Computers, Materials & Continua
DOI:10.32604/cmc.2021.014523
Article

Tech Science Press

SAPEM: Secure Attestation of Program Execution and Program Memory for IoT Applications

Nafisa Ahmed[1], Manar Abu Talib[2,*] and Qassim Nasir[3]

[1]Research Institute of Science and Engineering, University of Sharjah, Sharjah, UAE
[2]Department of Computer Science, University of Sharjah, Sharjah, UAE
[3]Department of Electrical Engineering, University of Sharjah, Sharjah, UAE
[*]Corresponding Author: Manar Abu Talib. Email: mtalib@sharjah.ac.ae
Received: 26 September 2020; Accepted: 26 October 2020

Abstract: Security is one of the major challenges that devices connected to the Internet of Things (IoT) face today. Remote attestation is used to measure these devices' trustworthiness on the network by measuring the device platform's integrity. Several software-based attestation mechanisms have been proposed, but none of them can detect runtime attacks. Although some researchers have attempted to tackle these attacks, the proposed techniques require additional secured hardware parts to be integrated with the attested devices to achieve their aim. These solutions are expensive and not suitable in many cases. This paper proposes a dual attestation process, SAPEM, with two phases: static and dynamic. The static attestation phase examines the program memory of the attested device. The dynamic program flow attestation examines the execution correctness of the application code. It can detect code injection and runtime attacks that hijack the control-flow, including data attacks that affect the program control-flow. The main aim is to minimize attestation overhead while maintaining our ability to detect the specified attacks. We validated SAPEM by implementing it on Raspberry Pi using its TrustZone extension. We attested it against the specified attacks and compared its performance with the related work in the literature. The results show that SAPEM significantly minimizes performance overhead while reliably detecting runtime attacks at the binary level.

Keywords: IoT; remote attestation; runtime attacks; trust; TrustZone; security

1 Introduction

DEVICES linked to the Internet of Things are used in various contexts, including the military, healthcare, and industry. However, it is generally known that IoT devices are restricted in terms of resources, which combined with the connected devices' scale, leads to privacy and security challenges [1]. As the IoT develops, more attacks surface, and the vulnerability of these devices to attacks becomes one of the most crucial security challenges. It is, therefore, necessary to verify

the trustworthiness of IoT devices [2]. Remote attestation is a popular technique used to examine the devices' internal state remotely.

Remote Attestation: As shown in Fig. 1, the attestation process is a challenge-response protocol between a trusted attester known as the verifier (*Vrf*) and an untrusted attested device known as the prover (*Prv*) [3]. The *Vrf* begins the attestation process by generating a challenge (e.g., nonce) and sends it to the *Prv*. The latter performs the required function and sends the result as a response to the *Vrf*. Generally, there are two categories: (i) Static Attestation and (ii) Dynamic Attestation.

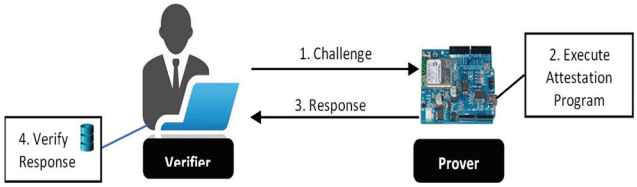

Figure 1: Remote attestation

Static attestation calculates a signature of the device's memory contents by traversing memory locations sequentially [4] or pseudo randomly [5]. In contrast, dynamic attestation calculates a program behavior signature to detect runtime attacks. Different parameters are used for dynamic signature calculations, including stack registers, a Program Counter (PC), and the executed program code parts [2].

Regardless of whether a static or a dynamic process is used, a pivotal point in attestation is the attestation technique's trustworthiness. To put it another way, the *Vrf* must be able to guarantee that the *Prv* did not tamper with the attestation program to produce a fake valid response. This concept is called evidence acquisition. There are three types of evidence acquisition: hardware attestation, software attestation, and hybrid attestation [3]. Hardware attestation involves particular secure hardware parts that must be integrated into the attested device. This type of attestation is enabled by the recent hardware technology and the arrival of the Trusted Platform Module (TPM) [6]. Although hardware attestation is exceptionally efficient, adding TPM to IoT devices is not always conceivable; furthermore, it is costly compared to software attestation.

Unlike hardware attestation, when software attestation is used, the *Prv* device does not have to acquire secured hardware parts. Instead, the *Prv's* physical constraints, such as the expected time for computations, are used to provide evidence acquisition [7]. As a result, the cost of using software attestation is significantly less than using hardware attestation, not to mention the fact that it is more convenient for embedded devices. However, existing software attestation techniques have been proven to be vulnerable to various attacks, and therefore cannot guarantee the security of the attested devices. The third type, hybrid attestation, lies between software and hardware attestation. It is a software attestation that utilizes the existing trust anchors in the *Prv's* device. A famous example of these trust anchors is the TrustZone provided by ARM microcontrollers.

Attacks: Researchers have proven that existing software attestation techniques are vulnerable to various attacks [3,7–9]. For instance, an attacker may predict the challenge and use it to compute the correct response in advance or make changes to the attestation program to fake the response [9]. Furthermore, an attacker may keep a copy of the original memory contents (i.e., either the full program or parts of it) and then redirect the memory addresses generated by the

attestation program to this copy [7]. When there is no available space for malicious code, an attacker may compress memory contents [8] or co-operate with other hijacked devices to divide the original memory contents among them [3].

Another category of attacks is runtime attacks. In the past, attackers used to exploit bugs to inject and execute malicious code in the program stack [10]. With the development of Data Execution Preventions (DEP), it became harder to execute the code injected in the data section. This development has led to the evolution of code reuse attacks, which are currently the most challenging since they don not modify the original code [8,11]. Difficult to detect with static attestation techniques, code reuse attacks are detectable only by dynamic attestation as the latter attests program behavior. The most common code reuse attack is Return Oriented Programming (ROP) [8]. In an ROP attack, the attacker uses of buffer overflow vulnerability, searching for small gadgets in the program ending in return statement and using them to serve the attacker's purposes [11]. Likewise, Jump-Oriented-Programming (JOP) attacks use gadgets ending in indirect branches [12].

Data attacks are another widely used example of runtime attacks [13,14]. Although these attacks only corrupt the data, they are used to execute unauthorized (but valid) paths in the program. Pure data attacks are used to corrupt data without affecting the program execution path.

Motivation and Contribution: The existing work of embedded device attestation in the literature has addressed several challenges:

1) Focus on static attestation, which cannot detect the runtime attacks.
2) The work in literature is either attesting memory integrity or program execution but not both, although they complement each other, especially in the case of tracking control-flow. As a result, the execution of injected/replaced code that does not affect the program flow will not be detected.
3) Dynamic approaches cannot detect the execution of unauthorized but valid path execution.
4) High overhead due to the use of complex operations.
5) Require costly secure hardware modules (TPM) for a secured attestation.

To the best of our knowledge, the literature didn't solve all above challenges in one design. We propose a novel approach, SAPEM, that achieves the following:

1) Dual attestation (static and dynamic): Static attestation checks memory integration and detects code injection attacks. Dynamic attestation checks program execution at the binary level and detects runtime attacks. The dual attestation addresses the above challenges 1 & 2.
2) All of the authorized program paths are represented by hash values known by the verifier. Therefore, SAPEM can detect even those data attacks that lead to the execution of unauthorized paths, although they are valid.
3) Minimized the attestation overhead by introducing an optimized approach for the hash computation of loops and conditional branch management.
4) Minimal security hardware requirement by utilizing the TrusTzone in ARM architecture to ensure the attestation program's secure execution.

The next section (Section 2) provides information about the problem. The threat model is illustrated in Section 3. Section 4 explains the design of the proposed attestation technique. Details about implementation are provided in Section 5. The following section (Section 6) discusses the

results, and Section 7 provides a review of different remote attestation techniques existing in the literature. Finally, a conclusion and pathways for future research work are presented in Section 8.

2 Problem Statement

We assume a setting in which there is an IoT device (*Prv*) that is resource-constrained. The IoT device is exposed to public networks. It interacts directly with the world to collect data or control physical components. These features make it attractive targets for runtime attacks [15]. Therefore, the IoT application is vulnerable to different kinds of remote software attacks. Like any IoT application, the *Prv* is collecting data and reacting to it. Hence, execution time is a crucial aspect of *Prv*.

On the other side, a trusted party (*Vrf*) aims to perform a remote check on the IoT application's internal state. The *Vrf* already knows the valid states of that IoT application. Many static software-based attestation techniques exist, but they can only detect attacks that change the application code on load time [3]. Recent research has attempted to detect these attacks using techniques that depend on extending hardware architecture. Others do not require specific hardware extensions, but they slow down the performance of the IoT application. Furthermore, most of them cannot detect control-flow hijacks caused by pure data corruption.

In this paper, we try to detect code injection attacks and runtime attacks (see Adversary below in this section) that hijack control-flow while minimizing the attestation overhead using software-based attestation in conjunction with a minimal trusted anchor. The main challenge is to minimize attestation overhead while preserving the ability to detect the specified attacks.

3 Threat Model

Device Architecture and Scheme: The *Prv* device considered in SAPEM has limited resources and is equipped with a trusted anchor (i.e., ARM TrustZone), considered secure. The secure world is isolated from the rest of the system, and it measures the *SS* and *PFS* and provides secure storage for all the critical variables needed by SAPEM. It is protected from an attacker by design. The normal world runs different applications and can be compromised.

The *Vrf* device considered in SAPEM is trusted and has powerful resources (e.g., Server).

The attestation scheme considered here is a one-to-one remote attestation where there are only one verifier and one prover involved in the attestation process. They communicate through SSL to allow a secure connection. The remote attestation is initialized at random periods by the *Vrf*, and the *Prv* will be listening for any incoming requests from the *Vrf*.

Adversary: We assume a remote adversary who knows memory corruption vulnerabilities, given that the IoT application is buggy but not malicious. The attacker may have identified the security flaws by analyzing the application source code or reverse-engineering the application binary. The attacker aims to execute a piece of code, corrupt specific data, or tamper with critical outputs of the IoT application. To achieve his goal, the attacker might inject malicious code when loading the application or reuse existing code by performing Return Oriented Programming (ROP), Jump Oriented Programming (JOP), or function reuse attacks. However, we assume that the attacker cannot tamper with the IoT hardware or with the firmware running on the target. Besides, pure data attacks (i.e., does not change the program execution path) cannot be detected using SAPEM.

4 Design of Dual Attestation

Fig. 2 illustrates the general steps of SAPEM. It includes two main phases: static and dynamic. The *Vrf* initiates both phases. The latter sends a nonce to the *Prv* and asks it to run the Static Attestation Routine (SAR). In the static phase, on the *Prv* side, the SAR computes a hash of the attested application's memory contents, called a Static Signature (*SS*). The *Prv* then sends the current *SS* to the *Vrf*, which we assume to have previous knowledge of valid *SSs* forms. Execution of the next phase is based on the received *SS*. An invalid *SS* received by the *Vrf* means that the IoT application has been hijacked or corrupted, and the attestation is immediately terminated. Otherwise, the *Vrf* sends another nonce to the *Prv* and asks it to run the Program Flow Attestation routine (*PFA*). The *PFA* routine computes a hash (*PFS*) of the application's executed path. Finally, the *Prv* sends the *PFS* to the *Vrf*, which compares it to the valid PFSs stored in its database. Further details about each phase are provided in the next sections. Tab. 1 shows the notations used to describe the proposed algorithms.

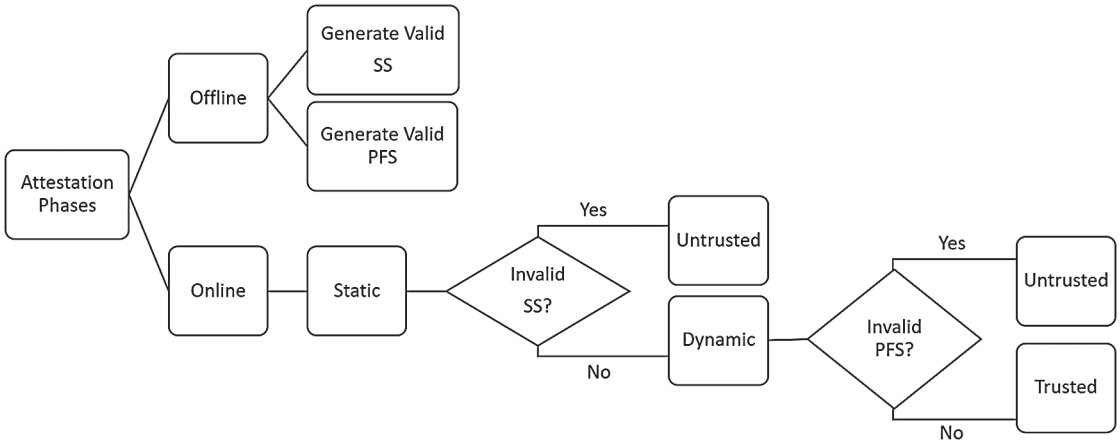

Figure 2: Design of dual attestation

4.1 Static Attestation

In this stage, the SS is computed by pseudo-randomly traversing the program memory of the IoT Application. The traversed memory addresses are generated by Phelix [16]. Phelix is a stream cipher with a built-in MAC known for its high speed. Like any stream cipher, it can be used as a cryptographic PRNG. Since encryption is not our aim, we discarded the ciphertext.

Assuming that application size will not exceed $(2^{32} - 1)$, we fix the highest significant 32 bits and randomly generate the lowest significant 32-bits of the memory address. This method will minimize overhead compared to randomly generating a 64-bit address. Phelix uses a 32-byte key, a 16-byte nonce, and generates a 16-byte MAC tag.

Phelix is executed as "many simple rounds" [16]. In our situation, the number of rounds (i.e., generated memory addresses) is initially determined by the program code's size in terms of its total number of memory locations. Then it is increased until it covers all program memory locations. As a result, the final number of iterations is determined by trial and error.

Table 1: Summary of notations

Notation	Description
SA	Source address
TA	Target address
NT	Node type (i.e., start/end of path, loop entry, normal node)
I	Input to the hash function which contains previous hash appended by current source and target addresses
H_M	Hash value of the main path
$H_{prev(M)}$	Previous hash value (cumulative hash till previous node) in the main path
L_{Depth}	Loop at specified Depth (e.g., L1: First nested loop)
$P_m L_{Depth}$	The m-path of loop at the specified depth
$H_{PmLDepth}$	Hash value of the m-path of Loop at the specified depth
$H_{prev(PmLDepth)}$	Previous hash value of $PmLDepth$

As shown in Fig. 3, in each round of Phelix, a 32-bit Pseudo Random Number (PRN) is generated, which can be used to represent a memory address. While the first generated PRN is based on a key and a nonce received from the Vrf, the generated PRN in each of the next rounds depends on the previously visited memory address's contents. Algorithm 1 shows the steps for generating the SS. Basically, each round depends on five state words ($w_0, \ldots, w_4$), two keywords (X_0, X_1), and the memory content ($MemoryContent$) of the previously generated address.

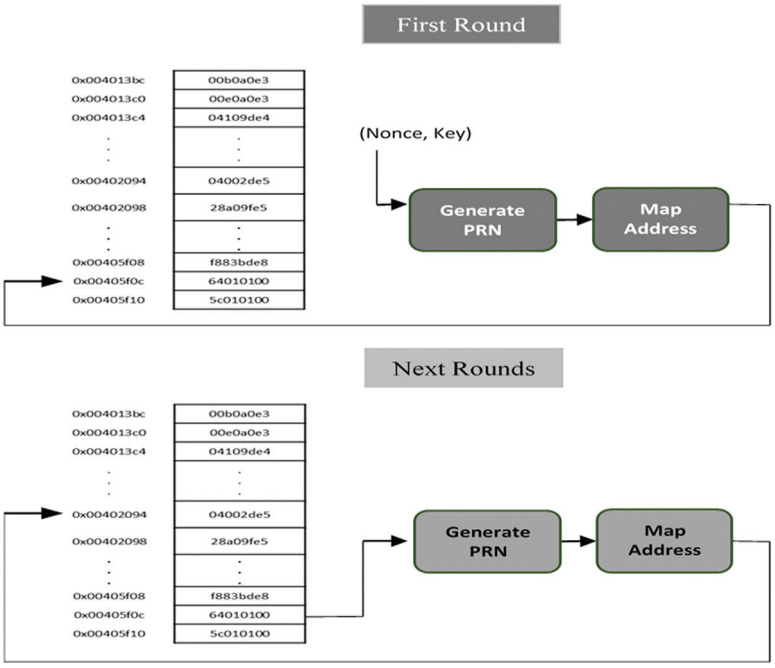

Figure 3: Static attestation routine (SAR)

Algorithm 1: Generating *SS*[1]

INPUT: *Rounds, Nonce, Key*
OUTPUT: *SS*
1: Receive Nonce and Key
2: Generate the working nonce (N) based on received *Nonce*
3: Generate the working key (K) based on received *Key*
4: Initialize the five state words based on N and K
5: Set *MemoryContent* $\leftarrow$ zero
6: For $n = -8$ to $n <= -1$) //first eight round
7: Compute the two key words based on N and K
8: Compute the temporary five state words based on the previous state words and the first key word
9: Compute the final five state words based on the temporary state words, the second key word and *MemoryContent*
10: End for
11: For $n = 0$ to $n <=$ Rounds
12: Do steps 7&8
13: *Compute PRN based on state words*
14: Compute *MemoryAddress* $\leftarrow$ *PRN* mod (*End_addr* $-$ *Start_addr*) $+$ *Start_addr*
15: *MemoryContent* $\leftarrow$ contents (*MemoryAddress*)
16: Do step 9
17: End for
18: *SS* $\leftarrow$ NULL
19: *MemoryContent* $\leftarrow$ size of (program) mod 4
20: For $n =$ *Rounds* $+ 1$ to $n <=$ *Rounds* $+ 11$
21: Do steps 7 to 9
22: Compute *PRN* based on state words
23: If $n >$ *Rounds* $+ 7$ // last four rounds
24: *SS* $\leftarrow$ append (*PRN*) to *SS*
25: End For

Please refer to [16] for more details about generating *N, K, keywords, state words* and *PRN*.

4.2 Dynamic Attestation

This phase aims to check whether the IoT application can be executed correctly. Our *PFA* routine tracks the IoT application at runtime and generates the Program Flow Signature (*PFS*) of the executed path. Since we wanted to capture the attested application's exact executed path, we constructed our *PFA* minutely, working on the application binary and not the source code. It is not practical to record every executed instruction or even their addresses because this will result in a very long signature and will significantly slow down the attested application. Therefore, we

only consider the dependent program flow instructions, computing a cumulative hash over these instructions. The generated *PFS* can be authenticated using MAC, where the secure world protects the key, and the *Vrf* sends the nonce.

For each node in the execution path, the hashing function H ($H_{previous}$, $N\_ID_{current}$) takes two parameters as input: the previous computed hash $H_{previous}$ and the current node ID $N\_ID_{current}$. Thus, it will result in a cumulative hash (i.e., *PFS*) where any change in the execution path will affect the PFS's value. For the first node in the execution path (N1), the previous hash Hprevious is zero. Since there are two valid paths (see Fig. 4), the final *PFS* is either H4a or H4b.

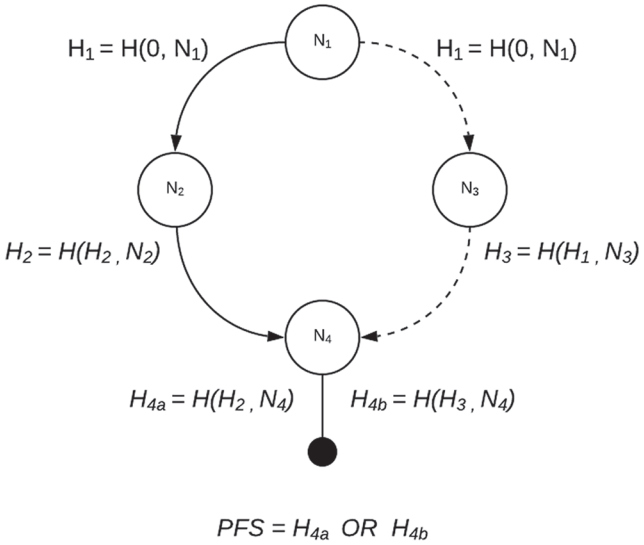

Figure 4: Simple control-flow graph (CFG)

Among the existing cryptographic hashing functions, we chose BLAKE2b for our hashing computation since it is speedy and has no known security issues [17].

4.2.1 Loop Management

Loops in the attested application code are another challenge because different iterations of loops may have different execution paths. Combining these paths with the main program path leads to a massive number of valid *PFSs*. Nested loops make the situation even worse. Like C-FLAT, we used to handle this problem is by considering each loop as a new execution path instead of combining it with the main path. In Fig. 5, when entering a loop, the main path hash is saved, and a new path computation starts. The hash of N_3 (i.e., loop entry/exit) is measured as H_{3b} (0, N_3), and the next nodes in the loop are measured cumulatively to H_{3b}. Besides, at the loop exit, the hash of the loop entry/exit node N3 is cumulatively measured with the hash of the main path H_{3a} (H_2, N_3).

Unlike C-FLAT, SAPEM does not only record the total iteration count but also determines the specific iteration ids for each path. Depending on the number of iterations per loop in the attested application, we can choose whether to report the iteration ids (H, <a, b, c>), where a, b, c represents the iteration number, or simply use the total iteration count for each path (H, #n), where #n represents the total count of iterations for the specific loop path. At the loop exit,

there will be a different number of paths. All of them, along with the main path hash, represents the *PFS*.

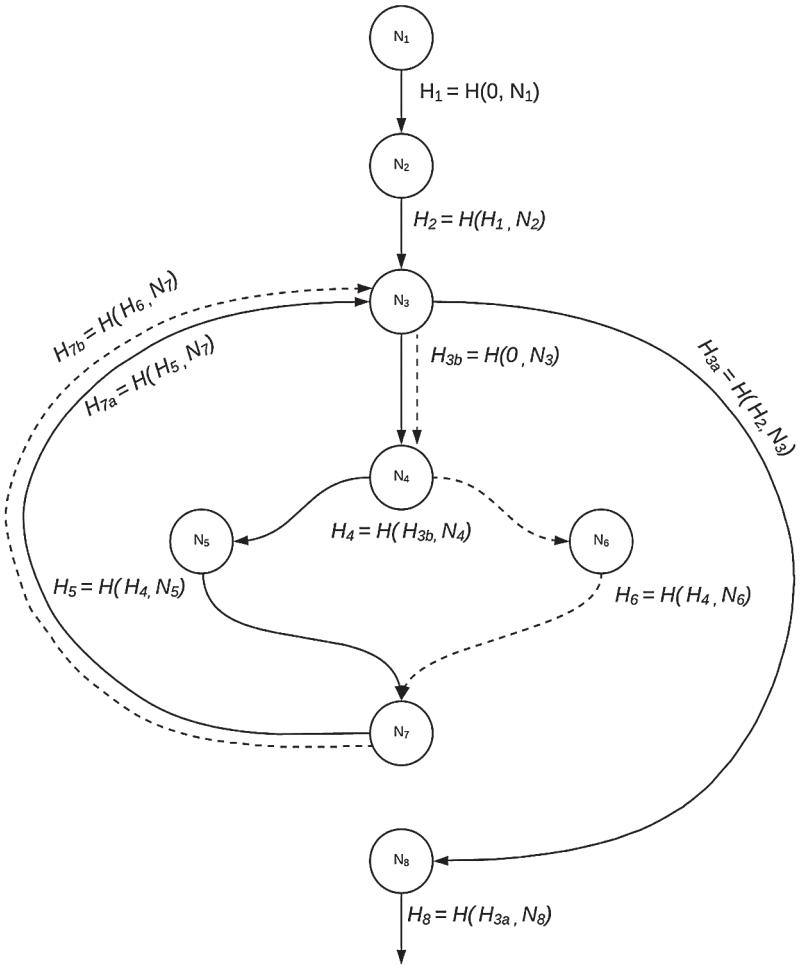

Figure 5: Loops management

The other problem introduced by loops is the iterations. A loop might be iterated hundreds of times. Computing the hash for each iteration, as was done in C-FLAT, degrades the attested application's performance significantly. Even if the number of iterations is small, it still adds unreasonable overhead. The other problem introduced by loops is the iterations. A loop might be iterated hundreds of times. Computing the hash for each iteration, as was done in C-FLAT, degrades the attested application's performance significantly. Even if the number of iterations is small, it still adds unreasonable overhead. Our approach tackles this problem by measuring the cumulative hash only the first time a loop path is executed. All node ids included in the path are also stored. Thus, whenever a path is repeated (e.g., in the next iterations), the process would simply compare the node ids instead of computing the cumulative hash for each node in the path. In other words, a loop with two different executed paths, each of which is repeated ten times, requires a computing hash of only two paths instead of 20 paths.

Algorithm 2 shows the steps for generating the PFS.

Algorithm 2: Generation of the *PFS*

INPUT: *SA, TA, NT*
OUTPUT: *PFS*
1: Initialize blake2 input $I \leftarrow SA + TA$
2: If $NT = 1$
3: Append $H_{prev(M)}$ to I
4: Initialize and update H_M
5: Else if $NT = 2$
6: Append $H_{prev(M)}$ to I, Update and Finalize H_M
7: Else if NT = 3
8: If $Depth = 0$
9: If $TA \neq SA + 4$ //enter loop
10: Initialize L_{Depth} and $P_m L_{Depth}$
11: Store (SA_1, TA_1) in $P_m\ L_{Depth}$
12: Initialize and Update $H_{Pm\ LDepth}$
13: Else Append $H_{prev(M)}$ to I and Update H_M
14: Else if $SA = SA\ (L_{Depth})$
15: UpdateLoopSig ()
16: If $TA \neq SA + 4$
17: Initialize*Iteration*
18: *Else FinalizeLoopSig( )*
19: If $(Depth \leftarrow Depth - 1) = 0$
20: Append $H_{prev(M)}$ to I and Update H_M
21: Else if InSimilarPath ()
22: If CheckNode $(SA, TA) \neq$ True
23: $m = m + 1$
24: Initialize $P_m L_{Depth}$ *and Initialize* $H_{Pm\ LDepth}$
25: $P_m\ L_{Depth}\ (SA_1, TA_1) = P_{similar}\ L_{Depth}\ (SA_1, TA_1)$
26: Initialize and Update $H_{Pm\,LDepth}$
27: $K = 2$
28: Repeat till $P_{similar}\ L_{Depth}\ (SA_{current-1}, TA_{current-1})$
29: $P_m L_{Depth}\ (SA_k, TA_k) = P_{similar}\, L_{Depth}\ (SA_k, TA_k)$
30: Append $H_{prev(Pm\,LDepth)}$ to I and Update $H_{Pm\,LDepth}$
31: Store (SA_1, TA_1) in $P_m\ L_{Depth}$, Append $H_{prev(Pm\,LDepth)}$ to I and Update $H_{Pm\,LDepth}$
32: Do Step 31
33: Else Do Step 31
34: Else if $NT = 4$
35: If $Depth > 0$
36: If InSimilarPath()
37: Do Steps 21 to 31
38: Else Do Step 31
39: Else Append $H_{prev(M)}$ to I and Update H_M

4.2.2 Conditional Branch Management

We introduce a new approach for handling the interception of conditional branches. Our approach can intercept all conditional branches rather than inferring untaken ones. Typically, the conditional branches are preceded or combined with some comparison. Depending on the comparison results, the process determines whether to branch to target or continue executing the subsequent instruction. For the sake of intercepting the instruction, all branching instructions are rewritten with link instructions, with the target address set to another address (i.e., the hook address) in order to transfer the control to the Hook, as shown in Fig. 6. The source and the original target addresses are stored in read-only memory.

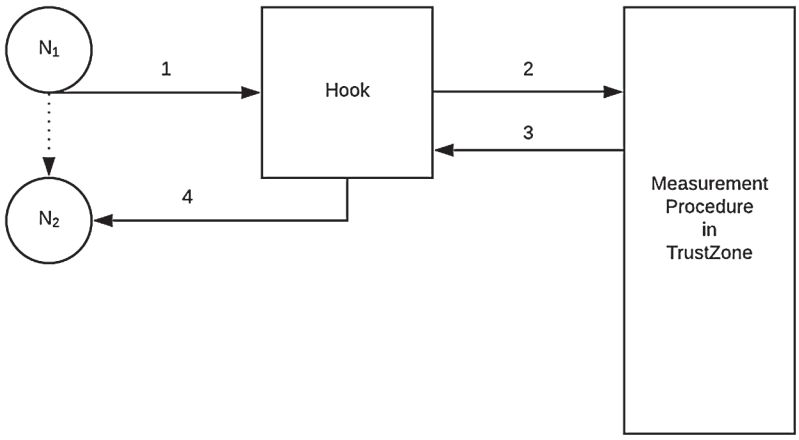

Figure 6: Control transition

This method also allows the attestation process to intercept the conditional branch, whether or not the original branch target will be taken.

The next step is to extract the correct target (either the target in the original branch instruction or the next instruction). Additional information is added to the branch table: mask and masked flags. Tab. 2 shows the mask and masked flags for common conditional branches in ARMv8 processors. For each different type of conditional branching, a different mask represents the register flags to be set by the comparison. During execution, the register flags are ANDed with the mask; then the result is compared with the masked flag. In the case of a match, the process will branch to the target address stored in the branching; otherwise, execution will continue.

5 Implementation

We implemented SAPEM on Raspberry Pi as a proof of concept. As discussed in Section III, the measurements (i.e., hash) of the *Prv*'s execution path should be done in an isolated secure environment. *Prv* We used Rpi3, which has ARMv8-A architecture [18] that features TrustZone-A embedded security technology. On the other side, the *Vrf* is represented by a computer with a processor speed of 2.7 GHz, 16 GB RAM, a 64-bit Intel Core i7 CPU, and Ubuntu-16.0 OS on top of it.

We used C language to implement the SAR and the PFA. Other implementation details are discussed in the following sub-sections.

Table 2: Conditional branch management

Condition	Affected register flags				Mask	Masked flags
	N	Z	C	V		
EQ	×	1	×	×	0 × 4	0 × 4
NE	×	0	×	×	0 × 0	0 × 4
CS	×	×	1	×	0 × 2	0 × 2
HS	×	×	1	×	0 × 2	0 × 2
CC	×	×	0	×	0 × 0	0 × 2
LO	×	×	0	×	0 × 0	0 × 2
MI	1	×	×	×	0 × 8	0 × 8
PL	0	×	×	×	0 × 0	0 × 8
VS	×	×	×	1	0 × 1	0 × 1
VC	×	×	×	0	0 × 0	0 × 1
HI	×	0	1	×	0 × 2	0 × 6
LS	×	1	0	×	0 × 4	0 × 6
	×	0	0	×	0 × 0	
	×	1	1	×	0 × 6	
GE	1	×	×	1	0 × 9	0 × 9
	0	×	×	0	0 × 0	
LT	1	×	×	0	0 × 8	0 × 9
	0	×	×	1	0 × 1	
GT	1	0	×	1	0 × 9	0 × d
	0	0	×	0	0 × 0	
LE	1	1	×	1	0 × d	0 × d
	1	1	×	0	0 × c	
	0	1	×	1	0 × 5	
	1	0	×	0	0 × 8	
	0	0	×	1	0 × 1	

5.1 The Challenge of Developing in a Trusted Execution Environment

We used OP-TEE for developing in the TrustZone. OP-TEE is an open-source platform for the Trusted Execution Environment (TEE) provided by Open-TEE in 2014 [19]. It is built on top of ARM Cortex-A cores as a companion to the normal world (i.e., Linux kernel) using TrustZone technology. It implements two APIs-TEE Internal Core API and TEE Client API that complies with the Global Platform API qualifications. While the first is exposed to Trusted Applications, the second is used to facilitate communications with a TEE.

5.1.1 OP-TEE Setup on Rpi3

Although Raspbian is the default OS for Rpi, it is not supported by OP-TEE. Instead, OPTEE uses a Buildroot environment to build a customizable distribution of the embedded Linux system. Setting up the environment was not a straightforward process, as we needed to configure the files to our requirements and then use Buildroot to build the kernel and the root file system image. Since Buildroot can build a limited Linux distribution, the resulting Linux image did not support commands like "Sudo apt-get install." These limitations burdened the

development process in the OP-TEE platform. Whenever we needed to install software, we edited the configuration file using "/../optee/build/common.mk" to add the software package name.

We needed to edit additional configuration files and finally rebuild the system. As a result, the software was installed and added to the image produced by Buildroot.

After configuring the files and building the image, we flashed the image onto the SD card. We used a USB-UART cable for serial communication between the PC and the Rpi to access the Rpi system. It is important to note that the HDMI connection was not provided by OP-TEE, even though Rpi supports it. We used a Picocom software program to open the serial port as follows:

Picocom—b 115200/dev/ttyUSB0

5.1.2 Compilation Scenarios

Due to the limitations of the Linux distribution provided by OP-TEE, compiling programs directly on this platform is not possible. After exhaustive research, we selected two compilation scenarios as follows:

First Scenario: Place the program source files in the examples folder (/../optee/optee_examples) provided by OP-TEE. It is set up to be compiled as the OP-TEE root file system is built. Therefore, all source files copied to this folder (e.g., optee_examples) are automatically compiled and placed in the right locations in the resulting root file system image.

Second Scenario: Compile the PC programs using cross toolchains, then place the resulting binaries in the folders (/../devel/optee/out-br) and use them with Buildroot to produce the root file system image. Specifically, the binary file (for the normal world) is copied to (/../devel/optee/out-br/target/usr/bin) and the other file (for the secure world), which has a .ta extension, is copied to (/../devel/optee/out-br/target/lib/optee_armtz). Then we built the root file system using Buildroot. Before building the image, previous images must be deleted from (/../devel/optee/out-br/images); otherwise the copied files would not be included in the next built image.

For programs that run only in the normal world, Qemu, which is a system emulator, is used to test programs on the PC before rebuilding the OP-TEE platform to include these programs. However, it is more complicated for programs that work in the secure world, as Qemu does not emulate the secure world. Therefore, we had to conduct the above compilation scenarios repeatedly when we were developing the attestation programs.

5.1.3 Running Secure World Programs

OP-TEE defines functions to initialize/finalize a session, open/close a session, and run a specific function in the secure world. The structure of the secure and normal world programs should include these functions to communicate with each other. Besides, each secure world binary has a Unique User IDentification (UUID), which is used for communicating with the normal world binary. Many online websites that can be used to generate the UUID. Consequently, the corresponding files that define the UUID should be modified to the new UUID. However, details about exchanging the parameters or setting the UUID were not provided by OP-TEE; we had to search for this information in the source code and header files used to compile these programs.

5.2 Extracting and Rewriting the Branch Instructions

As we explained, tracing the execution path requires intercepting each of the control-flow instructions in the binary. Calling subroutines in programs that comply with ARM Architecture Procedure Call Standard (AAPCS) are obtained through a branching with link instruction (bl)

and a dedicated return instruction (ret). When bl is executed, the return address is saved in the processor register (lr). When returning, ret instruction branches to the address in the lr register. While direct branching is obtained through b, conditional branching is provided by b.cond.

We used Radare2 for extracting the information (i.e., the source address, the target address, and the node type) of every control-flow instruction in the binary except ret, as ret can have a different target every time. Radare2 is an open-source reverse engineering framework used to analyze, assemble, and disassemble binaries of many different architectures. Despite the usability of Radare2 from the command line, it lacks some of the ARM assembly opcodes. We had to edit the source code of Radare2 to add these opcodes. The extracted information by Radare2 was then stored in the branching table with the mask and the masked flags in case of conditional branches, as shown in Tab. 2.

Besides, we used Radare2 to replace the branching instructions by <bl #hooking_address>, where the #hook_address is different for each branching type. We replaced instructions with bl as it allowed us to retrieve the instruction's source address because bl instructions store the return address, which is the current address plus 4, in the lr register. Consequently, we could extract the lr value and subtract 4 to get the source address.

We created an automated tool for both of the above processes (e.g., extraction and replacement) using Python programming language.

6 Results

In the first subsection, we simulate the attacks mentioned in section II and check the resulting signatures. In the second subsection, we compare the performance of *PFA* with that of C-FLAT.

6.1 Attack Detection

Below, we demonstrate the effect of code injection, ROP, and data attacks on SS and PFS. While Tab. 3 shows examples of the valid signatures collected offline, Tab. 4 summarizes the received signatures of the attestation process (online).

Code Injection: we used Radare2 to inject malicious code into the binary. We changed only one assembly instruction (less than 4 bytes) to detect the static attestation phase capability. The change in the binary was detected by the static attestation, making the resulting *SS* different.

ROP: ROP attacks use the return instructions of the system's connected gadgets. It overflows the stack to rewrite the target of the return instruction. Here, we simulated the effect of the ROP to change the target address of the return instruction. The attested code contains one loop and any functions called before entering the loop. The signature received after the ROP attack was different from the valid one. However, only the main path signature was affected; the loop signature was identical to the valid one. Although the simulated ROP attack redirected the return instruction to another location (execute another instruction), it was still executed before the loop.

Data Attack on Control Flow: As we explained earlier, our goal is to detect data attacks that hijack the control-flow. Attacks that corrupt data without affecting the control-flow are out of our scope. Here, we simulated the effect of corrupting a variable relied upon by the loop execution path. The *PFA* detected the deviation of the control-flow caused by this attack. *PFA* This deviation was reported in the received *PFS*, where only the loop path signature was different from the original. Specifically, iteration three was reported in the second loop path instead of the first path.

Table 3: Valid signatures

Valid static signature	9cc142a28bd003d9481e7a41187f5678
Valid Program Flow Signature	Main path: B16937263b7a41a218cfaeecb4772f83f20 ca8d0d65616b38d0d25cc362e2a3dbaa764da938 59167c0009ca403cc2c04007a366dcc709da9eb6 0fe0761b153c Loop0: Previous Signature: 543234756ff8eb4ee02955323f150d9d305dc4a 683a68329f552b23a9fd875e047e56553200 6fde7d81c7f94f82ad4ff600760ddc907a16055 0c6ea827245cd0 Loop Signatures: Total Paths = 2 Path#1: Iterations 0 1 2 3 8b79932bca2a5884add12554cb3ffc165064b 5b557ac6bedfb3773558d4fcd1f07be7f9a8417 b090259abc85ca372cc298205cde3c98a04587f0a4 7b8f7fd2bc Path#2: Iteration 5 De5074b0c3b81f0d66b3b1ee0d08d41f3c8884 afa223845c2a703a28b85ff87f65c0157883e 3ef 661fc95fb1b5ee8bde48fefb3b34fb858ef91c33 3a539c

6.2 Performance

We compared the real execution time of our *PFA* to that of C-FLAT by using the Linux command time. The time command records the time from the start of the attestation program till completion. The recorded time does not include the communication time because the evaluation is done on the *Prv* device. In this comparison, we focus on loops, as they are the most challenging part of the *PFA* and the vital difference between SAPEM and C-FLAT. The execution time is analyzed according to the number of loops, the number of decision nodes per loop, the number of iterations, and the number of different executed paths per loop. In addition to execution time, we also compare the signature size for both *PFA* and C-FLAT.

Since we aim is to measure the performance of SAPEM for IoT applications with different complexities, we have created a testing code that doesn't simulate a specific IoT functionality but can be adjusted to represent programs with different numbers of decision nodes, loops, and iterations. Building different kinds of IoT applications is not our aim, nor is it practical. However, the testing code is run on a simulated IoT environment (i.e., RaspberryPi 3) that uses TrustZone.

We looked at different open source IoT applications to choose the parameters' values (#loops, #decisions nodes, #iterations).

Table 4: Analysis of received signatures

Attack type	Received signature	
	SS	*PFS*
Code Injection	Different from the valid SS	Not executed
ROP	Similar to the valid SS	Similar loop paths signatures including iterations, but different main path signature
Data Attack	Similar to the valid SS	Similar main and loop paths signatures, but different iterations for each path

The Number of Loops: We fixed the number of decision nodes outside the loops to evaluate the number of loops' effect on execution time. Moreover, we repeated the same loops with each trial to keep the same number of decision nodes and paths for each of the loops. The number of loops was gradually increased from 0 to 40, each loop having ten iterations. As shown in Fig. 7, when the number of loops is zero, *PFA* and C-FLAT have similar performance because the *PFA* mainly aims to reduce the overhead caused by loops. As the number of loops is increased, the difference between the two processes steadily increases. We also allowed for other significant factors for each loop to show the effect of changing the number of loops in isolation.

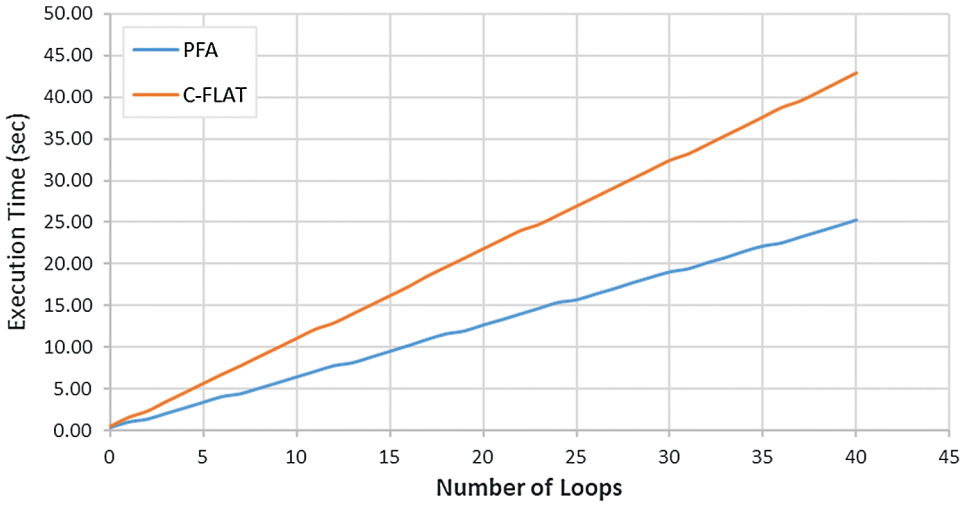

Figure 7: Execution time with various number of loops

The Number of Decision Nodes per Loop: We experimented and evaluated the effect of the number of decision nodes in the loop based on execution time. In this experiment, we considered only one loop with ten iterations, but the number of decision nodes in the loop varied from 2 to 36. As a result, we recorded a different number of paths for each run of the experiment. As

expected, increasing the number of decision nodes increases the difference in the execution time between *PFA* and C-FLAT (Fig. 8). It is interesting to note that execution time fluctuates slightly for both *PFA* and C-FLAT simultaneously. This fluctuation is due to the changing number of decision nodes executed in the attested binary. The X-axis in Fig. 8 represents the number of decision nodes in the attested application source code.

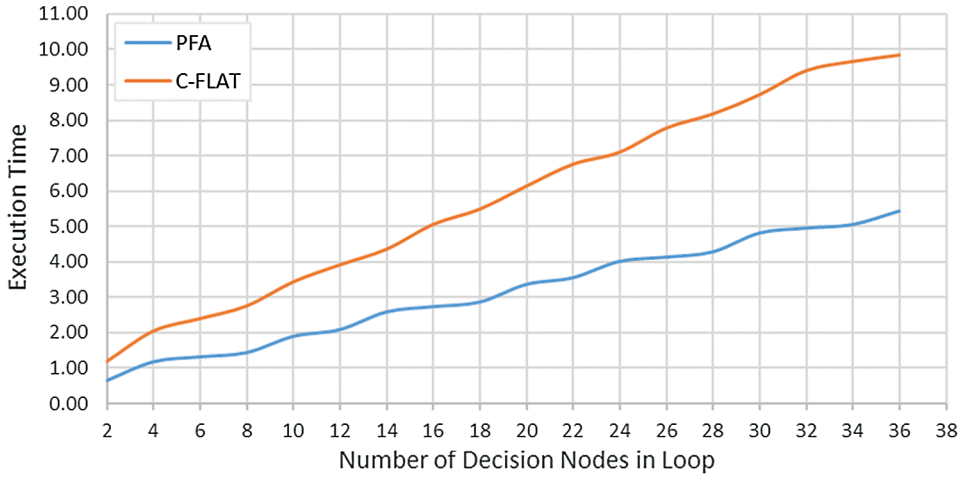

Figure 8: Execution time with increasing numbers of decision nodes in the loop

The Number of Iterations: We evaluated the effect of the number of iterations on execution time in both PFA and C-FLAT by fixing the number of decision nodes in the source code. The evaluation was done on one loop by increasing the number of iterations five steps each time. In Fig. 9, In Fig. 9, *PFA's* effect compared to C-FLAT is increasing with the increased number of iterations. In the beginning, the difference in execution time increases steadily. When more iterations are added, the difference rate becomes less predictable because the number of paths executed for each specific iteration is related to the iteration value. It is clear from Fig. 9 that PFAthe same paths are executed for the first (0–25) iteration.

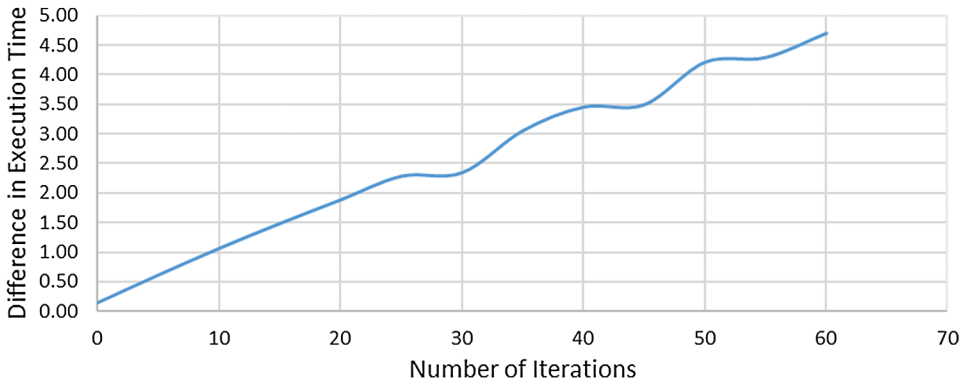

Figure 9: Difference of execution time between *PFA* and C-FLAT with various number of iterations

The Number of Different Paths Executed per Loop: The last factor we considered in our comparison of execution time between PFA and C-FLAT was the total number of different paths in the loop. In Fig. 10, the difference in the execution time between *PFA* and C-FLAT falls as the number of distinct paths executed in a loop increases. However, the difference is reduced because there is also a hidden increase in the *PFA's* difference PFA for the repeated paths.

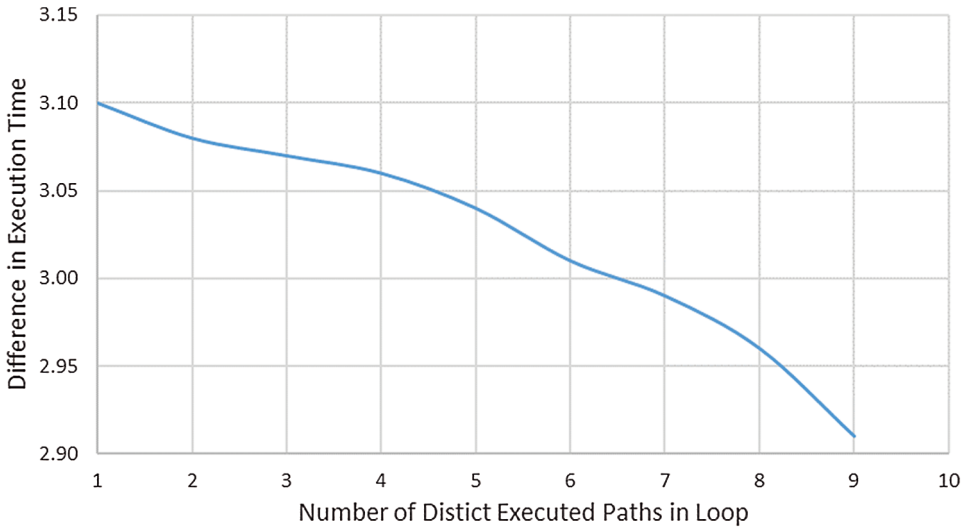

Figure 10: Difference in execution time between *PFA* and C-FLAT with various number of paths in loop

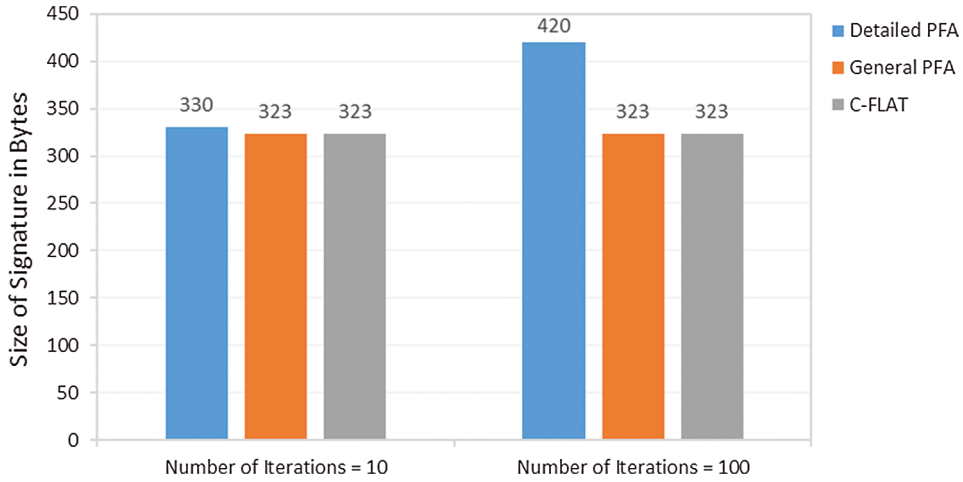

Figure 11: Comparison of signature size between *PFA*, detailed *PFA*, C-FLAT

The Signature Size: We also compared *PFA* and C-FLAT in terms of signature size. The size for each algorithm depends on the number of loops. Besides, if a detailed *PFA* is needed, the *PFA* signature depends on the number of iterations. While general *PFA* and C-FLAT signature

size are calculated using Eq. (1), a detailed *PFA* is calculated using Eq. (2).

$$sz + L * sz + (Pl1 + \cdots + Pln) * sz + (Pl1 + \cdots + Pln) \quad (1)$$

$$sz + \mathrm{L} * \mathrm{sz} + (Pl1 + \cdots + Pln) * sz + (Il1 + \cdots + Iln) \quad (2)$$

where sz is the main path signature size, L is the number of loops, Pln is the number of distinct executed paths in loop n, and Iln is the number of iterations in loop n. Fig. 11 shows the calculated signature size by setting $L = 1$ and $Pl1 = 3$. While C-FLAT and general *PFA* have the same signature, a detailed *PFA* signature increases by only 2% in the case of 10 iterations. However, the increment percentage is more significant for 100 iterations (30%).

7 Discussion

In this section, we conduct a detailed discussion about the results for both of the evaluated aspects: Attacks Detection and Performance. A summary is provided in Tab. 5.

Attacks: We demonstrated the effect of code injection, ROP, and data attacks that affect control-flow. As shown in Tab. 4, the attacks were detected and reflected precisely in the signature (e.g., invalid main path signature and invalid loop path count). It is important to realize that injected malicious code that does not affect control-flow can be detected only by static attestation. Therefore, both attestation techniques, static and dynamic, are required to detect attacks. Other control-flow attacks such as JOP and Function-Reuse attacks can be detected regardless of the method used to launch them because they all result in changing the target addresses, which affects the control-flow. As SAPEM intercepts each single branch instruction in the attested application, it can detect these attacks. On the other hand, attacks that corrupt data without affecting control-flow can not be detected by either the static or the dynamic parts of SAPEM. Also, physical attacks are out of our scope.

Performance: The results in Figs. 7–9 show that *PFA* execution time performance is strongly linked to loops. As we increase the number of loops, iterations, or decision nodes in the loop, the attestation overhead decreases compared with that of C-FLAT because, unlike C-FLAT, *PFA* does not compute the hash for each iteration; instead, it computes the hash for the first appearance of each path in the loop. When a path is repeated, the process will recognize it and compare its current nodes to the stored nodes instead of hashing it. It is to be noted that a simple comparison operation is less complicated than the hashing function in terms of the number of executed instructions.

According to C-FLAT, hashing causes 80% of the incurred overhead. With this in mind, it is plausible that as we continue to skip hashing for repeated paths, performance overhead decreases significantly compared to that of C-FLAT. However, the first time a distinct path appears, *PFA* calculates the hash and conducts additional operations for storing nodes. This added overhead is balanced by the reduced overhead caused by repeated paths. Consequently, with an increase in distinct paths and a decrease in iterations or repeated executed paths, *PFA's* performance degrades, as *PFA* shown in Fig. 10. As a solution, we can divide the path into sub-paths categorized according to common factors and store them separately instead of storing the whole path. As a result, the next time a distinct path appears, the hash will not be computed for the whole path; instead, hashing is done only for the new, different part. However, this solution does not apply to all loop cases. For example, applying it to simple loops (i.e., loops with fewer decision nodes or distinct paths) will negatively affect the execution time. The best case for applying this suggested

sub-path solution is when there is a loop with many decision nodes and has a long mutual path (i.e., a large number of sequentially executed decision nodes) between the distinct paths.

Table 5: Evaluation summary

	Evaluated aspect	Results
Attacks	Code injection	Detected by SAR. *PFA* detects it only in case of diverting control flow.
	ROP	Detected by *PFA*.
	JOP	We have theoretically demonstrated (*PFA* can detect it).
	Function-Reuse	We have theoretically demonstrated (*PFA* can detect it).
	Data attacks on control flow	Detected by *PFA*.
Performance comparison with C-FLAT	Number of loops	*PFA* exceeds C-FLAT performance. The difference in performance increases with the increase of the number of loops.
	Number of iterations	*PFA* exceeds C-FLAT performance. The difference in performance increases with the increase of the number of iterations, but at the same time, detailed *PFS* gets bigger in size. In this case, it is suggested to use general *PFS*.
	Number of decision nodes in loop	*PFA* exceeds C-FLAT performance. The difference in performance increases with the increase of the number of decision nodes in loop.
	Number of distinct paths	*PFA* performance degrades with the increase of the number of distinct paths. A suggested solution is provided.

Another point to discuss is the signature size (Fig. 11). We implemented two versions of *PFA*, where the only difference was in the signature. The detailed one shows the path signatures along with exact iteration numbers. Only in the case of a large number of iterations were signature sizes significantly increased. In these cases, we suggest using the standard *PFA* that shows only the total number of iterations per path, unless details are needed.

Another notable factor is the memory used by *PFA* for storing the path. At first glance, one might imagine that *PFA* uses much memory. It is not true as it stores the paths only during loop execution, and once it exits the loop, the stored paths are discarded, meaning that the same memory can be used for each loop. Nested loops can duplicate this effect, as the outer loops are retained during the inner loops' execution. However, storing a path means storing a

group of nodes where each node is represented by 16 bytes (source and target addresses). For example, storing 100 nodes requires only around 1.5 KB of space. As a workaround for the IoT applications that need a large memory for storing nodes, we can store the lowest significant half of the addresses, assuming that the IoT application is not greater than (2^{32}), which is a reasonable assumption that results in reducing the needed memory for storing nodes to half. Memory issues can be solved by providing larger memory, but timely execution is critical for the IoT application's functionality.

8 Related Work

Existing remote attestation techniques can be divided mainly into two categories: Static attestation and dynamic attestation. The proposed approaches in each category are further divided into software/hybrid and hardware.

8.1 Static Attestation

Several approaches are proposed using static attestation in the literature. Seshadri et al. proposed the first SoftWare-based ATTestation for Embedded Devices (SWATT) by pseudo-randomly traversing memory locations [20]. Like most static attestation techniques, SWATT depends on time constraints, making it inapplicable for multi-hop networks. Yang et al. [5] added some steps to SWATT to make it suitable for multi-hop networks by having the relay nodes stamp the packets' reception time before sending them to the verifier. However, this technique is inconvenient for hybrid networks as it assumes that all nodes have a similar time-delay. Park et al. strengthen attestation against attacks by randomizing the hash function and using it in conjunction with time constraints [21]. Other techniques that are time-independent rely on filling empty memory with noise [22,23], or using randomization, obfuscation, encryption, and self-modified code [24]. While compression attacks can compromise the former technique, the latter technique is not implemented. Little research focuses on detecting physical attacks either by including the checksum of the Physical Unclonable Functions (PUFs) of the device [4] or by periodically broadcasting a message and then examining the log of these messages [25]. Each of these has its drawback: the first approach results in high false-positive rates for identifying compromised nodes, while the second requires large storage. Another proposed attestation technique known as distributed attestation splits the attestation process between the attested devices [26–29], where the latter used a trusted anchor. The proposed schemes [26] have downsides, including a compromised head cluster and large communication overhead.

While all of the above research is based on software only, others have proposed hardware-based static attestation [30,31]. Hardware-based attestation relies on tamper-resistant hardware such as TPM. For instance, remote attestation has been proposed for Wireless Sensor Networks (WSN) that requires all sensor nodes to be equipped with TPM [31]. However, TPM and other successive techniques are more common in PCs and not suitable for IoT devices due to their proportionately high cost [32,33]. Hybrid-based static attestation [34,35] strengthens software-based attestation using minimal hardware trust anchors. Recent hybrid-based attestation techniques have introduced smart meters and swarm attestation (i.e., group attestation) to tackle the scalability issue [36,37]. As an alternative to the typical two-way attestation technique (i.e., challenge-response), Song et al. [38] have introduced one-way attestation to prevent network attacks.

Nunes et al. [39] proposed VRASED, the first formally verified SW/HW remote attestation. It implements a verified cryptographic software and a verified hardware design to assure the

correct design and implementation of remote attestation security properties. On the contrary, SIMPLE is a-based software-based, formally verified remote attestation that uses a memory isolation technique called Security MicroVisor (SμV) [40].

Nevertheless, given the static nature of all the above-mentioned techniques, none of them can detect runtime attacks.

8.2 Dynamic Attestation

In contrast, dynamic attestation techniques have not been the object of many investigations. Some authors have proposed attestation protocols [41,42] that are based on Indisputable Code Execution (ICE) to detect compromised nodes and establish secure keys, respectively. However, Castelluccia et al. [8] showed that adversaries could manipulate the proposed attestation's execution in [40] without being detected. Zhang et al. [43] proposed an attestation technique that uses data guards to surround program data and includes these data guards' checksum in the attestation, but it cannot detect objects' overflow attack blocks or array elements, nor is it effective against string format attacks. Other schemes like property-based and semantic-based attestations were proposed to check the program behavior [44]. While the first examined behavioral characteristics, the other worked on the Java byte code level [45] to enforce local policies. While each of these examples functions on a high level, neither could detect control-flow attacks at the binary level. Researchers also proposed schemes that demonstrated low-level properties of the attested applications, such as checking the integrity of functions base pointer [46] or exploring taint analysis on return instructions provided that taint analysis causes high-performance overhead [47]. However, these are not representative of the overall program flow of the attested application. Furthermore, approaches that focus only on the controller's software integrity cannot detect attacks at the application layer programs [48,49].

In [2], the researchers discussed the failure of existing software and hybrid attestation to detect control-flow attacks and suggested taking the hash of the execution path. The authors in [50] proposed Control Flow ATtestation (C-FLAT), where the Prv response represents the executed path at the binary level. Although C-FLAT could detect control-flow attacks, Ahmed et al. argued that it incurs high-performance overhead [51]. Another competing approach to C-FLAT is DO-RA, which can detect runtime attacks based on the Data-Oriented Control Flow Graph of the target application [52]. Unlike C-FLAT, DO-RA can detect injected malicious code that would not change the program flow and data-attacks that affect the actuator functionality but executing a valid program path. However, it incurs a larger overhead than C-FLAT in terms of memory and time. Abera et al. [53] proposed DIAT, an autonomous collaborative attestation that verifies data correctness using control-flow attestation, but it only considers the modules that process these data. On the other hand, recent hardware-based attestation techniques were proposed to undertake significant performance overhead caused by attesting and recording the control-flow path at the binary level [54–57]. Unfortunately, the hardware requirements required for these techniques make them impractical solutions for IoT systems. In a similar context to dynamic remote attestation, some enforcement techniques were proposed for preventing runtime attacks [58]. For example, Code Pointer Integrity focuses on finding irregular flows in program execution, and μRAI enforces the Return Address Integrity (RAI) property on MCUS [59,60]. Another technique called Control Flow Integrity guarantees that the executed path is a valid one in the CFG [61,62]. However, neither of these techniques can detect data attacks that execute unauthorized but valid CFG paths. The proposed approach in [63] enforces the control-flow integrity based on the application CFG, but it only considers indirect branches. Besides, Larsen et al. proposed a technique that

randomizes code at a low level; this makes the hijacking process more difficult for potential adversaries but does not make them impossible [64].

9 Conclusion and Future Work

Many essential security threats of IoT applications can be mitigated by remote attestation. Most of the existing remote attestation techniques are either static processes, which cannot detect runtime attacks, or dynamic hardware-based processes, which cannot be plausibly used in the context of IoT devices. In this research, we proposed a dual attestation where static and dynamic attestation complements each other. Our approach, SAPEM, does not require hardware extensions; instead, it utilizes trust anchors, which are increasingly used in recent microcontrollers. While the first part of SAPEM (static) can detect any change in the code from its original state, the second part (dynamic) can detect runtime attacks that hijack the control-flow. We evaluated SAPEM compared to C-FLAT, and our approach resulted in better coverage for attacks and better performance at the same time.

The implementation of SAPEM was challenging due to the use of TrustZone technology. While the platform we used (OP-TEE) provides the required environment, the development process was complicated by insufficient user details and the limitations of the normal world OS supported by OP-TEE. However, we managed these challenges through exhaustive online searches of similar issues and digging into the source files of the OP-TEE platform to figure out how to perform tasks. Our ongoing work includes the generalization of PFA's optimization technique to functions with similar repeated loops during program execution. PFAIn the future, we aim to use another technique to detect pure data attacks and reduce the number of attested paths from a security perspective. We're planning to use binary analysis tools such as Binary Analysis Platform (BAP) to find vulnerable paths and affected memory locations, significantly reducing the execution time and detecting data attacks, whether they are affecting control-flow or not.

Acknowledgement: We would like to thank OpenUAE Research and Development Research Group at the University of Sharjah.

Funding Statement: We would like to thank OpenUAE Research and Development Research Group at the University of Sharjah for funding this research project.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] Y. H. Hwang, "Iot security & privacy: Threats and challenges," in *Proc. of the 1st ACM Workshop on IoT Privacy, Trust, and Security, IoTPTS 2015*, Singapore, pp. 1, 2015.

[2] T. Abera, N. Asokan, L. Davi, F. Koushanfar, A Paverd *et al.*, "Things, trouble, trust: On building trust in IoT systems," in *Proc. of the 53rd Annual Design Automation Conf.*, Austin, TX, USA, pp. 1–6, 2016.

[3] R. V. Steiner and E. Lupu, "Attestation in wireless sensor networks: A survey," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, pp. 1–31, 2016.

[4] A. Ibrahim, A. R. Sadeghi, G. Tsudik and S. Zeitouni, "DARPA: Device attestation resilient to physical attacks," in *Proc. of the 9th ACM Conf. on Security & Privacy in Wireless and Mobile Networks*, Darmstadt, Germany, pp. 171–182, 2016.

[5] X. Yang, X. He, W. Yu, J. Lin, R. Li *et al.*, "Towards a low-cost remote memory attestation for the smart grid," *Sensors*, vol. 15, no. 8, pp. 20799–20824, 2015.

[6] D. Fu and X. Peng, "TPM-based remote attestation for Wireless Sensor Networks," *Tsinghua Science and Technology*, vol. 21, no. 3, pp. 312–321, 2016.

[7] F. Armknecht, A. R. Sadeghi, S. Schulz and C. Wachsmann, "A security framework for the analysis and design of software attestation," in *Proc. of the 2013 ACM SIGSAC Conf. on Computer & Communications Security, CCS'13*, Berlin, Germany, pp. 1–12, 2013.

[8] C. Castelluccia, A. Francillon, D. Perito and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proc. of the 16th ACM Conf. on Computer and Communications Security, CCS'09*, Chicago, Illinois, USA, pp. 400–409, 2009.

[9] L. Li, H. Hu, J. Sun, Y. Liu and J. S. Dong, "Practical analysis framework for software-based attestation scheme," in *Int. Conf. on Formal Engineering Methods, ICFEM 2014*, Luxembourg, pp. 284–299, 2014.

[10] D. Ray, "Defining and preventing code-injection attacks," M.S. thesis, Department of Computer Science and Engineering, University of South Florida, Tampa, Florida, 2013.

[11] Y. Ruan, S. Kalyanasundaram and X. Zou, "Survey of return-oriented programming defense mechanisms," *Security and Communication Networks*, vol. 9, no. 10, pp. 1247–1265, 2016.

[12] T. Bletsch, X. Jiang, V. W. Freeh and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. of the 6th ACM Sym. on Information, Computer and Communications Security, ASIACCS '11*, Hong Kong, China, pp. 30–40, 2011.

[13] S. Chen, J. Xu, E. C. Sezer, P. Gauriar and R. K. Iyer, "Non-control-data attacks are realistic threats," in *USENIX Security Sym., SSYM'05*, Baltimore, MD, 2005.

[14] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena *et al.*, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Sym. on Security and Privacy (SP)*, San Jose, CA, USA, pp. 969–986, 2016.

[15] W. Feng, Y. Qin, S. Zhao and D. Feng, "AAoT: Lightweight attestation and authentication of low-resource things in IoT and CPS," *Computer Networks*, vol. 134, pp. 167–182, 2018.

[16] D. Whiting, B. Schneier, S. Lucks and F. Muller, "Fast encryption and authentication in a single cryptographic primitive," *ECRYPT Stream Cipher Project Report*, vol. 27, no. 200, pp. 5, 2005.

[17] J. P. Aumasson, S. Neves, Z. Wilcox-O'Hearn and C. Winnerlein, "BLAKE2: Simpler, smaller, fast as MD5," in *Int. Conf. on Applied Cryptography and Network Security, ACNS 2013. Lecture Notes in Computer Science*, Verlag Berlin Heidelberg, pp. 119–135, 2013.

[18] A. Holdings, "ARM cortex-a series programmer's guide for ARMv8-A-15.2—dynamic voltage and frequency scaling," 2015. [Online]. Available: https://developer.arm.com/documentation/den0024/a/preface.

[19] B. McGillion, T. Dettenborn, T. Nyman and N. Asokan, "Open-TEE—An open virtual trusted execution environment," in *2015 IEEE Trustcom/BigDataSE/ISPA*, Helsinki, Finland, pp. 400–407, 2015.

[20] A. Seshadri, A. Perrig, L. Van Doorn and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *IEEE Sym. on Security and Privacy, 2004—Proc. 2004*, Berkeley, CA, USA, pp. 272–282, 2004.

[21] T. Park and K. G. Shin, "Soft tamper-proofing via program integrity verification in wireless sensor networks," *IEEE Transactions on Mobile Computing*, vol. 4, no. 3, pp. 297–309, 2005.

[22] Y. G. Choi, J. Kang and D. Nyang, "Proactive code verification protocol in wireless sensor network," in *Int. Conf. on Computational Science and Its Applications*, Berlin, Heidelberg, pp. 1085–1096, 2007.

[23] T. AbuHmed, N. Nyamaa and D. Nyang, "Software-based remote code attestation in wireless sensor network," in *GLOBECOM 2009–2009 IEEE Global Telecommunications Conf.*, Honolulu, HI, pp. 1–8, 2009.

[24] M. Shaneck, K. Mahadevan, V. Kher and Y. Kim, "Remote software-based attestation for wireless sensors," in *European Workshop on Security in Ad-hoc and Sensor Networks*, Berlin, Heidelberg, 27–41, 2005.

[25] S. Schulz, A. R. Sadeghi and C. Wachsmann, "Short paper: Lightweight remote attestation using physical functions," in *Proc. of the fourth ACM Conf. on Wireless Network Security, WiSec'11*, Hamburg, Germany, pp. 109–114, 2011.

[26] Y. Yang, X. Wang, S. Zhu and G. Cao, "Distributed software-based attestation for node compromise detection in sensor networks," in *2007 26th IEEE Int. Sym. on Reliable Distributed Systems (SRDS 2007)*, Beijing, pp. 219–230, 2007.

[27] S. Kiyomoto and Y. Miyake, "Lightweight attestation scheme for wireless sensor network," *International Journal of Security and Its Applications*, vol. 8, no. 2, pp. 25–40, 2014.

[28] F. Kohnhäuser, N. Büscher and S. Katzenbeisser, "A practical attestation protocol for autonomous embedded systems," in *2019 IEEE European Sym. on Security and Privacy (EuroS&P)*, Stockholm, Sweden, pp. 263–278, 2019.

[29] A. Visintin, F. Toffalini, M. Conti and J. Zhou, "SAFE^d: Self-attestation for networks of heterogeneous embedded devices. arXiv preprint arXiv: 1909.08168,2019.

[30] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin *et al.*, "New results for timing-based attestation," in *2012 IEEE Sym. on Security and Privacy*, San Francisco, CA, pp. 239–253, 2012.

[31] H. Tan, W. Hu and S. Jha, "A remote attestation protocol with Trusted Platform Modules (TPMs) in wireless sensor networks," *Security and Communication Networks*, vol. 8, no. 13, pp. 2171–2188, 2015.

[32] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter and H. Isozaki, "An execution infrastructure," in *Proc. of the 3rd ACM SIGOPS/EuroSys European Conf. on Computer Systems 2008 (Eurosys'08)*, New York, USA, pp. 315–328, 2008.

[33] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta *et al.*, "TrustVisor: Efficient TCB reduction and attestation," in *2010 IEEE Sym. on Security and Privacy*, Berkeley, Oakland, CA, pp. 143–158, 2010.

[34] F. Brasser, B. El Mahjoub, A. R. Sadeghi, C. Wachsmann and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Proc. of the 52nd Annual Design Automation Conf., DAC '15*, San Francisco, CA, USA, pp. 1–6, 2015.

[35] K. Eldefrawy, G. Tsudik, A. Francillon and D. Perito, "SMART: Secure and minimal architecture for (Establishing Dynamic) root of trust," in *Ndss*, pp. 1–15, 2012.

[36] M. Ambrosin, M. Conti, A. Ibrahim, G. Neven, A. R. Sadeghi *et al.*, "SANA: Secure and scalable aggregate network attestation," in *Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security*, New York, NY, USA, pp. 731–742, 2016.

[37] M. Ambrosin, M. Conti, R. Lazzeretti, M. M. Rabbani and S. Ranise, "Toward secure and efficient attestation for highly dynamic swarms: Poster," in *Proc. of the 10th ACM Conf. on Security and Privacy in Wireless and Mobile Networks*, New York, NY, USA, pp. 281–282, 2017.

[38] K. Song, D. Seo, H. Park, H. Lee and A. Perrig, "OMAP: One-way memory attestation protocol for smart meters," in *2011 IEEE Ninth Int. Sym. on Parallel and Distributed Processing with Applications Workshops*, Busan, pp. 111–118, 2011.

[39] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *28th USENIX Security Sym. (USENIX Security 19)*, Santa Clara, CA, USA, pp. 1429–1446, 2019.

[40] M. Ammar, B. Crispo and G. Tsudik, "SIMPLE: A remote attestation approach for resource-constrained IoT devices," in *2020 ACM/IEEE 11th Int. Conf. on Cyber-Physical Systems (ICCPS)*, Sydney, Australia, pp. 247–258, 2020.

[41] A. Seshadri, M. Luk, A. Perrig, L. Van Doorn and P. Khosla, "SCUBA: Secure code update by attestation in sensor networks," in *Proc. of the 5th ACM Workshop on Wireless security, WiSe'06*, Los Angeles, California, USA, pp. 85–94, 2006.

[42] A. Seshadri, M. Luk and A. Perrig, "SAKE: Software attestation for key establishment in sensor networks," *Ad Hoc Networks*, vol. 9, no. 6, pp. 1059–1067, 2011.

[43] D. Zhang and D. Liu, "DataGuard: Dynamic data attestation in wireless sensor networks," in *2010 IEEE/IFIP Int. Conf. on Dependable Systems & Networks (DSN)*, Chicago, IL, pp. 261–270, 2010.

[44] V. Haldar, D. Chandra and M. Franz, "Semantic remote attestation: A virtual machine directed approach to trusted computing," in *USENIX Virtual Machine Research and Technology Sym.*, 2004.

[45] S. K. Nair, "Remote policy enforcement using java virtual machine," Ph.D. thesis, VU University Amsterdam, 2010.

[46] C. Kil, E. C. Sezer, A. M. Azab, P. Ning and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence," in *2009 IEEE/IFIP Int. Conf. on Dependable Systems & Networks*, Lisbon, pp. 115–124, 2009.

[47] L. Davi, A. R. Sadeghi and M. Winandy, "Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks," in *Proc. of the 2009 ACM Workshop on Scalable Trusted Computing, STC '09*, Chicago, Illinois, USA, pp. 49–54, 2009.

[48] S. Adepu, F. Brasser, L. Garcia, M. Rodler, L. Davi *et al.*, "Control behavior integrity for distributed cyber-physical systems," in *2020 ACM/IEEE 11th Int. Conf. on Cyber-Physical Systems (ICCPS)*, Sydney, Australia, pp. 30–40, 2020.

[49] L. Cheng, K. Tian and D. Yao, "Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks," in *Proc. of the 33rd Annual Computer Security Applications Conf., ACSAC 2017*, Orlando, FL, USA, pp. 315–326, 2017.

[50] T. Abera, N. Asokan, L. Davi, J. E. Ekberg, T. Nyman *et al.*, "C-FLAT: Control-flow attestation for embedded systems software," in *Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security, CCS '16*, Vienna, Austria, pp. 743–754, 2016.

[51] N. Ahmed, M. A. Talib and Q. Nasir, "Program-flow attestation of IoT systems software," in *2018 15th Learning and Technology Conf. (L&T)*, Jeddah, KSA, pp. 67–73, 2018.

[52] B. Kuang, A. Fu, L. Zhou, W. Susilo and Y. Zhang, "DO-RA: Data-oriented runtime attestation for IoT devices," *Computers & Security*, vol. 97, pp. 101945, 2020.

[53] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A. R. Sadeghi *et al.*, "DIAT: Data integrity attestation for resilient collaboration of autonomous systems," in *Network and Distributed Systems Security (NDSS) Symposium 2019*, 2019, San Diego, CA, USA, 2019.

[54] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi *et al.*, "Lo-fat: Low-overhead control flow attestation in hardware," in *Proc. of the 54th Annual Design Automation Conf. (DAC)*, Austin, TX, pp. 1–6, 2017.

[55] R. De Clercq, R. De Keulenaer, B. Coppens, B. Yang, P. Maene *et al.*, "Software and Control Flow Integrity Architecture Design," in *2016 Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, Dresden, pp. 1172–1177, 2016.

[56] G. Dessouky, T. Abera, A. Ibrahim and A. R. Sadeghi, "Litehax: Lightweight hardware-assisted attestation of program execution," in *2018 IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, San Diego, CA, pp. 1–8, 2018.

[57] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim *et al.*, "Atrium: Runtime attestation resilient under memory attacks," in *2017 IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, Irvine, CA, USA, pp. 384–391, 2017.

[58] L. Szekeres, M. Payer, T. Wei and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Sym. on Security and Privacy*, Berkeley, CA, USA, pp. 48–62, 2013.

[59] K. Volodymyr, S. Laszlo, P. Mathias, C. George and R. Sekar, "Code-pointer integrity," in *Proc. of the 11th USENIX Sym. on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, 2014.

[60] N. S. Almakhdhub, A. A. Clements, S. Bagchi and M. Payer, "μRAI: Securing embedded systems with return address integrity," in *Network and Distributed Systems Security (NDSS) Sym.*, San Diego, CA, USA, 2020.

[61] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson *et al.*, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proc. of the 23rd USENIX Security Sym.*, San Diego, CA, USA, pp. 941–955, 2014.

[62] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *The Proc. of the 22nd USENIX Security Sym.*, Washington, D. C., USA, pp. 337–352, 2013.

[63] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris *et al.*, "Enforcing unique code target property for control-flow integrity," in *Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security, CCS '18*, Toronto, Canada, pp. 1470–1486, 2018.

[64] P. Larsen, A. Homescu, S. Brunthaler and M. Franz, "SoK: Automated software diversity," in *2014 IEEE Sym. on Security and Privacy*, San Jose, CA, USA, pp. 276–291, 2014.