on minimal hardware support to assure that this software implementation executes properly and securely.

![Fig. 2. Overhead comparison between CFA architectures and PoX (APEX)](image)

(a) Additional HW overhead (%) in Number of Look-Up Tables (b) Additional HW overhead (%) in Number of Registers

C. Control-Flow Attestation (CFA)

In addition to detection of code modification via RA, CFA detects runtime attacks that hijack the program’s control-flow. C-FLAT [1] is the earliest CFA architecture. It uses ARM TrustZone’s secure world [2] to implement CFA, by instrumenting the executable with context switches between TrustZone’s normal and secure worlds. At each instruction that alters the control-flow (e.g., jump, branch, return), execution is trapped into the secure world and the control-flow path taken is logged into protected memory. C-FLAT targets higher-end embedded devices (e.g., Raspberry Pi) and its dependence on TrustZone (plus, numerous context switches) makes it unsuitable for low-end MCU-s targeted in this work. (Section II-A describes the scope of low-end MCU-s that we consider).

To remove the TrustZone dependence, subsequent CFA techniques: LO-FAT [8] and LiteHAX [7], implement CFA using stand-alone hardware modules: a branch monitor and a hash engine. Atrium [24] enhances aforementioned CFA techniques, securing them against physical adversaries that intercept instructions as they are fetched to the CPU. Though less expensive than C-FLAT, such hardware components are still not viable for low-end MCU-s, since their cost (in terms of price, size, and energy consumption) is typically higher than the cost of a low-end MCU itself. This is evident from Figure 2, which compares hardware costs – in terms of Look-Up Tables (LUTs) and numbers of Registers – of aforementioned CFA techniques and the total hardware cost of the OpenMSP430’s core itself, represented by dashed lines.

D. Proofs of Execution (PoX)

PoX augments RA capability by proving to Vrf that: (1) the expected executable is stored in program memory, and (2) this code indeed executed, and any claimed outputs were produced by its timely and authentic execution.

The first PoX architecture targeting low-end MCU-s was recently proposed in APEX [6]. APEX implements a hardware module controlling the value of a 1-bit flag called EXEC, that cannot be written by any software. A value \( \text{EXEC} = 1 \) indicates to Vrf that attested code must have executed successfully, between the time when the challenge was received from Vrf (recall the RA protocol from Section II-B) and the time when the RA measurement occurs (via authenticated integrity ensuring function). Similarly, when it receives an attestation reply with \( \text{EXEC} = 0 \), Vrf can conclude that execution of said code did not occur, or that execution (or its output) was tampered with. In APEX, the RA measurement covers:
(i) the *EXEC* flag itself; (ii) the region where this execution’s output is saved (output region – *OR*); and (iii) the executable itself (stored in the executable region – *ER*). Thus, security of the underlying RA architecture guarantees that the contents of these memory regions cannot be forged/spoofed to something different from their values at time of the attestation computation. In turn, APEX considers a code to execute properly (and sets *EXEC* = 1) if and only if:

1) Execution is atomic (i.e., uninterrupted), from the executable’s first instruction (legal entry *ER*\textsubscript{min}), to its last instruction (legal exit *ER*\textsubscript{max});

2) Neither the executable (*ER*), nor its produced outputs *OR* are modified in between the execution and subsequent RA computation;

3) During execution, data-memory (including *OR*) cannot be modified, by means other than the executable in *ER* itself, e.g., no modifications by other software or Direct Memory Access controllers.

These conditions mean that *EXEC = 1* assures that memory contents (of *ER* and *OR*) are consistent between *ER*’s code execution and respective attestation, and that execution itself is untampered, e.g. via interruptions, or modification of intermediate results in data memory. *ER* and *OR* locations and sizes are configurable, allowing for PoX of arbitrary program and output sizes. APEX implementation is built atop the formally verified hybrid RA architecture VRASED [5], and APEX hardware module is itself formally verified to adhere to a set of formal logic specifications. These properties, along with VRASED verified guarantees, are proven sufficient to imply a security definition (stated using the cryptographic security game framework [12]) for unforgeable of proofs of execution. Due to space constraints, we do not overview APEX proofs and refer the interested reader to [6].

As discussed in [6], similar to Trusted Execution Environments (TEEs) targeting higher-end platforms (e.g., Intel SGX [11] and ARM TrustZone[2]), APEX assumes executable correctness, i.e., the user is responsible for programming *Prv* with bug-free and memory-safe code. Hence, by default, APEX does not provide any security against runtime (aka control-flow) attacks. In this work, we bridge this gap by introducing an automated code instrumentation technique that leverages APEX to implement CFA in low-end MCUs. In other words, we show that CFA on top of APEX (or more generally any PoX), without any additional hardware requirement, is both possible and affordable. As a clear advantage over prior techniques, our approach requires 5.4 times fewer additional LUTs and 50 times fewer additional registers than the second cheapest approach – LiteHAX; see comparison of APEX hardware overhead with other CFA techniques in Figure 2).

III. Tiny-CFA

Tiny-CFA couples a formally verified PoX with code instrumentation to obtain CFA. It uses APEX PoX that ties the executed code to its output, stored in a data-memory range of configurable size, called *OR*. The basic idea is to instrument the code to produce a log of the program control-flow path, and make it a part of output. The program instrumentation writes the destination address of each control-flow altering instruction into *OR*. We denote this control-flow log as CF-Log.

As shown in Figure 3, in Tiny-CFA, both regular program outputs and CF-Log are written to *OR*. Recall from Section II-D that *OR* size/location is configurable. Hence, *Vrf* can chose *OR* to be large enough to fit both the regular program output and its expected CF-Log. Note that, in any CFA scheme, *Vrf* must have *a priori* knowledge of the expected/benign control-flows and their sizes. Therefore, the appropriate *OR* size is trivially obtained by adding the regular output and CF-Log sizes. The regular program output is written to *OR* normally, bottom-to-top of *OR*, as in APEX. Whereas, Tiny-CFA instrumentation writes CF-Log to *OR* from top to bottom. This strategy is similar to how stack and heap are handled in RAM and it assures that the program output and CF-Log do not interfere or overlap with each other, as long as *OR* is appropriately sized.

We believe that this general idea is both intuitive and sensible; it guides Tiny-CFA’s design. However, ensuring that Tiny-CFA results in a secure CFA design is significantly more challenging. To see why, note that the executable to be attested, (i.e., security-critical code stored in *ER*) is itself subject to control-flow attacks. Thus, any values logged to CF-Log by the instrumented executable can, in principle, be modified as part of a control-flow attack. In other words, Tiny-CFA’s approach is only secure if CF-Log is an append-only log. Otherwise, upon completion of its nefarious tasks, a control-flow attack can overwrite CF-Log to reflect a benign or expected control-flow, erasing any trace of the compromised control-flow and thus fool *Vrf*. In higher-end CFA architectures (e.g., C-FLAT [1]), this property is obtained by logging the control-flow to dedicated secure memory, which is never accessible to untrusted/application code, e.g., C-FLAT uses TrustZone’s secure world. However, as discussed in Sections I and II, low-end MCU-s cannot afford such expensive security features. Below, we detail how Tiny-CFA can be made secure by relying exclusively on PoX and instrumentation, thus retaining its suitability for low-end MCU-s.

A. Design Rationale & Security

We now discuss Tiny-CFA design rationale and security properties (P1-P6) at high-level. Implementation details of an instance of Tiny-CFA on MSP430 are further specified in Section III-C. We postulate the properties that ensure that control-flow attacks are always detected under the following comprehensive adversarial model:

Adversarial Model – we assume that the adversary controls *Prv*’s entire software state, including code and data. *Adv* can modify any writable memory and read any memory that is not explicitly protected by hardware-enforced access control rules (e.g., APEX rules). Program memory modifications can be performed to change instructions, while data memory modifications may trigger control-flow attacks. Adversarial modifications are allowed before, during, or after the execution of the program whose control-flow is to be attested.

[P1]: Integrity of Code, Instrumentation and Output – Clearly, any instrumentation-based approach is only sound if unauthorized modifications to the instrumented code itself (e.g., to remove instrumentation) are detectable. Detection of modifications is offered by the underlying RA and PoX architectures (see Section II). In particular, these architectures guarantee that any unauthorized code modification is detected by *Vrf*. They also guarantee that modifications to attested
executable’s output (OR – which includes CF-Log) are only possible if done by the attested executable itself, during its execution.

**P2:** Secure logging of control-flow instructions – The first step in Tiny-CFA, is to instrument all control-flow altering instructions to log their destinations to CF-Log, in OR. CF-Log is implemented as a stack, from the highest value in OR (OR$_{\text{max}}$) growing downwards, as shown in Figure 3. The pointer to the top of this stack is stored in a dedicated register $\mathcal{R}$. Each control-flow instruction is then instrumented with additional instructions to push its destination address to this stack, i.e.: (i) write the destination of address to the memory location pointed to by $\mathcal{R}$; and (ii) decrement $\mathcal{R}$. At instrumentation time, the assembly code of the executable is inspected to assure that no other instructions utilize the MCU register $\mathcal{R}$. In all practical examples considered in this work, executables have at least one free register available. If no such register exists by default, the code can be recompiled to free up one register.

**P3:** Secure logging of conditional branches – Conditional branches determine control-flow at runtime, depending on a result of a conditional statement, e.g., a comparison or equality test. These instructions are used to implement loops and if-then-else statements used in high-level languages. Conditional branches are instrumented by pushing to CF-Log’s stack (using the same method as in P2) the possible destinations as well as the result of the conditional statement. This way, by inspecting CF-Log, $\mathcal{V}_{rf}$ can determine the exact path taken by the conditional branch.

**P4:** Write safety – Write operations are dangerous since they can be used during an attack to overwrite CF-Log, thus hiding the compromised control-flow from $\mathcal{V}_{rf}$. Direct writes (which modify constant addresses) are easy to deal with, because they can be statically inspected for safety at instrumentation time. In particular, the instrumenter can verify that no direct writes modify CF-Log reserved addresses in OR. Indirect writes modify memory addresses determined at runtime. Consequently, they require instrumentation to check their safety, also at runtime. After each indirect write, Tiny-CFA instrumentation introduces instructions to check whether the write destination is within CF-Log by checking if the write destination is within the range $[\mathcal{R}, \text{OR}_{\text{max}}]$ – the memory range currently in use to store CF-Log. Upon detection of an illegal write, execution is halted, implying an invalid control-flow.

**P5:** Wrap-around attack protection – Given the inability to modify CF-Log due to checks performed in previous steps, the last resort for a control-flow attack to go undetected is to keep executing control-flow instructions until $\mathcal{R}$ value overflows and wraps-around, coming back to its initial value $\mathcal{R} = \text{OR}_{\text{max}}$ and overwriting of CF-Log. To protect against such attacks, modifications to $\mathcal{R}$ have an additional check, ensuring that whenever $\mathcal{R}$ points to an instruction outside OR range, execution is halted.

**P6:** Initialization verification – Previous properties rely on the fact that $\mathcal{R}$ is initialized as $\mathcal{R} = \text{OR}_{\text{max}}$ at the start of execution, to assure that CF-Log is indeed stored in OR. However, performing this initialization inside the executable being attested allows for control-flow attacks that jump back to the $\mathcal{R}$ initialization code to reset $\mathcal{R}$ in the middle of the execution. Instead of initializing $\mathcal{R}$ inside the attested executable, Tiny-CFA instruments the executable to check if $\mathcal{R}$ has been previously properly initialized to $\mathcal{R} = \text{OR}_{\text{max}}$. In turn, the caller application becomes responsible for initializing $\mathcal{R}$ appropriately, making control-flow attacks that re-initialize $\mathcal{R}$ to reset CF-Log impossible, since they require jumping outside of the executable range – $\text{ER}$ – which is detected by APEX as a violation.

**Security Argument:** Let $\mathcal{P}$ denote a procedure/function/code-segment for which execution and control-flow need to be attested. Properties P2 & P3 assure that all changes to the control-flow of $\mathcal{P}$ are logged to CF-Log at runtime. Then, by inspecting an authentic (untampered) CF-Log, $\mathcal{V}_{rf}$ can determine the exact control-flow taken by that particular $\mathcal{P}$ execution. Meanwhile, properties P5 & P6 guarantee that CF-Log is stored inside OR, within $[\mathcal{R}, \text{OR}_{\text{max}}]$ range. Property P4 detects any illegal writes during execution that attempt to modify CF-Log, i.e., writes to $[\mathcal{R}, \text{OR}_{\text{max}}]$ range. Hence, for a given execution of $\mathcal{P}$, the combination of P4, P5 & P6 guarantees that each written value can never be overwritten or deleted from CF-Log. Finally, P1, inherited from the underlying PoX architecture, assures that neither $\mathcal{P}$ instructions (including instrumentation), nor its output (including CF-Log) can be modified by other means (e.g., other software on $\mathcal{P}$rv, interrupts, DMA) before, during, or after execution. Any such attempt is detectable by $\mathcal{V}_{rf}$, because it causes APEX to set EXEC = 0; recall the EXEC flag behavior described in Section II-D. Therefore, Tiny-CFA properties P1-P6 suffice to implement secure CFA, under the aforementioned adversarial model.

## B. Optimizations

In practice, CF-Log size determines the practicality of Tiny-CFA due to the resource-constrained nature of low-end MCU-s, especially, with respect to memory size. In fact, although secure, the approach described thus far tends to bloat rapidly for control-flow intensive code segments, e.g., loops with many iterations. In this section, we discuss two simple optimizations (O1 & O2) that significantly reduce CF-Log size without sacrificing overall security.

**O1**- Static Control-Flow Instructions – We observe that control-flow instructions with constant destination addresses (determined statically in the code) need not be logged to CF-Log, as their effect on the program control-flow can not change at runtime. This removes the need to log operations, such as usual function calls (with exception of callbacks), fixed-address $90 \cdot 0t \cdot 0s$, and similar.

**O2**- Loops – Loops are challenging to log efficiently due to their high number of control-flow operations. For instance, consider a delay function based on busy-wait, commonly used in MCU code. It essentially consists of a loop that increments a counter up to a certain constant. The higher the delay, the higher the number of iterations, implying the higher the number of control-flow instructions to be logged. In turn, even a simple loop, such as a 1-second delay, would require millions of iterations (assuming typical clock frequencies on the order of MHz) resulting in millions of symbols logged to CF-Log. To deal with such cases, we introduce an optimization that removes the requirement to store each control-flow instruction for loops for which number of iterations can be predicted statically, at instrumentation time.

Specifically, Tiny-CFA instrumenter inspects each conditional branch. For each loop branch instruction instruction $bi$, changing the control-flow to destination instruction $di$, the instrumenter inspects all instructions in the range $[bi, di]$. If no indirect control-flow instructions exist in this range, the number of iterations caused by such a loop can be determined exclusively by checking the branch condition and the variables involved in this condition. Therefore, instead of logging each branch at every iteration, Tiny-CFA simply logs the condition itself, only once. This allows $\mathcal{V}_{rf}$ to learn the exact control-flow generated by a loop (i.e., # iterations) without bloating CF-Log. In our 1-second delay example, instead of logging millions of symbols, the loop would log just a couple of bytes, corresponding to the loop exit condition (typically, a comparison to a constant, e.g.,
C. Implementing Tiny-CFA

We now describe how properties P1-P6 are securely implemented via automatic assembly instrumentation on the MSP430 MCU. Our instrumenter is implemented in Python with approximately 300 lines of code.

Figure 4 shows the instrumentation of indirect control-flow instructions: retention of the return address, which in MSP430 assembly must be loaded to register r1 before ret is called, to CF-Log. In our implementation R = r4. Hence, the content of r1 (destination address) is copied to the address pointed to by R in OR, as required by P2. To also enforce P5, upon writing to the address of R, and moving R to point to the next address, the comparison at line 3 checks if R is still inside OR, otherwise exiting the program, by jumping to an exit instruction at line 4.

Figure 5 depicts the instrumentation of indirect write instructions to enforce P4. Upon writing to a given memory location (address pointed to by r14, in this example), checks are performed to determine if this write operation did not modify CF-Log memory range: [R, ORmax]. If an illegal write occurs, program execution is halted (at line 5) and a control-flow attack attempt is detected.

Figure 6 shows the instrumentation, required by P6, at the beginning of the code segment. It ensures that R is properly initialized, otherwise halting execution at line 3.

Finally, Figure 7 depicts the instrumentation required by P3. It logs to CF-Log the results of conditional statements. Note that, after a conditional statement (e.g., at line 1) evaluation, the result is stored in the status register r2. Hence, the content of r2 is written to CF-Log (line 2), since it is sufficient to determine the destination of the conditional branch. The same check to enforce P5 in Figure 4, is also performed in this case, because information is being written to CF-Log. Since this check itself overwrites r2, the original value of r2 needs to be retrieved (at line 6) before the actual branch instruction at line 7.

Remark: Tiny-CFA can not be abused by control-flow attacks that jump in the middle of the instrumentation instructions. Such an illegal jump is logged to CF-Log and is thus detectable by Vrf. Since R never reverts (within a given execution), write checks (see Figure 5) make it impossible to delete any information logged to CF-Log, including jumps into the middle of instrumented code instructions.

i < 10^6). This optimization also applies to loops used in common memory/array manipulations, e.g., in memset and memcpy.

IV. CASE STUDY & EVALUATION

A. Case Study: Control-Flow Attacks in Low-End MCU-s

Control-flow attacks can be extremely harmful, especially, for low-end devices used for safety-critical tasks. To illustrate this point, we show an attack on a medical syringe pump application implemented on a low-end MCU. For clarity, we focus on a simplified version of the OpenSyringePump application. Later, in Section IV, we evaluate Tiny-CFA on three applications, including the original OpenSyringePump code, which is longer and more complex than the example used here. OpenSyringePump was also used to motivate and evaluate prior CFA approaches, e.g., C-FLAT.

Consider the C code segment in Figure 8. In this application, the MCU is connected through the general-purpose input/output (GPIO) port P3_OUT (used at lines 5 and 8) to an actuator, responsible for injecting a given dose of medicine, determined in software, according to commands received through the network, e.g., from a remote physician. The function injectMedicine injects the appropriate dosage according to the variable dose, by triggering actuation for an amount of time corresponding to the value stored in dose. To guarantee a safe dosage, the if statement (at line 4) assures that the maximum injected dosage is 9, thus preventing overdosing.

Dosage is determined according to a list of values, e.g., symptom severity measures received from a remote physician. The function parseCommands (line 11) is responsible for making a copy of the received values and processing them to determine appropriate dosage. However, this function can also be used to trigger a buffer overflow attack, leading to a malicious control-flow path. Specifically, because the size of copy_of_commands is static and equal to 5, an input array of larger size can cause other values in the program’s stack to be overwritten, beyond the area allocated for copy_of_commands, and including the memory location storing the return address of parseCommands. In particular, the return address is overwritten with the value recv_commands[5]. By setting the content of parseCommands[5] to the address of line 5 in Figure 8, such an attack causes the control-flow to jump directly to line 5 (when parseCommands returns), skipping the safety check at line 4, and potentially overdosing the patient.

The above attack example is detectable neither by static RA techniques nor by PoX techniques, since expected (unmodified) code still executes in its entirety, yet in an unexpected order. Tiny-CFA, however, detects such control-flow attacks, because the instrumenta-

1Available at: https://github.com/naroom/OpenSyringePump
tion of indirect control-flow instructions (e.g., return in Figure 4) commits the maliciously overwritten return address to CF-Log.

In Section IV we evaluate Tiny-CFA performance in 3 realistic safety-critical applications: (1) OpenSyringePump – the full implementation of our toy example in Figure 8; (2) FireSensor 2 – a fire detector based on temperature and humidity sensors; and (3) UltrasonicRanger 3 – a sensor used by parking assistants for obstacle proximity measurement.

![Fig. 8. Safety critical application exploitability by control-flow attacks.](image-url)

B. Experimental Results

Recall that, since Tiny-CFA requires no hardware support beyond that already provided by APEX [6], its hardware costs remain consistent with Figure 2. Therefore, this section focuses on other costs: code size increase, runtime overhead, and CF-Log size. As mentioned in Section IV-A, our evaluation instantiates Tiny-CFA on MSP430 with three real-world, publicly available, and safety-critical use cases: SyringePump, FireSensor, and UltrasonicRanger. Tables I and II present experimental results for these three applications in their unmodified forms and when instrumented by Tiny-CFA. In each case, the attested execution corresponds to one iteration of the application’s main loop (i.e., the application can report to \( V \) with the attestation response once per iteration), involving the respective program execution.

<table>
<thead>
<tr>
<th>Tiny-CFA application</th>
<th>Code Size</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyringePump</td>
<td>218 bytes</td>
<td>159644 cycles</td>
</tr>
<tr>
<td>FireSensor</td>
<td>434 bytes</td>
<td>20919 cycles</td>
</tr>
<tr>
<td>UltrasonicRanger</td>
<td>238 bytes</td>
<td>2799 cycles</td>
</tr>
</tbody>
</table>

**TABLE I**

**ORIGINAL APPLICATION COSTS**

<table>
<thead>
<tr>
<th>Tiny-CFA application</th>
<th>Code Size</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyringePump</td>
<td>416 bytes</td>
<td>162218 cycles</td>
</tr>
<tr>
<td>FireSensor</td>
<td>790 bytes</td>
<td>31818 cycles</td>
</tr>
<tr>
<td>UltrasonicRanger</td>
<td>442 bytes</td>
<td>3027 cycles</td>
</tr>
</tbody>
</table>

**TABLE II**

**INSTRUMENTED APPLICATION COSTS**

In all three cases, code size increases by \( \approx 80\% \), while CF-Log size ranges between 30 and 2k Bytes, and runtime overhead varies between \( \approx 2\% \) and \( \approx 50\% \). Tiny-CFA runtime overhead is \( \approx 76\% \), due to instrumentation of trampolines and context switches; see [1] for details. Meanwhile, in all considered applications, Tiny-CFA runtime overhead remains below \( \approx 50\% \). This is justified by: (1) simpler design that does not rely on trampoline hypercalls or context switches, and (2) optimization \( \mathcal{O}_2 \), which removes per-iteration instrumentation away from delay loops. Since delay loops are used frequently in sensing/actuation applications, this optimization comes in handy in most practical scenarios. However, we do not compare runtime overhead of Tiny-CFA with Lo-FAT and LimeHAX since these two techniques do not instrument code, instead detecting branches in hardware.

In summary, experimental results indicate that, in all sample applications, instrumented executables remain well within the capabilities of low-end MCU-s, thus supporting Tiny-CFA’s practicality.

V. CONCLUSIONS

We designed, implemented and evaluated Tiny-CFA: a low-cost CFA approach targeting low-end MCU-s. Tiny-CFA couples a formally verified PoX architecture with automated code instrumentation to yield an effective low-cost CFA. We argued security of Tiny-CFA and demonstrated, via a MSP430-based implementation, its ability to detect control-flow attacks.

Acknowledgments: We thank DATE’21 anonymous referees for their helpful comments. This research was supported by part from Army Research Office (ARO), under contract W911NF-16-1-0536 and Semiconductor Research Corporation (SRC), under contract 2019-TS-2907.

REFERENCES