C-FLAT: Control-Flow Attestation for Embedded Systems Software

Tigist Abera[1,*], N. Asokan[2], Lucas Davi[1], Jan-Erik Ekberg[3],
Thomas Nyman[2,3], Andrew Paverd[2], Ahmad-Reza Sadeghi[1], Gene Tsudik[4]

[1] Technische Universität Darmstadt, Germany
{tigist.abera,lucas.davi,ahmad.sadeghi}@cased.de

[2] Aalto University, Finland
{n.asokan,thomas.nyman,andrew.paverd}@aalto.fi

[3] Trustonic, Finland
{jan-erik.ekberg,thomas.nyman}@trustonic.com

[4] University of California, Irvine, USA
gts@ics.uci.edu

ABSTRACT

Remote attestation is a crucial security service particularly relevant to increasingly popular IoT (and other embedded) devices. It allows a trusted party (verifier) to learn the state of a remote, and potentially malware-infected, device (prover). Most existing approaches are static in nature and only check whether benign software is *initially* loaded on the prover. However, they are vulnerable to runtime attacks that hijack the application's control or data flow, e.g., via return-oriented programming or data-oriented exploits.

As a concrete step towards more comprehensive runtime remote attestation, we present the design and implementation of **C**ontrol-**FL**ow **AT**testation (C-FLAT) that enables remote attestation of an application's control-flow path, without requiring the source code. We describe a full prototype implementation of C-FLAT on Raspberry Pi using its ARM TrustZone hardware security extensions. We evaluate C-FLAT's performance using a real-world embedded (cyber-physical) application, and demonstrate its efficacy against control-flow hijacking attacks.

Keywords

remote attestation; control-flow attacks; embedded system security

1. INTRODUCTION

Embedded systems are becoming increasingly ubiquitous, permeating many aspects of daily life. They are deployed in automotive, medical, industrial, household and office settings, smart cities and factories, as well as in critical infrastructures. Unfortunately, this increased popularity also leads to numerous security issues [44]. Securing embedded devices is challenging, particularly because they are special-purpose and resource-constrained [23].

Remote attestation is a means of verifying integrity of software running on a remote device. Typically, it is realized as a challenge-

response protocol allowing a trusted verifier to obtain an authentic, and timely report about the software state of a (potentially untrusted and infected) remote device – a prover. Based on that report, a verifier checks whether prover's current state is trustworthy, i.e., whether only known and benign software is loaded on the prover.

The standard trust assumption needed for authenticity of the attestation report requires the existence of some trusted component – called a *trust anchor* – on the prover. Moreover, most attestation schemes assume malicious software (i.e., remote malware infestations) as their adversarial model. Prominent examples of trust anchors are secure components such as a Trusted Platform Module (TPM). Although available on many laptops and desktops, TPMs are too complex and expensive for deployment on low-end embedded devices. Ideally, trust anchors for small embedded devices should be light-weight, i.e., require minimal hardware features and assumptions, in light of recent proposals such as SMART [18], SANCUS [32], and Intel's TrustLite [24]. The next generation of ARM Microcontrollers (MCUs) [46] will feature TrustZone-M, a lightweight trust anchor.

Most current remote attestation approaches are *static* in nature. In such schemes, the prover's report is typically authenticated by means of a cryptographic signature or a MAC computed over the verifier's challenge and a *measurement* (typically, a hash) of the binary code to be attested. However, static attestation, though efficient, only ensures integrity of binaries and **not** of their execution. In particular, it does not capture software attacks that hijack the program's control flow [42]. These attacks tamper with the state information on the application's stack or heap to arbitrarily divert execution flow. State-of-the-art memory corruption attacks take advantage of code-reuse techniques, such as return-oriented programming, that dynamically generate malicious programs based on code snippets (gadgets) of benign code *without* injecting any malicious instructions [34]. As a result, the measurements (hashes) computed over the binaries remain unchanged and the attestation protocol succeeds, even though the prover is no longer trustworthy. These sophisticated exploitation techniques have been shown effective on many processor architectures, such as Intel x86 [39], SPARC [9], ARM [25], and Atmel AVR [19].

The problem arises because static attestation methods do not capture a program's runtime behavior (i.e., timely trustworthiness) of the underlying code. In particular, recent large-scale studies have shown [14, 11] that embedded software suffers from a variety of vulnerabilities, including memory errors (such as buffer overflows), that allow runtime exploits. To be truly effective, an attestation technique should report the prover's dynamic state (i.e., its current

---

executions details) to the verifier. As we elaborate in Section 9, there have been some proposals to enhance and extend static binary attestation [35, 20]. However, they either require involvement of an external trusted third party, or only attest higher level of policies at the Java bytecode layer by instrumenting the Java virtual machine. Hence, they do not capture control-flow related attacks at the binary level of embedded systems software.

Mitigation of runtime exploitation techniques has been a subject of intensive research. Prominent defenses against control-flow hijacking include: control-flow integrity (CFI) [1], fine-grained code randomization [13, 27], and code-pointer integrity (CPI) [26]. However, naïvely integrating these approaches into remote attestation protocols would provide limited state information to the verifier. In particular, these techniques only report whether a control-flow attack occurred, and provide no information about the actually executed control-flow path. Therefore, the verifier can not determine which (of the many possible) paths the prover executed. This limitation allows an attacker to undermine such defenses by means of so-called data-oriented exploits [12]. These attacks corrupt data variables to execute a valid, yet unauthorized, control-flow path. A prominent example of this attack is the corruption of an authentication variable, allowing the attacker to execute a privileged control-flow path. Recently, Hu et al. [21] demonstrated that such attacks allow Turing-complete malicious computation.

Goals and Contributions. This paper proposes **C**ontrol-**FL**ow **AT**testation (C-FLAT), a technique for precise attestation of the execution path of an application running on an embedded device. C-FLAT complements static attestation by measuring the program's execution path at binary level, capturing its runtime behavior. As a new approach, C-FLAT represents an important and substantial advance towards tackling the open problem of runtime attestation.

C-FLAT allows the prover to efficiently compute an aggregated authenticator of the program's control flow, i.e., of the exact sequence of executed instructions, including branches and function returns. This authenticator represents a fingerprint of the control-flow path. It allows the verifier to trace the exact execution path in order to determine whether application's control flow has been compromised. Combined with static attestation, C-FLAT can precisely attest embedded software execution so as to allow detection of runtime attacks.

In designing C-FLAT, we focus on embedded systems. As the initial proof-of-concept, we attest single-thread programs executed by small IoT MCUs, since C-FLAT is not meant for arbitrary complex applications. We discuss the path towards control-flow attestation of MCUs in Section 8.

The main contributions of this paper are:

- A novel and practical scheme for attestation of the application's execution path. In contrast to more traditional static attestation, C-FLAT captures application's runtime behavior (Section 4).
- Addressing several challenges unique to control-flow path attestation, such as handling loops and call-return matching (Section 4.2).
- A proof-of-concept implementation that features (i) a static analysis tool to determine valid control flows of an application, (ii) a static binary instrumentation tool to extend ARM binaries with C-FLAT functionality, and (iii) a background service implemented as a trusted application, using ARM TrustZone extensions, which monitors runtime control flow and generates the attestation response (Section 5).
- A systematic evaluation of C-FLAT in the context of *Open Syringe Pump*, a real embedded control application for a widely-used class of *cyber-physical* systems (Section 6). We also

demonstrate C-FLAT's resilience against various runtime attacks (Section 7).

We chose to initially instantiate C-FLAT using ARM TrustZone since ARM-based platforms (particularly Raspberry Pi) are widely available and popular for building embedded applications, e.g., the syringe pump described in Section 6.1. Although our implementation uses TrustZone-A hardware security extensions available in current general-purpose ARM processors, its more lightweight successor TrustZone-M will be available on ARMv8 MCUs [46] which are expected to come on the market soon. This will allow C-FLAT to be easily realizable on commercial MCUs (see Section 8).

Code Availability and Technical Report. To enable reproducibility of our results, and to encourage further research in this area, the source code for C-FLAT, our use-case cyber-physical programs, and the proof-of-concept exploits are available at https://goo.gl/pTiVdU. We also provide an extended version of this paper as a technical report [2].

2. PROBLEM SETTING

Runtime attacks exploit program vulnerabilities to cause malicious and unauthorized program actions. The most prominent example is a buffer overflow, allowing the attacker to corrupt memory cells adjacent to a buffer. The main target of these attacks is manipulation of control-flow information stored on the program's stack and heap.
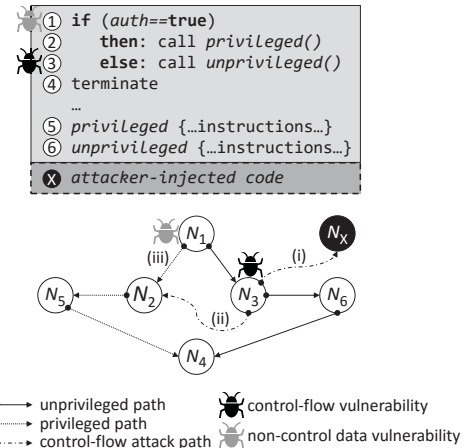

Figure 1: Abstract view of runtime attacks

Figure 1 shows a generic runtime attack example of a program that calls either a privileged or non-privileged subroutine based on the authentication variable *auth*.[1] Line numbers in the pseudo-code map to CFG nodes. The example program suffers from a control-flow vulnerability at node N_3 allowing the attacker to overwrite code pointers that store control-flow information. (Typically, these vulnerabilities allow the attacker to read from, and write to, the application's memory.) Upon encountering the corrupted code pointer, the program's control flow is deviated to either (i) previously injected code (node N_X) [3] or (ii) existing code (node N_2) such as system functions [41] or unintended code sequences [39]. The latter is commonly referred to as *code-reuse attack*, one type of which – called *return-oriented programming* – allows the attacker

---

[1]In general, any program can be abstracted through its corresponding control-flow graph (CFG), where nodes represent code blocks and edges control-flow transitions.

to generate arbitrary malicious program actions based on chaining short sequences of benign code. Since code-reuse attacks do not require injection of malicious code, they undermine the widely-deployed security model of data execution prevention (DEP) [31] which aims at preventing code injection attacks by either marking memory as writable or executable.

Code-reuse attacks have emerged as the state-of-the-art exploitation technique on various processor architectures. Both control-flow integrity (CFI) [1] and code-pointer integrity (CPI) [26] aim at mitigating these attacks. While CFI enforces the program always following a legitimate path, CPI ensures integrity of code pointers. However, these schemes do not cover so-called *non-control-data attacks* [12]. These attacks corrupt data variables which are used to drive the control flow of the program. In the example of Figure 1, node N_1 transitions the control flow to either N_2 or N_3, based on *auth*. Thus, if the program suffers from a non-control-data vulnerability, the attacker can change *auth* from false to true, so that execution continues in the privileged path although the user has not been authenticated to execute that path, i.e., attack path (iii) in Figure 2.

For the sake of completeness, attacks discussed thus far lead to unintended, yet valid, program flows. However, it is also possible to mount pure data-oriented programming (DOP) attacks which only corrupt memory load and store operations, without inducing any unintended program flows [21]. Consider a program that reads from a buffer via a data pointer and sends retrieved data over network. A pure DOP attack would only manipulate the data pointer, e.g., to reference a private key. Hence, the original program control flow would lead to leakage of the private key without incurring any suspicious control flows. Since we focus on control-flow attestation, such pure data-flow attacks are beyond the scope of this paper. As shown in Figure 1, we focus on control-flow related attacks launched either by manipulating control-flow information, or non-control-data, such as data variables, i.e., the attacks (i)-(iii).

3. SYSTEM MODEL

Figure 2 shows our system model: the verifier $\mathcal{V}er$ wants to attest runtime control flows of an application module on a remote embedded system – the prover $\mathcal{P}rv$. The application module is typically an entire program or a subset thereof, such as a specific function. Both $\mathcal{V}er$ and $\mathcal{P}rv$ are assumed to have access to the binaries of the underlying application.
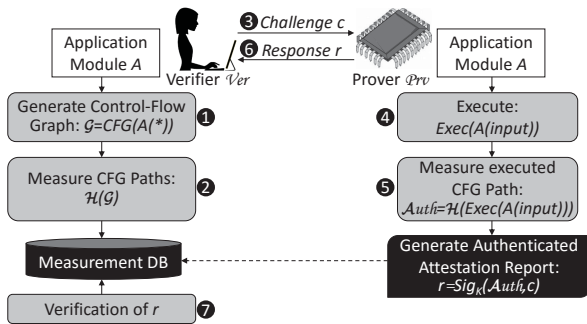
Figure 2: Overview of C-FLAT

3.1 Overview

C-FLAT requires $\mathcal{V}er$ to perform offline pre-processing: (1) generate the control-flow graph (CFG) of the application module via static analysis (❶), and (2) measure each possible control-flow path using a measurement function $\mathcal{H}$, and store the result in a measurement database (❷). This needs to be done only once per application module. Since $\mathcal{P}rv$ is assumed to be a low-end anemic embedded device with very limited memory, it cannot generate and store the entire CFG. However, this can be easily done by $\mathcal{V}er$ which has no such resource limitations.

NOTE: We realize that efficient computation of a generic program's CFG and exploration of all possible execution paths is an open problem. However, in this paper, we focus on static embedded application software which is typically much simpler than programs for general-purpose computers, e.g., the syringe pump program we analyze for our proof-of-concept consists of only 13k instructions.

In ❸, $\mathcal{V}er$ asks $\mathcal{P}rv$ to execute the application by transmitting a challenge that includes the name of the application module[2] and a nonce to ensure freshness. Next, $\mathcal{P}rv$ initiates execution of the application module (❹), while a dedicated and trusted Measurement Engine computes a cumulative authenticator $\mathcal{A}uth$ of the control-flow path (❺). At the end, $\mathcal{P}rv$ generates the *attestation report* $r = \text{Sig}_K(\mathcal{A}uth, c)$, computed as a digital signature over the challenge c and $\mathcal{A}uth$ using a key K known only to the Measurement Engine. Finally, $\mathcal{P}rv$ sends r to $\mathcal{V}er$ (❻) for validation (❼).

Since $\mathcal{P}rv$ attests the application code (with static attestation) and its control flow (with C-FLAT), $\mathcal{V}er$ can detect runtime attacks, as discussed in Section 2. Any deviation from the program's legitimate control flow results in an unrecognized measurement. Further, non-control data attacks output an unexpected, though valid, measurement allowing $\mathcal{V}er$ to detect attacks within the application's valid CFG. Static attestation (not shown in Figure 2), assures $\mathcal{V}er$ that $\mathcal{P}rv$ is running the intended application.

3.2 Requirements and Adversarial Model

As expected of an attestation scheme, we require that the attestation report (r) must be a *fresh* and *authentic* representation of the application's runtime control flows. The scheme itself must therefore be resilient against replay and masquerading attacks.

We make the following assumptions about the behavior of the adversary, denoted by $\mathcal{A}dv$.

$\mathcal{A}dv$ can introduce arbitrary malware into the prover. However, we rule out physical attacks, which is a standard assumption in all single-prover attestation protocols. We focus on attacks that hijack the execution path of the code to be attested (see Section 2). Such attacks are based on providing malicious inputs to $\mathcal{P}rv$'s public interfaces. Furthermore, we assume the following $\mathcal{P}rv$ features:

- Data execution prevention (DEP) to prevent an attacker from injecting and executing malicious code into running processes. All modern platforms – including embedded systems – provide hardware to support enforcement of this principle.
- A trust anchor that: (1) provides an isolated measurement engine, which cannot be disabled or modified by non-physical means (i.e., it allows measurements for both the static and control-flow path attestation), and (2) generates a fresh, authenticated attestation report. In Section 8, we discuss several concrete instantiations of this trust anchor.

These assumptions are in line with all previous work on remote attestation[3]. However, we consider a stronger $\mathcal{A}dv$, capable of control-flow hijacking attacks.

4. C-FLAT DESIGN

In order to perform control-flow attestation, $\mathcal{V}er$ asks for a measurement of $\mathcal{P}rv$'s execution path. This measurement should allow $\mathcal{V}er$ to efficiently determine and verify $\mathcal{P}rv$'s control-flow path.

---

[2]In practice, the verifier could transmit the actual application code.
[3]Assuming a trust anchor is typical for *remote* attestation protocols.

It is clearly infeasible to record and transmit every executed instruction, since that would: (1) result in a very long attestation response which $\mathcal{P}rv$ would have to store, and (2) require $\mathcal{V}er$ to walk through every single instruction. The same applies to another intuitive approach that would record and transmit source and target addresses of every executed branch, since such instructions frequently occur during program execution. To keep the attestation response short and allow fast verification, we propose a cumulative hash-based control-flow attestation scheme that builds a hash chain of executed control-flow transitions.

4.1 C-FLAT: High-Level Description

The main idea is to extend static (hash-based) attestation of binary files to dynamic (runtime) control-flow paths. Figure 3 illustrates this idea based on a simple control-flow graph (CFG) already shown in Section 2. Each CFG node contains a set of assembler instructions, and each edge represents a node transition by means
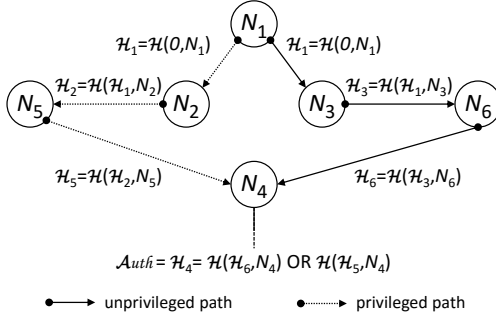

Figure 3: C-FLAT control-flow attestation

C-FLAT depends on every executed branch instruction. [...] ploys a measurement function $\mathcal{H}$ which takes as input: (1) [...] ID of the source node N_i, and (2) previous measurement: [...] $\mathcal{H}(\mathcal{H}_{prev}, N_i)$. At the beginning, when no previous measure[...] is available, we start with $\mathcal{H}_{prev} = 0$. As a result, there is a c[...] lative measurement for each possible program path, e.g., the [...] ileged path outputs $\mathcal{H}_5$, while the unprivileged path leads t[...] in Figure 3. Any unexpected measurement indicates to $\mathcal{V}e$[...] an illegal path has been executed. Furthermore, based on t[...] ported measurement, $\mathcal{V}er$ can easily determine whether the priv[...] leged path has been executed.

Due to its speed and simplicity, we chose the cryptographic hash function BLAKE-2[4] as $\mathcal{H}$ for cumulative hashing that yields $\mathcal{A}uth$. Hash-based measurements are already deployed in static attestation and allow mapping of an input to a unique[5] fixed-size result. To generate the final attestation report r, $\mathcal{P}rv$ can use a public key signature or a MAC over $\mathcal{V}er$'s challenge c and $\mathcal{A}uth$. (Without loss of generality, we use signatures in the rest of the paper.) In either case, the secret (private) key is assumed to be protected within the trust anchor.

Obviously, the number of valid measurements depends on the complexity and size of the application module, particularly, the

---

[4]https://blake2.net

[5]With overwhelming probability.

number of indirect and conditional branches. Indirect branches may target from 1 to n nodes, and conditional branches target 2 nodes in the CFG. Loops and recursive calls also lead to a high number of valid measurements, depending on the number of loop iterations, or recursive calls. In Section 4.2 we address this challenge using an approach for efficient loop and recursion handling, allowing us to limit the number of possible (legal) measurements.

4.2 Challenges

There are several challenges in instantiating C-FLAT. First, naïvely applying it to arbitrary code containing loops can lead to a combinatorial explosion of legal $\mathcal{A}uth$ values, since each execution involving a distinct number of loop iterations would yield a different $\mathcal{A}uth$ value. Furthermore, a static CFG does not capture call-return matching, e.g., a subroutine might return to various call locations. This would necessitate allowing too many possible $\mathcal{A}uth$ values that could be exploited by $\mathcal{A}dv$ [16].

Loops. Figure 4 depicts a CFG and its corresponding pseudo-code for a classic *while* loop, which contains an *if-else* statement. Note that conditional statements (lines 2 and 3) introduce nodes (N_2, N_3) with multiple outgoing edges. Hence, based on a condition check, they follow one of these edges. The main challenge is that the cumulative hash computed at N_2 is different at each loop iteration, since it subsumes the cumulative hash produced after the previous iteration. For an application that contains multiple (and even nested) loops, the number of legal $\mathcal{A}uth$ values grows exponentially, making control-flow attestation cumbersome.
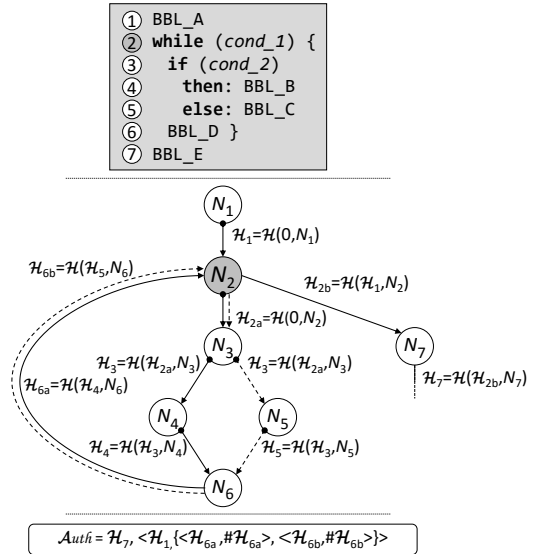

Figure 4: Loop Handling in C-FLAT

Our approach to tackling this problem is based on handling a loop as a sub-program. We measure each loop execution separately and merge its cumulative value with that of the previous execution, at loop exit. Consider the example in Figure 4: first, we identify the loop starting node by means of static analysis – N_2. Second, we initiate a new computation when the loop is entered, i.e., $\mathcal{H}_{2a} = \mathcal{H}(0, N_2)$. To avoid losing the previous value, we store $\mathcal{H}_1$.

Our example also has an *if-else* statement within the loop. It diverts control flow at N_3 to either N_4 or N_5, depending on *cond_2*. Consequently, each loop iteration can either output $\mathcal{H}_{6a}$ (solid line) or $\mathcal{H}_{6b}$ (dashed line). Upon loop exit, it is also desirable to attest the number of times a loop is executed. To do so, we track each

loop measurement separately, by storing the number of times a distinct measurement was encountered at N_2, and including this result $\#\mathcal{H}_{6a}, \#\mathcal{H}_{6b}$ in $\mathcal{A}uth$ where $\#\mathcal{H}_i$ reflects the number of loop iterations for each possible path. This ensures that every loop iteration is attested. However, the size of $\mathcal{A}uth$ is now expanded by the number of different paths inside the loop. In Section 6.2, we demonstrate some actual $\mathcal{A}uth$ sizes for several sample runs of the syringe pump application.

Upon loop exit, we simply take the recorded measurement of the execution right before the loop was entered ($\mathcal{H}_1$) for the computation of $\mathcal{H}_{2b}$. The measurement for $\mathcal{H}_{2b}$ does not capture whether the loop has executed. However, observed loop path measurements are reported in $\mathcal{A}uth$. We denote these with $< \mathcal{H}_i, \#\mathcal{H}_i >$ to report the loop path measurement and the number of times this measurement has been observed. Since a program might enter the loop at different execution stages, we need to separate loop counters for each stage. Otherwise, $\mathcal{V}er$ will not be able to determine the time when the loop has been entered. For this, we maintain a reference that indicates the program stage at which loop measurements were taken. This is achieved by simply taking the previous measurement right before the loop is entered, e.g., $\mathcal{H}_1$ in Figure 4.

This general approach also supports nested loops and recursive function calls, since each starting node initiates a separate measurement chain and the result at loop exit (or recursion return) is reported as separate measurements in $\mathcal{A}uth$. We also perform call-return matching on $\mathcal{P}rv$ in order to distinguish multiple instances of a particular loop from each other. This includes recursive calls to subroutines that contain loops. Such loops occur in the CFG only once but at runtime multiple loop instances might exist concurrently. In such cases, each distinct instance is represented separately in $\mathcal{A}uth$. For further details we refer the reader to the accompanying technical report [2].

Break Statements. Another related challenge arises when break statements occur inside loops or switch-case statements. Once encountered during program execution, they immediately terminate the innermost loop. Figure 5 shows a code example and its associated CFG for such a case. Compared to the previous loop example in Figure 4, node N_5 now executes a break instruction, where the path leading to the break is indicated by a dotted line.
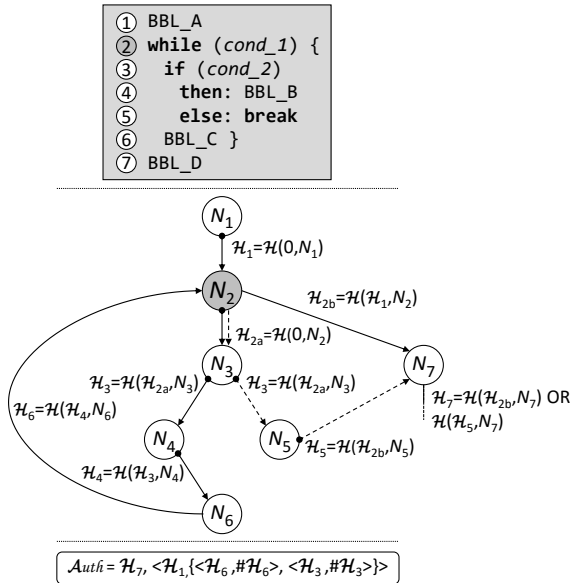
We need to treat break statements inside loops as special loop exit nodes with a slightly modified measurement function, for the following reason. Typically, loop exits only occur at the conditional check at node N_2. However, break in N_5 will not return back to the conditional statement, but immediately to N_7. Hence, we slightly extend the original measurement function to capture the fact that the measurement right before the loop was $\mathcal{H}_1$ and the loop has been terminated over the break statement. To do so, the measurement at N_5 indicates that the loop has been entered at N_2 with the previous hash value $\mathcal{H}_1$ ($\mathcal{H}_{2b}$) and terminated via N_5. The same principle is applied to goto statements that terminate a loop, e.g., a goto from a nested loop that also terminates an outer loop.

Call-Return Matching. Function calls and returns pose another challenge, especially, when a subroutine can be invoked from multiple calling locations. Figure 6 depicts a sample scenario where two function calls (N_2, N_3) call subroutine N_4. In the static CFG, the return instruction at N_4 can either target N_5 or N_6. However, if the previous flow is not considered, $\mathcal{A}dv$ can execute a malicious path, e.g., $N_2 \mapsto N_4 \mapsto N_6$. As recent work demonstrates, such malicious flows allow for a variety of control-flow attacks [16].

To cope with this issue, we index call and return edges during static analysis. In the CFG of Figure 6, the call from $N_2 \mapsto N_4$ is indexed with C_1 and the corresponding return edge $N_4 \mapsto N_5$ – with the same index marked as R_1. Hence, the only $\mathcal{A}uth$ values $\mathcal{V}er$ considers legal are $\mathcal{H}_{4a}$ and $\mathcal{H}_{4b}$.
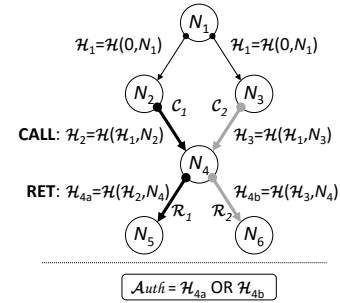

Figure 6: Call-return matching

5. IMPLEMENTATION

This section presents our prototype instantiation of C-FLAT and discusses its key implementation aspects.

5.1 Architecture of Proof-of-Concept

To demonstrate the feasibility of C-FLAT, we prototyped it on a popular embedded platform. As discussed in Section 3.2, remote attestation requires a trust anchor on $\mathcal{P}rv$. Because lightweight hardware trust anchors for small embedded devices, such as Intel's TrustLite or ARM's TrustZone-M, are not currently commercially available we opted for a more powerful ARM application processor that features TrustZone-A [4] security extensions for our prototype. Specifically, we chose Raspberry Pi 2 as the underlying embedded device platform. In our experimental setup, Raspberry Pi 2 is set up to execute a *bare metal*, monolithic, single-purpose program in the "*Normal World*" [4] of the ARM core, *without* a separate OS. In Section 8, we explain why the same approach is also applicable in systems where applications run on top of an embedded OS.

The prototype has two main components: i) a program analyzer to compute legal measurements of the target program, and ii) a *Runtime Tracer* and an isolated, trusted *Measurement Engine* to trace and measure the runtime control-flow path. The former can

Figure 5: Handling Break Statements in C-FLAT

be realized either as: (1) a static binary analyzer that generates the program's CFG by identifying basic blocks and their connection through branch instructions, or (2) a dynamic analyzer that produces valid measurements for a set of inputs by tracing execution of the target program. We opted for a dynamic analyzer. We also developed a binary rewriting tool for ARM binaries used to instru~~ment the target program in order to allow the Runtime Tracer to in~~
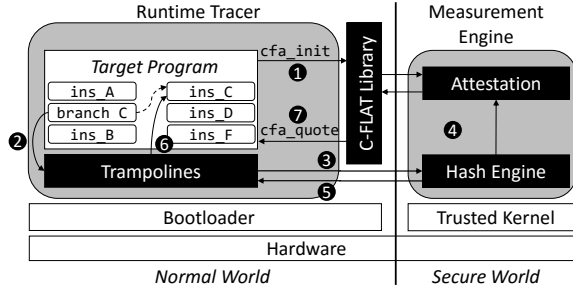

Figure 7: Proof-of-concept architecture

Instrumentation Phase. Initially the program binary is analyzed and instrumented using our instrumentation tool. During the analysis, information about direct branches and loops is collected from the binary, and stored in the *branch table* and *loop table* data structures included as part of the Runtime Tracer in the software image loaded onto Raspberry Pi 2. At runtime, these data structures are made available in read-only memory to the Runtime Tracer and Measurement Engine.

During instrumentation, each control-flow instruction in the target program is modified to pass control to *Trampolines* belonging to the Runtime Tracer. Further details about the instrumentation tool can be found in Section 5.2. We note that the analysis and transformations done to the target program could also be performed at program load time, either by a trusted bootloader, or a program loader in configurations where an embedded operating system is present on the device.

Runtime Operation. The C-FLAT Library (❶) serves as a mediator between the attested program on $\mathcal{P}rv$ and the Measurement Engine. It provides an API that allows software on $\mathcal{P}rv$ to initiate the attestation procedure, and obtain an attestation response from the Measurement Engine. Specifically, `cfa_init` initializes a new control-flow trace for attestation, and `cfa_quote` generates the attestation response from initialized trace. Once $\mathcal{P}rv$ receives the challenge c from $\mathcal{V}er$, it initiates a runtime trace by executing the `cfa_init` function (❷) and proceeds to execute the target program. The attestation procedure on $\mathcal{P}rv$ may be initialized *from within* the target program in cases where an attestation of only a particular operation is desired. The target program can, based on e.g., input from the verifier, decide *at runtime* which portions of program operation are to be attested.

As a result of the instrumentation of the target program binary, all control-flow instructions are intercepted by the Runtime Tracer Trampolines (❸). The Runtime Tracer determines the source and destination address of the observed branch, and the type of control-flow instruction that caused it (see Section 5.2). It then triggers the Measurement Engine (❹). The Measurement Engine performs the incremental measurement of the runtime control-flow path as described in Section 4, i.e., computes the cumulative $\mathcal{A}uth$ (using the BLAKE2 hash function). The Measurement Engine then trans-

fers control back to the Runtime Tracer, which in turn resumes the execution of the target program at the destination address of the observed branch (❺). Once the program procedure that is to be attested completes, $\mathcal{P}rv$ executes the `cfa_quote` function (❻) with the challenge c provided by $\mathcal{V}er$. This triggers the Measurement Engine to finalize $\mathcal{A}uth$ and generate the attestation response.

Trust Assumptions are as follows:

- Assumption 1: The Normal World software stack, including the attested target program and Runtime Tracer components are covered by the static attestation from $\mathcal{P}rv$ to $\mathcal{V}er$.
- Assumption 2: The Measurement Engine is trusted and cannot be disabled or modified.

Static attestation of the Normal World software stack ensures that $\mathcal{A}dv$ cannot modify this without being detected by $\mathcal{V}er$. This is important because it allows $\mathcal{V}er$ to detect if $\mathcal{A}dv$ tampers with Normal World software to manipulate the inputs to the Hash Engine computing cumulative hashes. Isolation of the Measurement Engine from the Normal World is important since, in the event of a compromised control flow (even without modifying Normal World software), $\mathcal{A}dv$ might attempt to influence hash computation prior to its delivery to $\mathcal{V}er$.

It is also possible that an adversary may attempt to mount a runtime attack against the Runtime Tracer and through this tamper the input to the Measurement Engine. However, the Measurement Engine can prior to relinquishing control back to the Trampoline that triggered the measurement validate that the actual destination of the recorded branch matches the input to the algorithm.

We implemented the Measurement Engine as a monolithic trusted kernel executing in the ARM TrustZone "Secure World" [4], thus isolating it from the untrusted target program, and the Runtime Tracer, to prevent $\mathcal{A}dv$ from tampering with measurements.

Node IDs. Recall that C-FLAT requires unique node IDs for CFG nodes. There are several options for doing this: **(1):** Identify entry and exit addresses of basic blocks. These addresses are unique and already present, without extra instrumentation. However, they change for each run of the program, if memory layout randomization (ASLR) [27] is used. **(2):** Using labels, instrument the program to embed a unique ID at the beginning of each basic block, similar to label-based CFI schemes [1]. **(3):** Maintain a dedicated table that maps memory addresses of basic blocks to node IDs.

We selected **(1)**, since many embedded systems do not yet support ASLR due to lack of an MMU. However, if ASLR is used, we can either report the base address in the attestation result, or adopt alternative approaches.

5.2 ARM Instrumentation

Another consideration for our implementation is the instrumentation of ARM binaries. 32-bit ARM processors feature 16 general-purpose registers. All these registers, including the program counter (`pc`) can be accessed directly. In addition to a 32-bit RISC instruction set, ARM processors also support a 16-bit *Thumb* instruction set. Thumb instructions act as compact shorthands for a subset of the 32-bit ARM instructions, allowing for improved code density. Modern ARM processors extend the Thumb instruction set with 32-bit instructions (distinct from those in the 32-bit ARM instruction set) that may be intermixed with 16-bit Thumb instruction. The resulting variable-length instruction set is referred to as *Thumb-2*.

Programs conforming to the ARM Architecture Procedure Call Standard (AAPCS) [5] perform subroutine calls either through a **B**ranch with **L**ink (`bl`) or **B**ranch with **L**ink and e**X**change (`blx`). These instructions load the address of the subroutine to the `pc` and the return address to the link register (`lr`). The *non-linking* variants, **B**ranch `b` and **B**ranch and e**X**change, are used to direct the

control flow within subroutines as they update the `pc`, but leave `lr` untouched. The ARM architecture provides no dedicated return instruction. Instead, any instruction that is able to write to the program counter may function as an effective return instruction.

In order to faithfully reproduce the control flow of the program being attested, our instrumentation approach needs to be able to intercept all instructions sequences that affect the control flow of the monitored program. On our evaluation platform programs execute solely in the 32-bit ARM state, allowing us to instrument the program binary such that all control-flow instructions are replaced by instructions that transfer control to a fixed piece of code belonging to the Runtime Tracer.

Binary rewriting. In the instrumentation phase, the original branch targets for direct branch instructions are collected from the target program binary to be stored at runtime in the read-only branch table data structure that resides in target program memory. Each target address is indexed in the branch table by the location of the original branch instruction. These locations, the locations of indirect branches, as well as other control-flow instructions are overwritten with *dispatch instructions* that redirect program control flow to a short piece of assembler code, the *trampoline*. We use the `bl` instruction as the dispatch instruction. The value of `lr` after the dispatch is used to identify the call site of the trampoline. Our tool utilizes the Capstone disassembly engine[6] to identify control-flow instructions for rewriting.

Trampolines. Each different type of control-flow instruction requires a separate trampoline, but unlike previous work [15], our approach does not leverage a separate trampoline for each instrumented branch. The trampoline manages the return address register as well as the register holding the destination address in indirect branches. If the original instruction was a non-linking branch, the trampoline must retain the original link register value from the prior `bl` originally present in the executable, and restore the program link register upon returning to program code.

In order to avoid overwriting values stored in `lr`, the target program must not use `lr` as a general purpose register. This can be enforced at compile time using appropriate compiler options.

During program execution, the trampoline will collect the return address from `lr` and determine the original branch target. For direct branches, the trampoline will look up the original destination address from the branch table. For indirect branches, the trampoline will consult the corresponding register holding the destination address, and for instructions utilized as function returns, the trampoline will either read the destination address from the stored `lr` value, or read the return address from the program stack. It will then invoke the Measurement Engine through a Secure World transition, passing as parameters the source and destination address. After trampoline execution resumes, control is transferred back to the target program, to the original branch destination.

Conditional branches. All conditional branch instructions in the ARM instruction set have a linking `bl` variant that updates `lr`. During instrumentation, we use the `bl` variant to redirect execution to the corresponding trampoline, but do not change the condition under which the branch is executed. Conditional branches that are not taken must at runtime be inferred from the branch information collected during the instrumentation phase, or by introspecting the program code. In C-FLAT, we make the branch table available to the Measurement Engine, and infer non-taken conditional branches based on the start of the previous basic block known to the Measurement Engine, and the source address of the branch that caused us to enter the Secure World.

6. EVALUATION

We evaluated our design and prototype implementation by applying C-FLAT to a real embedded application. This section describes our case study application, explains the attestation results, and discusses runtime performance. The practical attacks and mitigations we demonstrated using this case study are presented in Section 7.

6.1 Case Study of a Cyber-Physical System

A *syringe pump* is an electromechanical system designed to dispense (or withdraw) precise quantities of fluid. It is used in a variety of medical applications, especially those requiring small quantities of medication to be administered relatively frequently. Many research fields, including chemical and biomedical research, also use syringe pumps. Commercial syringe pumps are usually expensive, especially if they have been certified, but there has recently been significant interest in producing low-cost open-source alternatives [45].

Functionality. A syringe pump typically consists of a fluid-filled syringe, a controllable linear actuator (e.g., using a stepper motor), and a control system. The control system's task is relatively simple and would typically be implemented on an MCU – it translates the desired input value (e.g., millilitres of fluid) into the control output for the actuator (e.g., setting a digital IO line high for a calculated duration). However, given its intended usage, this system must provide a very high degree of assurance that it is operating correctly. In both commercial and open-source designs, the control system may be a weak point since it runs embedded software and accepts external input (potentially from remote sources). Undetectable runtime attacks affecting either the control flow or critical data variables could have serious consequences. This is therefore a prime use case for C-FLAT.

Open Syringe Pump. For this case study, we used *Open Syringe Pump*, an open-source, open-hardware syringe pump design.[7] It is controlled by an Arduino MCU running an embedded application written in Arduino Script. Since C-FLAT is not yet available for this type of MCU, we ported the application to a Raspberry Pi, which provides the required ARM TrustZone extensions. Other open-source syringe pump designs already use the Raspberry Pi as a controller (e.g., [45]), but run the control software as an application on top of a Linux OS. To retain the embedded nature of the controller, we chose to port Open Syringe Pump as a *bare-metal* implementation, which removes potential vulnerabilities introduced by an OS. Our port required only minimal changes to the source code (less than 25% of the overall application) in order to adapt it from Arduino Script to C. Neither the functionality nor the control-flow graph (CFG) of the application was changed. In terms of functionality, this application reads user input via the serial connection, and then performs one of the following types of actions:

- `set-quantity`: a numeric input changes a state variable representing the quantity of fluid to be used;
- `move-syringe`: a trigger input causes the syringe pump to dispense or withdraw the set quantity of fluid;

These types of actions naturally have different control flows, but even for the same action, the exact control flow depends on the state variable (i.e., the user-defined quantity of fluid). Overall, the compiled application has a total of 13,000 instructions, and its static CFG contains 332 unique edges, of which 20 are loops.

6.2 Attestation Results

We evaluated the functionality of our prototype implementation by instrumenting the syringe pump controller application and attesting

---

[6]http://www.capstone-engine.org/

[7]https://hackaday.io/project/1838-open-syringe-pump

the critical section of this application. Specifically, we attested the sequence of actions that take place immediately after an external input is provided. This ensures that any action which exercises the critical functionality (e.g., causes the syringe to move) is covered by the attestation, whilst eliminating irrelevant time periods when the state of the system does not change. We supplied a range of different inputs and recorded the resulting $\mathcal{A}uth$ values.

For the `set-quantity` path, the maximum length of $\mathcal{A}uth$ was for any input value was 1527 bytes. This path contains 16 loop invocations, and 18 unique loop paths. Similarly, for the `move-syringe` path, the maximum $\mathcal{A}uth$ length was 1179 bytes (12 loop invocations and 14 unique paths).

As expected, whenever the same control-flow path was taken (e.g., `set-quantity` or `move-syringe`), the same final hash value was obtained. These hash values therefore enable the verifier to determine the precise control-flow path that was actually executed. In this application, these hash values remained the same irrespective of the quantity of fluid dispensed/withdrawn by the syringe. The reason for this is that the state variable is only used to determine how many times a critical loop should be executed. The value of the state variable is therefore captured entirely in the additional loop information provided with the attestation. Although the purpose of this loop handling behavior is to overcome the exponential increase in complexity caused by loops, in this application (and others like it) this also greatly simplifies the attestation output.

6.3 Performance Impact

We measured the time taken to perform C-FLAT attestation in the syringe pump case study in order to determine (a) which factors affect the attestation overhead, and (b) whether this overhead is tolerable in real applications. All timing measurements were obtained using the hardware's built-in performance counters. It is important to emphasize that, in a real-world deployment, this type of attestation would not be used on every execution of the program. The verifier can choose when and how frequently to attest the prover.

Attestation overhead. We measured the time required to perform a complete attestation of the system, whilst varying the application's state variable through a representative range of values (i.e., changing the fluid quantity from 1 μL to 1000 μL). We use the term *control-flow event* to refer to any execution of a control-flow instruction (e.g., branch, return, etc.). In other words, a control-flow event is a traversal of an edge in the CFG. Since C-FLAT executes its algorithm on every control-flow event, the overall attestation time varies depending on the number of control-flow events, as shown in Figure 8.
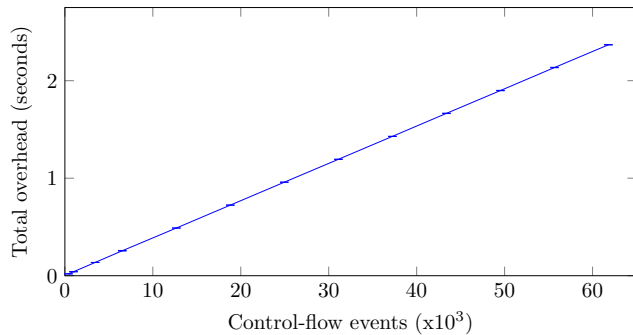

Figure 8: Attestation overhead (average over 20 repetitions)

The attestation overhead is linearly proportional to the number of control-flow events. This is as expected, since each control-flow event results in a trampoline invocation, a context switch, and an execution of the relevant Secure World algorithm. The type of event also impacts the overhead (e.g., loop handling is more costly), but this is not visible in Figure 8 due to the large number of events. The variance of these measurements is negligible, since there is no other software running on the platform.

The attestation overhead can be decomposed into three categories: i) trampolines, ii) context switches, and iii) the Secure World algorithm. For both the `set-quantity` and `move-syringe` paths, the relative percentage composition of the overhead remained roughly constant, irrespective of the state variable. For the `set-quantity` path, 8.3% of the overhead came from the trampolines, 11.9% from the context switches, and 79.8% from the algorithm itself. These ratios were similar for the `move-syringe` path at 10.2%, 17.5%, and 72.3% respectively. As expected, the majority of the overhead is due to the algorithm itself.

Performance impact. For the intended class of applications, this overhead would not necessarily have an impact on the real-world performance of the application. In cyber-physical systems, the main timing requirements usually arise from the system's physical processes and components (e.g., sensors and actuators), which operate on comparatively slow physical timescales. For example, in medical applications, typical flow rates for a syringe pump system could be $0.5\,ml/h$, $1.0\,ml/h$, and $2.0\,ml/h$.[8] In our case study, the attestation overhead is $1.2\,s$ for dispensing $0.5\,ml$, $2.4\,s$ for $1.0\,ml$, and $4.8\,s$ for $2.0\,ml$. Therefore, the real-world attestation overhead for these parameters ranges from 0.03% to 0.13%. Timescales in the order of minutes or hours are common in cyber-physical systems, for example, a typical embedded humidity sensor could measure and report a value once every 15 minutes[9], and an embedded thermostat would not switch on a compressor unit more than once every three minutes.[10]

Even if the attestation overhead begins to affect the system's timing requirements, various strategies can be used to ameliorate this situation. Firstly, we expect that the attestation would be run only occasionally, at a time determined by the verifier. When the application is not being attested, the overhead decreases significantly (e.g., by 72% to 80% for the syringe pump). This strategy could also be applied to other types of systems (e.g., attesting an embedded system in a car while not in use).

Secondly, the application could be adapted to accommodate this additional overhead. The above measurements were obtained by applying C-FLAT to an unmodified application, but if the source code is available, minor modifications could be made to the application to compensate for the attestation overhead (e.g., reducing the durations of application-defined delay loops).

Implementation considerations. An unavoidable side-effect of C-FLAT is that it may affect the way an application measures time. If the application relies on time-calibrated loops to delay for a specific duration, the calibration will change when the application is instrumented. Since the time spent in the Secure World varies depending on the type of control-flow operation, more complicated and/or experimental calibration would be required. However, this can be easily avoided by instead using well-defined timing capabilities, such as system timers.

6.4 Second Case Study

To further evaluate C-FLAT, we completed a second case study using a significantly larger application.

---

[8]http://www.ncbi.nlm.nih.gov/pubmed/11032282

[9]http://vanderleevineyard.com/1/category/vinduino/1.html

[10]https://github.com/bbustin/climaduino/tree/develop

We applied C-FLAT to a soldering iron temperature controller, which allows users to specify a desired temperature and maintains the soldering iron at that temperature using a proportional-integral-derivative (PID) controller.[11] This type of system could also be used to control temperature in other settings, such as a building or an industrial process, and thus is representative of a large class of cyber-physical systems. This application is significantly larger than the syringe pump as it consists of 70,000 instructions and has 1,020 static control-flow edges of which 40 are loops.

The security-critical functionality is the PID controller, which continuously reads the current temperature, compares it to the desired temperature, and calculates the PID output to adjust the temperature. We instrumented the application to attest the control flow of this critical section, and verified that the correct hash values were obtained. The total overhead added by C-FLAT to a single iteration of the PID controller was $237\,\mu s$ (average over 100 runs) with a standard deviation of $1\,\mu s$. As before, this overhead is significantly lower than the physical timescales on which these cyber-physical systems operate (e.g. attesting the controller once per second results in 0.03% overhead). For this attestation, the maximum length of $\mathcal{Auth}$ was 490 bytes.

7. SECURITY CONSIDERATIONS

In this section, we evaluate the security guarantees provided by C-FLAT based on practical embedded exploits and different runtime exploitation techniques. Note that our discussion refers to the runtime attack threat model outlined in Figure 1. The security of C-FLAT itself is described alongside the implementation in Section 5. As explained in that section, the integrity of the Runtime Tracer is ensured through static attestation, and the integrity of the measurement engine is protected by the trust anchor (e.g., the TrustZone Secure World). As usual, the freshness of the attestation is guaranteed by the challenge-response protocol between $\mathcal{Prv}$ and $\mathcal{Ver}$ (as described in Section 3).

Exploiting the syringe program. We constructed three exploits that affect the move-syringe function. Our first attack exploits a self-implanted memory error to trigger the movement of the syringe pump although no movement has been requested. As a consequence, liquid is dispensed at an unexpected time. We implemented this attack by reverse-engineering the syringe program binary and searching for return-oriented gadgets that allow us to load a function argument into ARM register r0 and transfer control to the move-syringe function. We were able to construct a suitable gadget chain to implement this attack. With C-FLAT enabled, we could detect this attack since the gadget chain leads to an unexpected hash measurement indicating to the verifier that a malicious program path has been executed.

Our second exploit is based on a non-control-data attack: it corrupts a local state variable that controls the amount of liquid to use. This variable is used to determine the number of iterations of the critical loop in order to dispense the correct amount of fluid. Hence, this attack does not violate the program's CFG. However, for the case of the syringe program, the loop iterations indicate to the verifier how much liquid has been dispensed. Again, we can detect this attack with C-FLAT since the precise loop iteration counts are reported in the attestation response.

Our third exploit, a non-control-data attack, corrupts the static key-map array used for processing input from the keypad. When a key is pressed, the keypad produces an analog value in a pre-defined range, which is read by the program. The program iterates

through a key-map array of known ranges, checking whether the analog input is the range for any recognized key. By changing the pre-defined ranges in the key-map array, our attack misleads the program into believing that a physical key has been pressed, thus causing the program to perform the relevant action (e.g. the right key triggers the move-syringe function, causing the syringe to dispense liquid). This attack can be detected by C-FLAT because the number of iterations of the key processing loop reveals which key appears to have been pressed. Assuming the verifier knows that the physical key has not been pressed, this can therefore be identified as unintended behavior.

Code Injection. Conventional control-flow hijacking attacks rely on the injection and execution of malicious code [3]. For this, the attacker needs to inject the malicious code into the data space of an application and corrupt control-flow information to redirect the execution to the injected code. Since malicious code resembles a new unknown node N_X (see Figure 1), C-FLAT can easily detect this attack. The reported hash value will include node N_X although it does not belong to the original control-flow graph (CFG) of the application. To undermine C-FLAT detection, an attacker may attempt to overwrite an existing node, e.g., replacing the benign Node N_2 with N_X. However, such an attack is not directly possible as we target platforms (see Section 3.2) that enforce data execution prevention (DEP). Hence, the memory area of node N_2 is not executable and writable at the same time. Furthermore, the so-called multi-stage exploits which deploy code-reuse attack techniques in the first place to mark the code region as writable and inject malicious code thereafter are not feasible under C-FLAT, because C-FLAT will report the execution of the corresponding memory modification system calls (hereafter denoted as *mem-calls*).

The only possibility to launch a code injection attack requires a program that benignly *mem-calls* to change memory permissions of code pages. For such cases, the attacker only corrupts input arguments to the *mem-calls* so that node N_2 is eventually overwritten with N_X. These attacks resemble pure data-oriented exploits since they do not require any code pointer corruption. As such, they cannot directly be detected by means of control-flow attestation. On the other hand, if the injected code performs unexpected calls to benign code, C-FLAT can still detect this pure data-oriented attack since an unknown hash value would be calculated.

Return-oriented programming. Code-reuse attacks that only maliciously combine benign code snippets from different parts of the program are more challenging to detect because they do not inject any new nodes. The most prominent instantiation is return-oriented programming [39]. It combines short code sequences, each ending in a return instruction, to generate a new malicious program. C-FLAT is able to detect return-oriented programming (including just-in-time return-oriented programming [40]) since unknown control-flow edges are taken upon execution of a return.

In general, these attacks exploit return instructions to transfer control to either (i) an arbitrary instruction in the middle of a CFG node, (ii) the beginning of an arbitrary CFG node to which no valid control-flow edge exists, e.g., the attack path from N_3 to N_2 in Figure 1, or (iii) another valid CFG node that is not the actual caller, i.e., originating node. Case (i) is detected by C-FLAT since no valid measurement can be generated when a node is not executed from its original beginning; even for nodes to which valid control-flow edges exist. For instance, if an attacker exploits the vulnerability in N_3 to target an instruction in the middle of the otherwise valid node N_6, the reported hash value will be different to the one expected. This is due to the fact that our measurement includes source and end address of each node visited.

For Case (ii), the attacker follows a control-flow path that is not

part of the CFG, e.g., node N_3 has no connection to node N_2. Due to the missing control-flow edge, the measurement will output an unknown hash value.

The last case is challenging to handle since the attacker targets a valid CFG node, i.e., there exists a valid control-flow edge but the current context indicates that the control-flow edge should not have been taken. We already discussed such scenarios in Section 4.2 for the call-return matching. C-FLAT indexes control-flow edges for function calls and returns to detect call-return mismatches. While this strategy works without special effort for direct function calls, indirect calls still pose an extra obstacle since static analysis may fail in resolving their target nodes. For those calls that static analysis cannot resolve, one can either leverage dynamic analysis or do an over-approximation by allowing each return instruction to target the instruction right after an unresolved indirect call. Note that this trade-off is not specific to our scheme, but a general problem in control-flow analysis. However, as we have shown in Section 6.1, embedded codes are typically tailored to a specific use-case allowing us to perform precise CFG analysis. In fact, basic dynamic analysis quickly allowed us to identify the valid paths in our syringe pump example.

Jump-oriented programming. Another variant of return-oriented programming are jump-oriented programming attacks [10, 7]. These attacks exploit indirect jump and call instructions at the end of each invoked code sequence. Similar to return-oriented programming, C-FLAT detects these attacks since they deviate the program's control flow to an unintended CFG node. As long as static or dynamic analysis identifies the set of target addresses, we are able to determine valid measurements for each indirect jump and call thereby allowing C-FLAT to detect malicious control-flow deviations incurred by jump-oriented programming.

Function-reuse attacks. Function-reuse attacks invoke a malicious chain of subroutines. Typically, these attacks are leveraged to undermine mitigation schemes that specifically aim at detecting return-oriented programming attacks. C-FLAT detects function-reuse attacks since it attests the program's entire control flow, i.e., it also captures the order in which functions are executing. For instance, the counterfeit object-oriented programming (COOP) [36] exploitation technique abuses a loop gadget to invoke a chain of C++ virtual methods. C-FLAT attests the execution within the loop, i.e., the number of times the loop has executed, and the order of virtual methods being called. Since the invoked chain does not resemble benign program execution, the resulting hash value will indicate to the verifier that a malicious path has been executed.

Non-control-data attacks. Non-control-data attacks manipulate variables that affect the program's control flow [12], e.g., to execute the privileged attack path (iii) in Figure 1. C-FLAT is able to detect these attacks as each individual control-flow path leads to a different measurement. Hence, the verifier $\mathcal{V}er$ can immediately recognize which path has been taken by the program at the prover's device $\mathcal{P}rv$. Another example is our attack reported in Section 7 which only modifies a data variable to manipulate the amount of liquid dispensed through the syringe pump. Since the amount of liquid dispensed is proportional to the number of times a loop has executed, and that number is reported in the attestation response, $\mathcal{V}er$ can detect the malicious modification by validating the loop counters.

8. DISCUSSION

Towards control-flow attestation of MCUs. The Raspberry Pi series of credit card-sized single-board computers have a become a popular platform among developers, researchers and hobbyists for building embedded applications. Raspberry Pi 2 represents the higher-end of the embedded device spectrum. The hardware trust anchor for the isolation of the C-FLAT Measurement Engine is TrustZone-A, an architectural feature in ARM application cores that has successfully been used as a hardware base for Trusted Execution Environments in mobile phones for the last decade, and has also made inroads in the top-end Internet of Things (IoT) processor families.

The Microcontroller (MCU) market for embedded computing in industry, automotive and home is rapidly moving towards 32-bit processing cores at the expense of small 8- and 16 bit controllers. For the latest version of its MCU core line (ARM-M), ARM introduced TrustZone-M [46] – hardware primitives for isolating security-critical functionality such as firmware upgrades or secure channel establishment from the rest of the controller software. Intel's TrustLite [24] provide very similar functionality. Both TrustZone-M and TrustLite can provide the necessary hardware fundament to implement C-FLAT on IoT MCUs.

Reducing context switch overhead. There are several optimization strategies depending on the properties of underlying secure hardware. On platforms equipped with a full-featured TEE, the context switch into the Secure World is expensive. In the case of TrustZone-A on an ARMv8 Cortex-A53 the context switch is approximately 3700 cycles or $5\mu s$ at 800MHz [33]. In the TrustZone-A architecture, the transition between the Secure and Non-Secure modes occurs via hardware exceptions mediated by a special exception handler referred to as the *Secure Monitor*. The transition can only be triggered from the Non-Secure World privileged mode. Under these circumstances, overall performance can be increased by caching control-flow information in the Non-Secure World before transferring it in bulk to Measurement Engine in the Secure World.

Hardware security architectures for highly resource constrained embedded devices, such as TrustLite or TrustZone-M, reduce the switching overhead associated with trusted application invocation to a few cycles. For example, TrustZone-M has no need for a dedicated Secure Monitor context. The transition between Secure and Non-Secure Worlds is handled by the processor circuitry independently of software through a mechanism based on protection regions with marked call gates enforced by the Memory Protection Unit.

Embedded operating systems. While the prover in our current prototype runs unassisted on bare-metal, many IoT devices feature embedded operating systems. The main purpose of an embedded OS (e.g., FreeRTOS[12], Zephyr[13], ChibiOS[14]) is to schedule tasks or functions, often based mainly on interrupt input. The OS may also provide memory management for the tasks and support debugging. A real-time OS adds pre-emptive scheduling to provide timing guarantees for individual tasks. However, hardware in controllers traditionally do not support many processing contexts or memory protection to achieve isolation between tasks or even between OS and application. From a security perspective, C-FLAT on such devices should therefore consider a single isolation domain even if an OS is present, and instrument the OS as a single bare-metal program.

Thumb-2 instrumentation. Modern ARM-M MCUs utilize the Thumb-2 instruction set because of its improved code density in order to maximize the usage off on-chip Flash memory. In ongoing work, we enhance our instrumentation approach to include support for the variable-length Thumb-2 instructions.

---

[12]http://www.freertos.org/

[13]https://www.zephyrproject.org/

[14]http://www.chibios.org

Data-flow attestation. As already mentioned in Section 2, we focus on control-flow related attacks. On the other hand, pure data-oriented attacks can lead to leakage of sensitive information, e.g., a data pointer that is processed in a network operation is altered to point to sensitive information. Hence, in our future work we will explore data-flow attestation, i.e., mechanisms that allow us to track data pointers and validate function arguments and return values.

Probe-based runtime attestation. In our ongoing work, we explore control-flow attestation for larger and very complex programs, i.e., programs with a complex control-flow graph that contains a large number of different program paths. These might lead to a combinatorial explosion of valid hash values thereby rendering validation cumbersome. To support such programs, we envision probe-based attestation that (i) generates and reports a cumulative hash for program segments, e.g., reporting an authenticator for each subroutine, and (ii) allows probing a program during its execution. The latter enables an interactive attestation protocol in which $\mathcal{V}er$ can repeatedly validate the evolution of the program execution at arbitrary time intervals.

9. RELATED WORK

Prior work on static attestation falls into three main categories: (1) TPM-based, (2) software-based, and (3) hybrid. Although TPMs and subsequent techniques (e.g., [30, 29]) are pervasive in PCs, they are not well-suited for low-end embedded devices due to their relatively high cost. Meanwhile, software-based methods (e.g., SWATT [38], Pioneer [37], and VIPER [28]) can be used – under some strict assumptions – for static attestation of certain legacy devices, such as peripherals. Between these two extremes, hybrid schemes such as SMART [18], TrustLite [24], and TyTAN [8] require minimal hardware trust anchors in order to provide strong *remote* attestation guarantees. C-FLAT is complementary to all these schemes in that it attests runtime behavior, which is orthogonal to static attestation.

Mitigation of runtime exploits is an ongoing research field with many solutions proposed over the last few years [42]. The main directions are control-flow integrity (CFI) [1] and fine-grained code randomization [13, 27]. The former ensures that a program only follows a valid path in the program's CFG. Many improvements have been made in the last years making CFI an efficient and effective mitigation technology [47, 43]. On the other hand, CFI does not cover non-control-data attacks which lead to execution of unauthorized but valid CFG paths. The latter randomizes the code layout by randomizing the order of memory pages, functions, basic blocks, or instructions [27]. However, the attacker can still exploit branch instructions to jump at her target address of choice. As such, code randomization cannot provide control-flow information which would allow $\mathcal{V}er$ to attest the program's execution. Another scheme is code-pointer integrity (CPI) [26] which aims at ensuring the integrity of code pointers. While it provides high assurance against control-flow hijacking attacks, it does not provide any information on the actual control-flow path taken. As such, similar to CFI, CPI does not cover non-control data attacks.

Property-based attestation explores behavioral characteristics beyond hash values of applications' binaries [35]. However, such schemes typically require a trusted third-party. Software-based attestation enables remote attestation for embedded devices without requiring a trust anchor [38]. The measurement is based on the calculation of a hash over the application's code memory, where the attestation is only successful if the prover responds in a certain time interval. However, this assumes no noise on the channel, and requires the hash function to be optimal. Semantic remote attestation enables attestation of program behavior by enforcing local policies

at the Java bytecode layer [20]. Unlike C-FLAT, neither property-based, semantic, nor software-based attestation cover control-flow attacks at the binary level.

There also exist several approaches to enable dynamic remote attestation: ReDAS [22] explores runtime properties such as the integrity of a function's base pointer. While checking against such properties enhances static remote attestation, it only captures specific application fields and pointers rather than the whole control-flow path. Trusted virtual containers [6] allow control-flow attestation but at a coarse-grained level. That is, they only attest the launch order of applications modules, but not the internal control flows of an application. Lastly, DynIMA [17] explores taint analysis to achieve runtime attestation. However, its implementation only validates invariants on return instructions such as the number of instructions executed between two consecutive returns.

10. CONCLUSION

Memory corruption attacks are prevalent on diverse computing platforms and can lead to significant damage on embedded systems that are increasingly deployed in safety-critical infrastructures. In particular, there is not yet a mechanism that allows precise verification of an application's control flow to detect such attacks. C-FLAT tackles this problem by means of control-flow attestation allowing a verifier to detect control-flow deviations launched via code injection, code-reuse, or non-control-data attacks. Our prototype implementation on an ARM-based Raspberry Pi demonstrates that C-FLAT can be leveraged to detect different runtime exploitation techniques launched against embedded software such as a syringe pump program, i.e., C-FLAT detects when an attacker manipulates the amount of liquid dispensed through the syringe. The source code of C-FLAT is available at https://goo.gl/pTiVdU.

11. ACKNOWLEDGMENTS

12. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM TISSEC*, 13(1), 2009.

[2] T. Abera, N. Asokan, L. Davi, J. Ekberg, T. Nyman, A. Paverd, A. Sadeghi, and G. Tsudik. C-FLAT: control-flow attestation for embedded systems software. *CoRR*, abs/1605.07763, 2016.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 2000.

[4] ARM Limited. *ARM Security Technology - Building a Secure System using TrustZone Technology*, 2009.

[5] ARM Ltd. Procedure call standard for the ARM architecture, 2009.

[6] K. A. Bailey and S. W. Smith. Trusted virtual containers on demand. In *ACM-CCS-STC*, 2010.

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *ACM ASIACCS*, 2011.

[8] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. TyTAN: Tiny Trust Anchor for Tiny Devices. In *ACM/IEEE Design Automation Conference*, 2015.

[9] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *ACM CCS*, 2008.

[10] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM CCS*, 2010.

[11] D. D. Chen, M. Egele, M. Woo, and D. Brumley. Towards automated dynamic analysis for Linux-based embedded firmware. In *ISOC NDSS*, 2016.

[12] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security*, 2005.

[13] F. B. Cohen. Operating system protection through program evolution. *Computer & Security*, 12(6), 1993.

[14] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large scale analysis of the security of embedded firmwares. In *USENIX Security*, 2014.

[15] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *ISOC NDSS*, 2012.

[16] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security*, 2014.

[17] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *ACM CCS-STC*, 2009.

[18] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: secure and minimal architecture for (establishing dynamic) root of trust. In *ISOC NDSS*, 2012.

[19] A. Francillon and C. Castelluccia. Code injection attacks on Harvard-architecture devices. In *ACM CCS*, 2008.

[20] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *Virtual Machine Research And Technology Symposium*, 2004.

[21] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the effectiveness of non-control data attacks. In *IEEE S&P*, 2016.

[22] C. Kil, E. Sezer, A. Azab, P. Ning, and X. Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *IEEE/IFIP DSN*, 2009.

[23] P. Kocher, R. Lee, G. McGraw, and A. Raghunathan. Security as a new dimension in embedded system design. In *ACM/IEEE Design Automation Conference*, 2004.

[24] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *ACM SIGOPS EuroSys*, 2014.

[25] T. Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-University Bochum, 2009.

[26] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX OSDI*, 2014.

[27] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *IEEE S&P*, 2014.

[28] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the Integrity of PERipherals' Firmware. In *ACM CCS*, 2011.

[29] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE S&P*, 2010.

[30] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *ACM SIGOPS EuroSys*, 2008.

[31] Microsoft. Data execution prevention (DEP), 2006.

[32] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security*, 2013.

[33] M. Paolino, A. Rigo, A. Spyridakis, , J. Fanguède, P. Lalov, and D. Raho. T-KVM: A trusted architecture for KVM ARM v7 and v8 virtual machines. In *IARIA Cloud Computing*, 2015.

[34] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM TISSEC*, 15(1):2:1–2:34, 2012.

[35] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *NSPW*, 2004.

[36] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE S&P*, 2015.

[37] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *ACM SIGOPS Operating Systems Review*, 2005.

[38] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE S&P*, 2004.

[39] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM CCS*, 2007.

[40] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE S&P*, 2013.

[41] Solar Designer. lpr LIBC RETURN exploit, 1997.

[42] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE S&P*, 2013.

[43] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security*, 2014.

[44] J. Viega and H. Thompson. The state of embedded-device security (spoiler alert: It's bad). *IEEE Security & Privacy*, 10(5):68–70, 2012.

[45] B. Wijnen, E. J. Hunt, G. C. Anzalone, and J. M. Pearce. Open-source syringe pump library. *PloS one*, 9(9):e107216, jan 2014.

[46] J. Yiu. ARMv8-M architecture technical overview. https://community.arm.com/docs/DOC-10896, 2015.

[47] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, 2013.