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 mitigation techniques 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 ($\mathcal{R}A$) has been proposed as a means to secure the software state of embedded systems. $\mathcal{R}A$ is a challenge-response protocol (see Section II-B for details) whereby a trusted verifier ($\mathcal{V}rf$) obtains an authentic and timely report about the software state of an untrusted (and potentially infected) remote device, called prover ($\mathcal{P}rv$). This report allows $\mathcal{V}rf$ to learn whether $\mathcal{P}rv$'s current state is trustworthy, i.e., whether it hosts benign software. $\mathcal{R}A$ 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) $\mathcal{R}A$ 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 $\mathcal{R}A$. In particular, $\mathcal{R}A$ cannot detect runtime software attacks that hijack the program's controlflow. 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 controlflow 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 Vrf with 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 Vrf with an authentic and unforgeable report that allows Vrf 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 *C*FA 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 *C*FA 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 $\mathcal{P}rv$, it is also the first *C*FA 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 MCUs with only a few KBytes of program and data memory (such as the aforementioned Atmel AVR ATmega and TI MSP430), with 8and 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 $\mathcal{R}A$ [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, $\mathcal{R}A$ allows a trusted verifier ($\mathcal{V}rf$) to detect unauthorized code modifications (e.g., malware infections) on an untrusted remote device, called a prover ($\mathcal{P}rv$) by remotely measuring the latter's software state. Per Figure 1, $\mathcal{R}A$ 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)- *P*rv receives the attestation request and computes an *authenticated integrity check* over a pre-defined memory region (e.g., program memory) and *C*hal.

3)- $\mathcal{P}rv$ returns the result to $\mathcal{V}rf$.

4)- \mathcal{V} rf receives the result from $\mathcal{P}rv$, and checks whether it corresponds to a valid memory state.



Fig. 1. RA interaction

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

 $\mathcal{R}A$ 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. Hardwarebased 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 MCUs. Hybrid $\mathcal{R}A$ [9], [3], [5] aims to achieve security equivalent to hardware-based mechanisms, with low(er) 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.



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

Fig. 2. Overhead comparison between CFA architectures and PoX (APEX). Dashed lines represent the total hardware cost of MSP430 core itself. Hardware costs are as reported in the original papers [8], [7], [24], [6].

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 $\mathcal{R}A$ capability by proving to $\mathcal{V}rf$ 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 EXEC = 1 indicates to Vrfthat 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 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 $\mathcal{R}A$ 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_{min}), to its last instruction (legal exit ER_{max});

2)- Neither the executable (ER), nor its produced outputs OR are modified in between the execution and subsequent $\mathcal{R}A$ 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 code 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 $\mathcal{P}rv$ 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 MCU-s. 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



Fig. 3. OR region used to store regular program outputs and CF-Log.

appropriate OR size is trivially obtained by adding the regular output and CF-Log sizes. The regular program output is written to ORnormally, 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 is 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 controlflow, 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 $\mathcal{R}A$ and PoX architectures (see Section II). In particular, these architectures guarantee that any unauthorized code modification is detected by $\mathcal{V}rf$. 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_{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, Vrf 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 $[\mathcal{R}, OR_{max}]$ – 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 $\mathcal{R} = OR_{max}$ 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]: \mathcal{R} initialization verification – Previous properties rely on the fact that \mathcal{R} is initialized as $\mathcal{R} = OR_{max}$ 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 $\mathcal{R} = OR_{max}$. 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, sine they require jumping outside of the executable range – 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 P are logged to CF-Log at runtime. Then, by inspecting an authentic (untampered) CF-Log, Vrf 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 $[\mathcal{R}, OR_{max}]$ range. Property P4 detects any illegal writes during execution that attempt to modify CF-Log, i.e., writes to $[\mathcal{R}, OR_{max}]$ range. Hence, for a given execution of 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 Prv, interrupts, DMA) before, during, or after execution. Any such attempt is detectable by Vrf, 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 controlflow 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 go-to-s, 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 Vrf 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.,



(a) Original (b) Instrumented Fig. 5. Instrumentation example: indirect write instructions.

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

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: return in this particular example. It writes the return address, which in MSP430 assembly must be loaded to register r1 before ret is called, to CF-Log. In our implementation $\mathcal{R} = r4$. Hence, the content of r1 (destination address) is copied to the address pointed to by \mathcal{R} in OR, as required by **P2**. To also enforce **P5**, upon writing to the address of \mathcal{R} , and moving \mathcal{R} to point to the next address, the comparison at line 3 checks if \mathcal{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: $[\mathcal{R}, OR_{max}]$. 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 \mathcal{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 r^2 . Hence, the content of r^2 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 r2needs 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 retracts (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.







Fig. 7. Instrumentation example: conditional branches.

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 application¹. 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 P3OUT (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 of 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 $\mathcal{R}A$ 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) FireSensor² - a fire detector based on temperature and humidity sensors; and (3) UltrasonicRanger³ – a sensor used by parking assistants for obstacle proximity measurement.

1	<pre>int dose = 0;</pre>
2	
3	<pre>void injectMedicine(){</pre>
4	if (dose < 10){ //safety check preventing overdose
5	P30UT = 0X1;
6	<pre>delay(dose*time_per_dose_unit);</pre>
7	}
8	P30UT = 0×0;
9	}
10	
11	<pre>void parseCommands(int *recv_commands, int lenght){</pre>
12	<pre>int copy_of_commands[5];</pre>
13	<pre>memcpy(copy_of_commands, recv_commands, lenght);</pre>
14	<pre>dose = processCommands(copy_of_commands);</pre>
15	return;
16	}

Fig. 8. Safety critical application exploitable by control-flow attacks.

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: SvringePump, 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 Vrf with the attestation response once per iteration), involving the respective sensing and actuation tasks.

	SyringePump	FireSensor	UltrasonicRanger
Code Size	218 bytes	434 bytes	238 bytes
Runtime	159644 cycles	20919 cycles	2799 cycles
		TABLE I	

ORIGINAL APPLICATION COSTS

	SyringePump	FireSensor	UltrasonicRanger			
Code Size	416 bytes	790 bytes	442 bytes			
Runtime	162218 cycles	31818 cycles	3027 cycles			
CF-Log size	400 bytes	2068 bytes	30 bytes			

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 $\approx 50\%$. CF-Log size depends on the number of control-flow transfers occurring in the application. Programs performing simple tasks need smaller log size (< 1k bytes), while those with complex tasks would need larger log sizes.

Tiny-CFA exhibits lower runtime overhead than C-FLAT [1]. C-FLAT is only evaluated using the SyringePump example, and its reported 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

³Available at: https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/ Grove Modules/ultrasonic ranger

 $\approx 50\%$. This is justified by: (1) simpler design that does not rely on trampoline hypercalls or context switches, and (2) optimization O2, 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 LiteHAX 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 in part by funding from Army Research Office (ARO), under contract W911NF-16-1-0536 and Semiconductor Research Corporation (SRC), under contract 2019-TS-2907.

REFERENCES

- [1] T. Abera et al., "C-flat: Control-flow attestation for embedded systems software," in ACM CCS, 2016.
- Arm Ltd., "Arm TrustZone," 2018. [Online]. Available: https: //www.arm.com/products/security-on-arm/trustzone F. Brasser et al., "Tytan: Tiny trust anchor for tiny devices," in *DAC*.
- [3] ACM, 2015.
- C. Cowan *et al.*, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *IEEE DISCEX*. IEEE, 2000. [4]
- I. De Oliveira Nunes et al., "VRASED: A verified hardware/software [5] co-design for remote attestation," USENIX Security'19, 2019.
- [6] "APEX: A verified architecture for proofs of execution on remote devices under full software compromise," in USENIX Security, 2020. G. Dessouky et al., "Litehax: lightweight hardware-assisted attestation [7]
- of program execution," in 2018 IEEE/ACM ICCAD, 2018, pp. 1–8. "Lo-fat: Low-overhead control flow attestation in hardware," in [8]
- DAC. [6] —, Eberlat, Edweyerhead control now attestation in hadware, in *DAC*. ACM, 2017, p. 24.
 [9] K. Eldefrawy *et al.*, "SMART: Secure and minimal architecture for (establishing dynamic) root of trust," in *NDSS*. Internet Society, 2012.
 [10] O. Girard, "openMSP430," 2009.
 [11] Intel, "Intel Software Guard Extangions (Intel SGX)," [Online].
- Intel, "Intel Software Guard Extensions (Intel SGX)." [Online].
- [11] Available: https://software.intel.com/en-us/sgx
- [12] J. Katz and Y. Lindell, Introduction to modern cryptography. CRC ress, 2014.
- [13] R. Kennell et al., "Establishing the genuinity of remote computer systems," in USENIX Security, 2003.
 [14] X. Kovah et al., "New results for timing-based attestation," in IEEE S&P
- 12 2012
- [15] J. Noorman et al., "Sancus 2.0: A low-cost security architecture for iot devices," ACM TOPS, 2017.
- [16] J. Petroni et al., "Copilot A coprocessor-based kernel runtime integrity monitor," in USENIX Security, 2004.
 [17] R. Roemer et al., "Return-oriented programming: Systems, languages, and applications," ACM TISSEC, 2012.
- [18] D. Schellekens et al., "Remote attestation on legacy operating systems with trusted platform modules," *Science of Comp. Programming*, 2008.
 [19] A. Seshadri et al., "SWATT: Software-based attestation for embedded
- [20]
- devices," in *IEEE S&P '04*, 2004.
 —, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in *ACM SOSP*, 2005.
 —, "SAKE: Software attestation for key establishment in sensor
- [21] networks," in *DCOSS*, 2008. L. Szekeres *et al.*, "Sok: Eternal war in memory," in *2013 IEEE*
- [22] Symposium on Security and Privacy. IEEE, 2013, pp. 48–62. Trusted Computing Group., "Trusted platform module (tpm),"
- Trusted Computing Group., "Trust 2017. [Online]. Available: http://work-groups/trusted-platform-module/ [23] http://www.trustedcomputinggroup.org/
- [24] S. Zeitouni et al., "Atrium: Runtime attestation resilient under memory attacks," in IEEE ICCAD, 2017.

²Available at: https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/ Grove_Modules/temp_humi_sensor