EMMA: Hardware/Software Attestation Framework for Embedded Systems Using Electromagnetic Signals

Nader Sehatbakhsh*, Alireza Nazari, Haider Khan, Alenka Zajic, and Milos Prvulovic
Georgia Institute of Technology
*nader.sb@gatech.edu

ABSTRACT

Establishing trust for an execution environment is an important problem, and practical solutions for it rely on *attestation*, where an untrusted system (prover) computes a response to a challenge sent by the trusted system (verifier). The response computation typically involves calculating a checksum of the prover's program, which the verifier checks against expected values for a "clean" (trustworthy) system. The main challenge in attestation is that, in addition to checking the response, the verifier also needs to verify the integrity of the response computation itself, i.e., that response computation itself has not been tampered with to produce expected values without measuring the verifier's actual code and environment. On higher-end processors, this integrity is verified cryptographically, using *dedicated* trusted hardware. On embedded systems, however, constraints prevent the use of such hardware support. Instead, a popular approach is to use the *request-to-response* time as a way to establish confidence. However, the overall request-to-response time provides only one coarse-grained measurement from which the integrity of the attestation is to be inferred, and even that is noisy because it includes the network latency and/or variations due to micro-architectural events. Thus, the attestation is vulnerable to attacks where the adversary has tampered with response computation, but the resulting additional computation time is small relative to the overall request-to-response time.

In this paper, we make a key observation that the existing approach of execution-time measurement for attestation is only one example of using externally measurable side-channel information and that other side-channels, some of which can provide much finer-grain information about the computation, can be used. As a proof of concept, we propose EMMA, a novel method for attestation that leverages electromagnetic side-channel signals that are emanated by the system during response computation, to confirm that the device has, upon receiving the challenge, actually computed the response using the valid program code for that computation. This new approach requires physical proximity, but imposes no overhead to the system, and provides accurate monitoring *during* the attestation. We implement EMMA on a popular embedded system, Arduino UNO, and evaluate our system with a wide range of attacks on attestation integrity. Our results show that EMMA can successfully detect these attacks with high accuracy. We compare our method with the existing methods and show how EMMA outperforms them in terms of security guarantees, scalability, and robustness.

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358261

CCS CONCEPTS

• **Security and privacy → Embedded systems security**.

KEYWORDS

hardware security, trusted execution environment, embedded systems, side-channels, electromagnetic emanations.

ACM Reference Format:
Nader Sehatbakhsh*, Alireza Nazari, Haider Khan, Alenka Zajic, and Milos Prvulovic. 2019. EMMA: Hardware/Software Attestation Framework for Embedded Systems Using Electromagnetic Signals. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358261

1 INTRODUCTION

Establishing trust for an execution environment is an important problem and practical solutions for it rely on *attestation*, a security primitive that allows a trusted system (verifier) to verify the integrity of program code, execution environment, data values, etc. in an untrusted system (prover). Attestation typically relies on a *challenge-response* paradigm [38], where the prover is asked to calculate a checksum over verifier-requested parts of a program/data memory contents. The response computation typically involves measurement (e.g., checksum) of the prover's execution environment, which the verifier checks against expected values for a "clean" (trustworthy) system. The verifier considers the prover's integrity to not be compromised if *(i)* the checksum provided by the prover matched with the *expected* value computed by the verifier, *AND (ii)* the computation that produced the response itself has not been tampered with, e.g., to falsify the expected values without actually computing them from the verifier's actual code/data.

In high-end modern processors, the assurance that the response computation itself was not tampered with is typically provided by using a hardware-supported Trusted Execution Environment (TEE) [2, 15, 16, 23], which uses dedicated hardware (e.g., SGX, TPM, etc.) within the prover. In low-end processors and/or embedded systems, however, form factor, battery life, and other constraints prevent the use of hardware-supported enclaves or other hardware supports. Instead, a popular approach, *Software Attestation* [12, 19, 27, 35, 36, 38], is to compute the checksum in software, using ordinary execution on the prover, and to leverage measurement of the request-to-response time as a way to establish some level of confidence about the integrity of the response computation itself.

To implement this method, the verifier utilizes the challenge-response paradigm by asking the embedded system (prover) to compute a checksum of its program memory, while measuring the response time to prevent the adversary from computing a correct response, e.g., by temporarily restoring the program memory while the response is computed or by forwarding the challenge to another system that computes the response, etc. The prover passes the attestation test only if it provides the correct response to the challenge (i.e., $Response_{prover} = Response_{expected}$) without violating the timing requirement (i.e., $t_{response} < t_{threshold}$)[38].

Unfortunately, the overall request-to-response time provides only one coarse-grained measurement, and this method is not able to monitor the prover *during* the attestation without imposing a significant performance and cost overhead to the system. This, in turn, makes the software attestation schemes vulnerable to attacks which have very low-latency compared to the overall response time (i.e., $t_{attack} << t_{thr}$). Moreover, due to the network limitations and/or micro-architectural events, this request-to-response time may be noisy since it includes the round-trip network latency and/or variations caused by the micro-architectural events (e.g., cache miss) which consequently, causes a further increase in t_{thr} (to tolerate the variance and reduce the false positive rate), and hence, potentially makes these schemes even more vulnerable to low-latency attacks.

In this paper, we introduce EMMA, a new approach for attestation of embedded devices that leverages the side effects of the prover's hardware behavior, but without requiring any specific support from that hardware. Our scheme is based on this key insight that *the existing approach of execution-time measurement for attestation is only one example of using externally measurable side-channel information*, and that other side channels, some of which can provide much finer-grain information about the response computation, can be used. In particular, instead of the overall challenge-to-response timing, EMMA uses *electromagnetic (EM)* side-channel signals unintentionally emanated by the system during attestation. To create EMMA, we first study the possible attack scenarios on attestation methods, and then design EMMA such that it fully addresses these vulnerabilities. Also, to increase the accuracy (and reduce false-positives), we first investigate the different sources of variations (e.g., micro-architectural events), and then carefully design EMMA such that it effectively minimizes these variations while tolerating uncontrollable sources of variations (e.g., environmental). Using an extensive set of measurements, we will show EMMA's ability to achieve high accuracy under different attack scenarios while being robust against different sources of environmental variations.

We will show how EMMA can be implemented using an inexpensive setup to monitor an embedded device, and how it can be scaled to monitor multiple devices with very low cost overhead per device (<$10). We envision that EMMA can be used to attest a group of embedded systems that are mostly dedicated to a specific task. This includes but not limited to a network of sensors or peripherals that are connected to a main controlling unit, cyber-physical systems in hospitals and/or factories, industrial IoT (IIoT) systems, etc. In these scenarios, the cost (per device) and complexity of deploying EMMA is relatively low because it requires no changes to the monitored device, and thus creates no regulatory, safety, or disruption concerns for the system. More importantly, it has zero-overhead on the monitored system while physically separated from the monitored device. In a practical scenario, EMMA can leverage an already existing infrastructure for controlling the CPS such as industrial control systems (ICS), SCADA, etc. which further simplifies its implementation and reduces the costs. Furthermore, we envision EMMA is being useful in other scenarios such as checking the integrity of legacy systems which are notoriously difficult to manage and verify, providing a secure execution environment on sensor nodes and/or IoT devices for secure code update, error recovery, key exchange, etc. Finally, EMMA can be used as a *portable* setup to occasionally monitor one or a small group of devices. In this scenario, EMMA can be used as a low-cost, powerful tool to debug under-the-test systems.

Followings are the main contributions of this paper:

- A new attestation method based on electromagnetic emanations of the prover,
- A proof-of-concept implementation of this attestation method, which we call EMMA,
- Evaluation of EMMA on four different types of attacks and its comparison with the state-of-the-art,
- Further analysis on EMMA for its applicability on other platforms and its robustness against variations.

The rest of this paper is organized as follows. § 2 provides a brief background on EM side-channel signals and software attestation. It also overviews the threat model, possible attack models, and our assumptions and considerations in designing EMMA. § 3 describes EMMA in details. § 4 presents our evaluations including results for four different attacks. In § 5, we evaluate the robustness of EMMA. § 6 discusses the related work. Finally, § 7 concludes this paper.

2 BACKGROUND, ATTACKER MODELS, AND ASSUMPTIONS

Software Attestation. The main goal of attestation is to dynamically establish a *dynamic root of trust* (DRT) [7, 10, 27] on an untrusted platform. After successful attestation, the code and data within this DRT is assumed to be unmodified, and this can be leveraged to measure the integrity of other parts of the untrusted system (e.g., checking the integrity of an arbitrary piece of code by computing the hash using the code that is within the dynamic root of trust), and/or initiate execution of other code in the system without concerns about tampering with their initial execution environment.

In software attestation, the DRT is instantiated through the *verification function*, a self-checking function that computes a checksum over its own instructions and sends it to the verifier. To establish trust, this checksum has to match with the expected value, and other measurable properties of the checksum calculation itself must pass certain tests. The function typically consists of *(i)* an initialization phase, *(ii)* a main computation loop, and *(iii)* an epilogue.

In the existing frameworks, the measured property is the request-to-response time, which is assumed to correspond to the execution time of the checksum computation, and the test consists of checking if the response time was fast enough. In this work, the measured property is the prover's EM emanations during the checksum computation, and the test consists of verifying that signal against a model of emanations for a legitimate checksum computation.

Attack Models. To attack an attestation framework, the attacker has two options: *(i)* to forge the checksum value using classic checksum collision attacks [41, 42]. This attack, however, can be easily defeated using a sufficiently long checksum [36]. *(ii)* more realistically, to modify the checksum code to compute the correct checksum, or to hide the malware without violating the requirements (e.g., $t_r < t_{th}$). To launch such an attack the adversary has two options: *(1)* she can modify the checksum calculation main loop to calculate the checksum on another region of the memory which stores the unmodified copy of the code. This attack is called *memory copy* [36–38] attack. *(2)* she can modify the prologue/epilogue phases by *(a)* forwarding the challenge to another device which called *proxy* attack [27], or *(b)* by removing the malicious code before calculating the checksum and hiding it in other parts of the memory which called *rootkit* attack [11].

The main challenge for the adversary is that while changing the checksum calculation is more effective and feasible, any change (even single instruction) in the checksum computation phase will be significantly magnified due to a large number of iterations of the loop. Thus, the adversary will be faced with a fundamental choice between having either a large but short-term malicious activity in the epilogue/prologue, or alternatively, a small, but long-term malicious activity in the checksum phase. Hence, an ideal detection framework should be able to detect single-instruction modifications in the checksum loop, and tiny changes in the epilogue/prologue phases. In the next section we will present how EMMA is designed to satisfy these requirements.

Electromagnetic Side-Channel Signals. In practice, every electronic device generates unintended electromagnetic (EM) signals, as changes in current flows within the device are converted (according to Faraday's law) into EM signals that emanate from the device. In a processor, program activity governs most of the electronic activities, so its EM signal is highly related to program execution.

There have been several proposals for analyzing EM signals, mainly in time-domain, using different techniques such as machine learning, signal processing, etc. [8, 17, 28, 30]. Alternatively, recent methods [20, 31, 34] proposed a different approach by analyzing the EM signals in the frequency-domain. These methods are mainly based on this key observation that spectrum of the EM signal emanated during the execution of periodic activities (e.g., loops) have a strong peak at the frequency of the processor's clock (f_{clk}), and also spikes at frequencies $f_{clk} \pm f_l, f_{clk} \pm 2f_l, etc.$, where f_l corresponds to the per-iteration time, T, of the current loop ($f_l = 1/T$). These additional spikes are a result of the clock-frequency EM emanations being amplitude-modulated by the periodic program activity. The main advantage of this approach over time-domain methods is that instead of analyzing a fairly noisy signal in the time-domain, the "average" behavior of the signal can be analyzed accurately in frequency-domain which, in turn, significantly improves the signal-to-noise ratio and simplifies the analysis. Analyzing in frequency-domain, however, comes with a consequential loss in temporal resolution since, fundamentally, to achieve a "good" frequency resolution, a wider time *window* should be used which, in turn, reduces the temporal resolution (this problem is also known as Gabor-Heisenberg uncertainty principle). This is particularly important in side-channel signals for cases where a short-term change (e.g., a malicious activity to hijack the control-flow, or a short-term unauthorized activity) should be detected by analyzing the side-channel (EM) signal.

Given these challenges, to build an effective framework for utilizing EM signals for attestation, two requirements should be met: *(i)* to protect the system against short-term attacks, i.e., attacks on prologue/epilogue phases (e.g., *proxy attacks* [27]), the detection framework should be able to analyze the signal with fine time-resolution (i.e., a time-domain analysis), and *(ii)* to detect small changes during the main checksum computation's loop, the detection framework should be able to detect small changes in the per-iteration time of the loop (i.e., a frequency-domain analysis). Unfortunately, existing side-channel analysis frameworks [8, 20, 31] are unable to satisfy both conditions.

To achieve that, in this paper we develop an EM-Monitoring algorithm that can *(a)* analyze the signal in time-domain to detect even small changes before/after the main checksum computation loop begins/ends, and *(b)* check whether the attestation process (during the checksum computation) matches with a known-good model (to ensure that this process is not modified by an adversary) by using a frequency-domain analysis.

Threat Model and Assumptions. We assume that the adversary has installed malicious code on the target embedded system, with full control over the hardware and software of the device, including the ability to arbitrarily modify program and data memory, or any other memories available on the device. The attack succeeds if the device passes the attestation despite the presence of a malicious code. Note that attestation does not depend on how malicious code was originally installed on the device, and methods for doing so are abundantly represented in the research literature, although we do not discuss them in detail in this paper.

Unlike most prior software-based methods, we assume that the prover (attested device) *can send messages* to and collude with other *faster* malicious peers, e.g., to use them as proxies for calculating the checksum faster. Also, *the attacker can modify the clock speed of the embedded system*. Finally, we assume that the verifier has full information about the prover's architecture (e.g., address space, memory architecture, etc.).

3 EM-MONITORING ATTESTATION (EMMA)

Overview. At the high-level, EMMA is designed such that it is able to extract the exact *begin* and *end* time of the checksum calculation, along with the accurate *per-iteration* execution time of the main checksum loop. An overview of the EMMA framework is shown in Figure 1. This framework consists of a Verifier, $\mathcal{V}$, (e.g., a trusted PC) and a Prover, $\mathcal{P}$, (e.g., an embedded system). The Verifier includes, or is connected to, a monitoring system (EM-Mon) that can receive and analyze the EM signals unintentionally emanated by $\mathcal{P}$. Attestation begins with $\mathcal{V}$ preparing a challenge locally (❶). The challenge includes a seed value (which will be used later to initialize a Pseudo-Random Number Generator (PRNG) in $\mathcal{P}$), an address range, the total number of iterations for checksum loop, a random value to initialize the checksum in $\mathcal{P}$, and a random *nonce*. The challenge is then sent to the $\mathcal{P}$ via a communication link (❷)

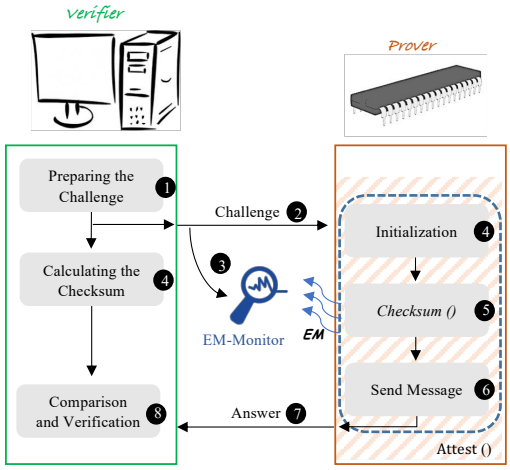

Figure 1: Overview of EMMA framework.

which invokes the *verification function*, `attest()` by causing an interrupt on $\mathcal{P}$. This function runs at the highest prover's processor privilege level with interrupts turned off.

Upon sending the challenge to $\mathcal{P}$, $\mathcal{V}$ also starts the "monitoring process" (❸) on EM-Mon. Through analysis of EM signals emanated from $\mathcal{P}$, EM-Mon determines three critical values about $\mathcal{P}$'s checksum computation and reports an error if any of them deviates from a known-good model (*reference* model). These tasks/values include *(i)* the delay between reception of the challenge and start of the checksum loop, and the signal signature during this initialization phase. *(ii)* the checksum loop's EM signature (i.e., frequency of spikes). *(iii)* the total attestation time on $\mathcal{V}$.

These values/tasks are primarily chosen because, fundamentally, there are two critical durations in the execution of the attestation process on the prover: *(a)* the time that is taken between receipt of the command for performing attestation and the start of the actual checksum process (because an adversary could contact another device during this period, or quickly hide the malicious code before starting the procedure), and *(b)* the time taken for the checksum process itself (because an adversary could try to do "extra work" during checksumming to hide the malicious code).

After sending the challenge, the verifier, independently, calculates the "expected" checksum on its own (known-good) copy of the embedded system's program memory (❹). At the same time, the self-checking verification function starts with initializing its local variables based on the received challenge (❹), and then it starts the "checksum calculation", `Checksum()` (❺). This function is an optimized loop which in each iteration, it reads a memory line in a pseudo-random fashion, and updates the checksum based on the content of that address. The address range and the total number of iterations of the loop are determined by the challenge.

Once `Checksum()` is finished, $\mathcal{P}$ forms a response (❻) that includes the calculated checksum and the random nonce which initially was sent by $\mathcal{V}$, and sends this response to $\mathcal{V}$ (❼). The original nonce acts as an identification and helps to increase the overhead for a proxy attack (see § 4). Finally, $\mathcal{V}$ compares the information received from $\mathcal{P}$ with its pre-computed checksum and the original challenge and compares the results from EM-Mon to the expected ones. If they all match, one trial of attestation protocol will finish successfully (❽). At this point the *dynamic root of trust* has been established; thus, the verification function can either invoke an executable, and hence, provides a TEE, or invoke a *hash* computation function to compute the hash value over the prover's memory contents (entirely or partially). This hash value can then be sent back to the verifier, which in turn, provides the current state of the prover to the verifier. Note that all of these functionalities are still part of the verification function; thus, they have been used in computing the checksum, which means it is guaranteed that the hash function or invoking the executable is also untampered with.

Verification Function (`attest()`). This function has three parts: a *prologue* or initialization phase, *checksum computation*, and an *epilogue* which is responsible for sending the response back to the verifier and invoking the executable or hash computation function.

To defeat possible attacks against attestation, this function should be carefully designed to have low runtime variation. Knowing this fact, from the computer-architecture perspective, `attest()` has to be designed such that *(a)* the prologue and epilogue phases are fairly deterministic and sufficiently long such that a simple time-domain analysis can be used to find a potential deviation (due to malicious activities caused by the attacker). *(b)* the computation loop should have minimum per-iteration variation and non-parallelizable.

In addition to considering these runtime requirements, the checksum computation function should be secured against "static" attack scenarios namely *pre-computation* and *replay* attacks [36, 37], where the attacker leverages previous challenge-response pairs to calculate the new response. To avoid these attacks, the checksum computation should be a function of the challenge (to prevent replay attacks), and the memory address generation in each iteration should be done in a pseudo-random fashion (to prevent pre-computation attack).

Using these two criteria (i.e., runtime and security requirements), we design our attestation algorithm (mainly based on prior work [27, 35–38]) to satisfy these conditions. Algorithm 1 shows the checksum's pseudo-code. We will first describe the algorithm and then provide a brief formal security analysis of the code to prove that the checksum computation is also cryptographically secure.

Checksum Algorithm. The main checksum loop consists of a series of alternating *XOR* and *ADD* instructions. This series has the property of being *strongly-ordered* [37]. A strongly-ordered function is a function whose output differs with high probability if the operations are evaluated in a different order. Using a strongly-ordered function requires an adversary to perform the same operations on the same data in the same sequence as the original function to obtain the correct result; thus none of the operations can be re-ordered or removed. Furthermore, using this sequence prevents parallelization and out-of-order execution since, at any step, the current value is needed to compute the succeeding values.

We use a 160-bit long checksum to keep all the registers busy and to significantly reduce the checksum collision probability [36]. The checksum is stored as a vector in a set of 8/16-bit general purpose registers (blocks) depending on the architecture of the processor (i.e., AVR, ARM, etc.). To traverse the memory in a pseudo-random fashion, we use a PRNG. Similar to previous work, we use a 16-bit

***Algorithm* 1** The checksum computation algorithm used in EMMA.

```
1:  Initialization:
2:  RNum = seed
3:  Set MASK based on beginAddress and endAddress
4:  Offset = beginAddr
5:  cSum = seed
6:  Checksum:        // checksum main loop (Checksum())
7:  for i=1 to totIter do
8:      for j=1 to 10 do
9:          RNum = RNum + (RNum^2 ∨ 5) mod 2^16
10:         memAddr = memAddr ⊕ RNum
11:         memAddr = (memAddr ∧ MASK) + Offset
12:         cSum_j = cSum_j + (Mem[memAddr] ⊕ cSum_{j-1})
13:         cSum_j = cSum_j + (i ⊕ PC)
14:         cSum_j = cSum_j + (RNum ⊕ memAddr)
15:         cSum_j = cSum_j + (SR ⊕ cSum_{j-2})
16:     end for
17: end for
```

T-function [22] to generate these random numbers. Each partial checksum block is also dependent on *(a)* the last two calculated partial sums; to avoid parallelization and pre-computation attack, *(b)* a key; to avoid replay attack, *(c)* current memory address (data pointer) and PC (if available depending on the architecture); to avoid memory copy attack, *(d)* the content of the program memory; to avoid changing the attestation code, and *(e)* the Status Register (SR) to check the status of the interrupt-disable flag. In § 4, we will show different attack scenarios and discuss why all these properties are needed to prevent different attacks.

To avoid variations due to possible branch mis-predictions, in the actual implementation of the checksum, the inner loop is unrolled to calculate all the partial sums in one iteration. To avoid variations due to cache misses, the *MASK* is generated such that the data access address range fit into an L1 cache (if any). Note that *MASK* is a function of the received challenge (see line 3) thus it is the verifier (trusted user) responsibility to generate correct challenge to set the *MASK* properly. Also, to cover the full address space, the verifier can send multiple challenges with different address ranges. Further, the attestation code itself is compactly designed so that it fits into an instruction cache. The detailed implementation of this code on an Arduino Uno will be shown in § 4.

Security Analysis. Based on the framework proposed in [5], in general, to analyze the security of any software attestation framework, two core components should be analyzed: memory address generator (*Gen*) and the checksum computation/compression function (*ChK*).

To be cryptographically secure, the addresses, a_i, generated by *Gen* should be "sufficiently random" [5]. To achieve that, a_i should be *computationally indistinguishable* from uniformly random values within a certain time-bound, t_{min}, (assuming that $\widetilde{\mathcal{P}}$ does not know the seed in advance). In practice, $\widetilde{\mathcal{P}}$ can use an arbitrary seed value to compute all the possible addresses on its own, making them easily distinguishable from random values. However, it can be shown that to maintain the security [5], it is only required that $\widetilde{\mathcal{P}}$ cannot derive any meaningful information about a_{i+1} from a_i and the seed without investing a certain minimum amount of time, $t_{compute} \geq t_{Gen}$. Specifically, we assume that an algorithm with input s that does not execute *Gen* cannot distinguish $a_{i+1} = Gen(a_i)$ from uniformly random values. This property holds true for the T-functions as shown in [22], since either the adversary needs to spend the same amount of time as *Gen* to compute the next address or, alternatively, pre-record all possible (addr, nextAddr) pairs. For a 16-bit T-function, saving all the pairs requires more than 128KB memory, which means to access this data in run-time, the attacker needs to access either L2 or the main memory, thus $t_{compute} = t_{mem}$. In our design (line 9 in Algorithm 1) t_{Gen} is only a few cycles (<5) which is clearly much less than $t_{mem} > 20$ in typical low-end processors.

The purpose of the checksum function, *ChK*, is to map the memory state of the prover, $\mathcal{P}$, to a smaller attestation response, r, which reduces the amount of data to be sent from $\mathcal{P}$ to the verifier $\mathcal{V}$. A mandatory security requirement on *Chk* is that it should be *hard* for $\widetilde{\mathcal{P}}$ to replace the correct input, S, to *Chk* with some other value, $S' \neq S$, that yields the same attestation response r (i.e., *second pre-image resistance* of cryptographic hash functions). However, unlike the hash functions where the adversary may know the correct output (and searches for the second output), in the software attestation schemes the adversary does not even know the correct (first) response. The reason is that, as soon as $\widetilde{\mathcal{P}}$ knows the correct response, he could send it to $\mathcal{V}$, and would not bother to determine a second pre-image. As a resut, this leads to a much *weaker* second pre-image resistance requirement for attestation frameworks.

Using this fact, our checksum is designed such that it significantly reduces the chance of the collision while being computationally hard for a second pre-image attack. This can be proven, as shown in [26], that *ChK* used in this paper provides an almost full coverage (i.e., almost all possible numbers in the $[0, 2^{16} - 1]$ range for a 16-bit partial checksum), which, in turn, makes *ChK* resistant to (blind) pre-image attacks. In fact, using the framework in [5], the probability of a checksum collision in our framework (for 160-bit checksum and $totIter = 100$) is $< 10^{-40}$.

EM-Monitoring. The EM-Mon component ensures that the attestation computation in $\mathcal{V}$ is not tampered with. Figure 2 shows this monitoring framework. Using an antenna (e.g., a magnetic probe) and a signal acquisition device (e.g., a software-defined-radio), the EM signal is captured and received as a time-series (❶). As mentioned in § 2, depending on the required analysis, either *time-domain* or *frequency-domain* analysis is selected (❷) where for analyzing the prologue and epilogue, time-domain is used, and for the computation loop, frequency-domain analysis is used (the decision is made by the current state of the FSM which will be discussed in the following).

For frequency-domain analysis, the signal is then transformed into a sequence of Short *Frequency-Domain* Samples (SFDS) using a Short-Time Fourier Transform (STFT). This transformation consists of dividing the EM signals into consecutive, equal-length, and overlapping segments of size t and then computing the STFT from each of these segments to obtain the corresponding SFDS (❷ and ❸-bottom). Segment size, t, has to be chosen such that it provides a balance between the time resolution and EM-Mon's computational

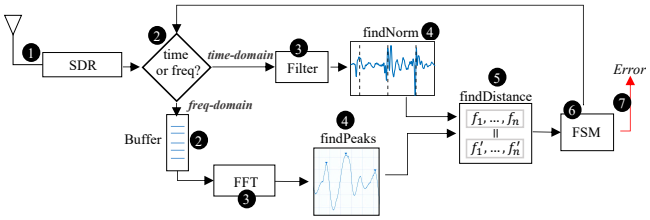

Figure 2: Overview of EM monitoring framework.

needs. Also, t should be long enough to capture several iterations of the checksum loop to model the *average* behavior of the loop. Each block in the main checksum loop on Arduino Uno takes about 20 machine cycles, and calculating the entire 160-bit checksum takes $\approx$ 400 cycles (about 25μs). Therefore, in this paper, we use 1ms segment size with 80% overlap (i.e., each segment equals to 40 iterations), and consecutive segments differ only in 8 iterations.

While these numbers provide adequate time-resolution for the checksum computation loop, it is not sufficient to detect small changes before/after the main loop. Thus, time-domain analysis is used in these regions where the signal is first low-pass filtered and normalized using mean subtraction and scale normalization (❸-top), and then segmented into consecutive, equal-length, and overlapping segments of size t (❹-top). We used $t = 24$ samples in our setup. Consecutive segments differ only in one sample (we used an SDR with 2.4 MHz sampling-rate), to maximize the time-resolution (the time-resolution in this setup is about 6 cycles vs. 3200 cycles for the frequency-domain analysis).

For frequency-domain analysis, each SFDS then goes into the `findPeaks()` module (❹-bottom) where n spikes are selected and later used as "signatures" for each SFDS. In `findPeaks()`, the first step in finding a spike is that for each frequency, f, that is of interest in an SFDS, we first compute the corresponding normalized frequency as $f_{norm} = (f - f_{clk})/f_{clk}$, where f_{clk} is the clock frequency for that SFDS. This normalized frequency is expressed as an offset from the clock frequency so that a shift in clock frequency does not change f_{norm} with it and is normalized to the clock, so it accounts for the clock frequency's first-order effect on execution time. We call this technique *"clock-adjustment"*. The criteria for selecting spikes are choosing the n largest amplitude local maxima (peaks), excluding the spike for the clock, that are not part of the *noise*. To find the noise, we record the spectrum once without executing the attestation function and save all the spikes that are 3-dB above the noise floor as *noise*. For our evaluations, we select $n = 7$, to capture the checksum loop's fundamental frequency and its second and third harmonics (in both sidebands). The frequency of the clock is also reported to prevent attack scenarios where the clock speed is increased to hide malicious activities.

To check the correctness of the execution, the time-domain segments generated by `findNorm()` or the frequency peaks (a vector of size n) generated by `findPeaks()` should be compared to a known "reference model" that is achieved during the secure execution of `attest()` (❺). Note that the reference model needs to be created only once. For time-domain signals, the reference model is a *dictionary* of segments each of which with size of $t = 24$ (note that the entire initialization phase takes about 500μs, i.e., the dictionary size is about 5000 elements). For the frequency-domain, the model is a vector of size n which stores the frequency bins of the computation loop's spikes. We assume that either the manufacturer or the end-user is able to achieve a correct reference model.

We use Pearson-correlation for frequency-domain, and cross-correlation for time-domain as the distance metric. In time-domain, the entire signal can be reconstructed by concatenating the best-match segments in the dictionary. The reconstructed signal is then compared to the original signal (i.e., the signal obtained during the actual execution of the initial phase) using mean-square-error (MSE) method. Finally, to decide whether the signal "matches" with the correct execution, a simple moving average (SMA) filter is used. The SMA is mainly used to remove short-term runtime noises (i.e., when only a few samples deviate from the model due to environmental/measurement noise). If the outcome of the SMA always stays below a a threshold ($th_t = 0.1$), the signals are *matched*. In frequency-domain analysis, the signal is matched if the correlation coefficient is larger than $th = 0.8$. Based on these distance comparisons, `findDistance()` outputs two boolean values (`isMatched_t`, `isMatched_f`) showing whether the signal matches with either the initialization or the computation phase or none of them.

The final stage of EM-Mon is a Finite-State Machine (FSM ❻). The default state for the FSM is when EM-Mon is waiting for the attestation to start ($state = 0$). Upon receiving a challenge from $\mathcal{V}$, FSM switches to $state = 1$, starts a timer called `challengeTimer`, and starts the time-domain analysis. During this phase, FSM throws an error if `findDistance()` reports "not-matched" (for time-domain analysis). FSM switches to $state = 2$ once `findDistance()` reports "match" for the frequency-domain analysis (i.e., this is when the computation loop begins). Also, the FSM throws an error if `challengeTimer` is larger than a threshold. Checking this value ensures that the system can be protected against *proxy*, *code compression*, and *return-oriented rootkit* attacks, where the attacker needs to spend some (non-negligible) time to set up the attack before actually starting the `Checksum()`.

Note that this is an important and unique feature of `EMMA` since existing methods are all unable to measure this delay (*even when they are directly connected to the verifier by a cable*), and can only measure the time between sending the challenge and receiving the checksum value.

The output of `findDistance()` becomes zero (for the frequency-domain analysis) when the checksum loop completes, so the FSM switches to $state = 3$, and checks the `challengeTimer` once again. This check ensures that the total execution time of attestation does not exceed a threshold which is defined by $initTime + perIteration \times totIter$, where $perIteration$ is the checksum loop per-iteration time, $totIter$ is the total number of iterations for calculating the checksum, and $initTime$ is a constant.

Lastly, in $state = 3$, EM-Mon starts a timer called `checksumTimer` and waits for an acknowledge from $\mathcal{V}$ that the checksum is received. At this point, if `checksumTimer` is larger than a constant, FSM again throws an error. Otherwise, it successfully switches back to $state = 0$ and waits for the new attestation challenge. This check ensures that the adversary can not spend any extra time after the checksum calculation is finished and before actually sending the checksum to $\mathcal{V}$. Note that in all cases, FSM can only transit from state n to $n + 1$ to enforce the correct ordering in attestation.

4 EXPERIMENTAL EVALUATION

Measurement Setup. To evaluate the effectiveness of our method, we used a popular embedded system, Arduino Uno, with an ATMEGA328p microprocessor clocked at 16MHz. To receive EM signals, the tip of a small magnetic probe [1] was placed about 10 cm above the Arduino's microprocessor (with no amplifier). To record the signal, we used an inexpensive compact software-defined radio (RTLSDRv3 [33]) cost about 30$. We recorded the signals at 16 MHz with a 2.4 MHz sampling rate. Note that all of our measurements were collected in the presence of the other sources of EM interference including an active LCD that was intentionally placed about 15 cm behind the board. A set of TCL scripts were used to control the attestation process. The real-time EM-Monitoring algorithm was implemented in MATLAB2017b.

Implementation. Arduino Uno uses an ATMEGA328p microprocessor, an Atmel 8-bit AVR RISC-based architecture, with a 16KB Program memory and a separate Data memory (unlike most other architectures where a single memory is used for both data and for executable instructions). This micro-controller has 32 8-bit general purpose registers where the last 6 registers can be combined in groups of two, and form three 16-bit registers (namely X, Y, and Z). The Z register can be used to access/read the program memory using *LPM Z* assembly instruction. Note that, unlike most of the micro-controller architectures, AVR does not provide direct access to the Program Counter (PC) register, so the value of the PC cannot be used during checksum calculation.

Checksum is saved as a vector in 20 8-bit registers ($r0 - r19$). Z register ($r31 : r30$) is used for reading the program memory (*memAddr*), Y register is used to store the random number generated by PRNG. Inputs from the challenge are pushed to the stack prior to invoking `attest()`, and later are read in the *initialization* phase. $r25 : r24$ are used to save the *MASK* value. $r23 : 21$ is used to save the *nonce*, and $r20$ is used to store the content of the memory. Finally, X is used for saving the current index (i). In our framework, each partial checksum calculation ($cSUM$) takes 20 cycles. Hence, adding an extra one-cycle instruction (e.g., an *ADD*) to the partial checksum block should increase the per-iteration time (and the corresponding spike in the frequency domain) by about 5%. The initialization phase takes about 500μs to receive the challenge via a Serial communication protocol.

Attacks. We evaluate the security of EMMA by implementing different attacks on a software-based attestation framework and showing that EMMA can indeed detect these attacks, and protect the system against them. These attacks can be divided into ***(i)*** attacks to the main checksum loop (shown as L-1 and L-2), and ***(ii)*** attacks to the epilogue/prologue phase (EP-1 and EP-2) as described in § 2.

L-1: Memory-Copy Attack: The most straightforward attack against software-based attestation methods is the *memory-copy attack*, where the adversary has created a copy of the original code *elsewhere* in memory, and the checksum code is modified to use that range of addresses instead of the original ones. Since the challenge sent by $\mathcal{V}$ could request to read any memory line in the program memory address space, potentially including the supposedly "empty" memory space where the "clean" copy of the original code is kept, to avoid detection this modified code must check addresses that are used during checksum computation, and then perform accesses without modification for unmodified memory ranges, redirect them to "clean" copies for modified memory ranges or use override values for supposedly empty ranges that now actually contain the attacker's data (including "clean" copes of original values from program and data memory).

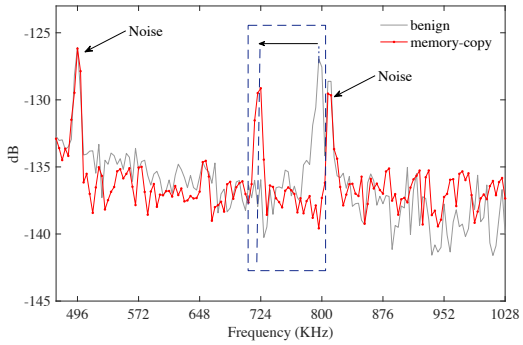

Figure 3: EM spectrum during checksum computation for the original code (gray) and Memory-Copy attack code (red). The x-axis is the frequency offset relative to the processor's clock frequency.

This checking and redirection of memory requests introduce overheads during checksum computation. Specifically, the adversary needs to change *memAddr* register (register Z in our implementation) to point to another address in the memory (at least one added instruction). Moreover, since we are using *memAddr* in the checksum calculation, the value itself has to be changed back to the correct value (another instruction). Note that, in our implementation, since accessing the program memory is only possible through Z, the adversary's only option is changing Z. Even for program/-data location that are unchanged by the adversary, the checksum code must suffer overheads of checking (a compare and a branch instruction) that the address falls in a range that still contains original instructions/data. Overall, to implement this attack, the adversary has to add at least two instructions per check-summed location.

One countermeasure to defeat this attack is to fill the unused memory regions with random values only known to the verifier [43]. Castelluccia *et al.* [11], however, showed that this defense can be circumvented using a simple *compression* attack where the unused parts of memory are compressed and stored in non-executable regions. Hence, to provide a stronger security guarantee, in this paper we relax the assumption that "free" space is filled with random values and allow all "free" memory locations to be filled with the same value (e.g., 0xFF). This allows the attacker to store malicious code in an empty region of program memory, and to modify checksum computation so that *LPM Rd*, Z (i.e., load from program memory) for that region of the memory is replaced with *SER Rd* (set *Rd* to 0xFF). In our experiments, an *LPM* instruction uses 2 more cycles than *SER* does, thus removing 2 cycles from the 4-cycle performance penalty introduced by the *compare-and-branch* check, that is still needed to determine which region of program memory is being

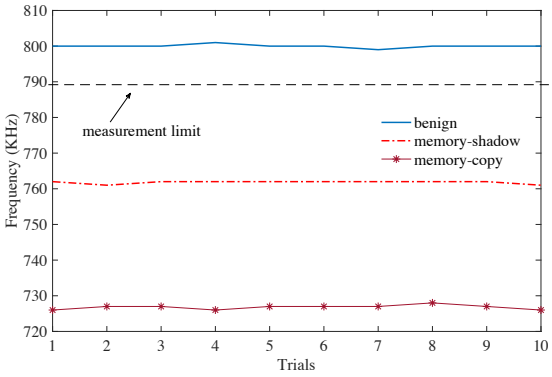

Figure 4: Frequency of the checksum computation loop for (a) attack-free, (b) memory-shadow attack, and (c) memory copy attack code.

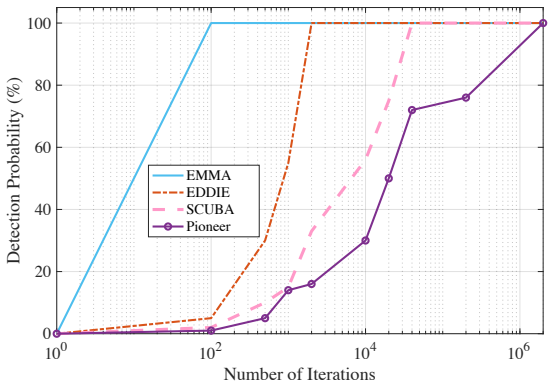

Figure 5: The probability of detecting memory-shadow attack for EMMA and prior work.

addressed. The reduced additional latency makes the modification of the checksum computation more difficult to detect.

To evaluate our framework, we implemented the Memory-Copy attack, and we trained EMMA on (only one) attack-free instance of attestation. We then applied EMMA to both attack-afflicted and attack-free instances of the attestation. The spectra of the resulting signals (Figure 3) show the spikes that correspond to the original checksum computation loop, and also the spikes that correspond to the modified checksum computation (red), which are shifted closer to the processor clock's frequency because the per-iteration time of the loop has increased. Figure 4 shows the spike's frequency for 20 attestation instances, 10 attack-afflicted, and 10 attack-free, showing a consistent difference among them. We find that EMMA successfully labels all these instances, i.e., all attack-afflicted instances are labeled as attack-afflicted (successful detection), and all attack-free are labeled as attack-free (no false positives). The "measurement-limit" line refers to the threshold that was used in EMMA for the labeling decision.

L-2: Memory-Shadow Attack: Castelluccia et al. [11] showed that instead of adding a *branch and compare* instructions to compare the address, the adversary can reduce the number of added cycles by one (i.e., 2 instead of 3) by copying the malicious code into a specific "free" area of the program memory (assuming it is not filled with random values) and called it Memory-Shadow Attack.

This attack leverages the fact that most of the program memory available on an embedded system is usually empty. In this case, if the second half of the program memory (addresses with $MSB = 1$) is empty, all accesses to the last quarter of the memory (addresses $11x...x$) could be redirected to the third quarter (addresses $10xx...x$) by just flipping one bit in the address. The last quarter of the program memory would then be shadowed and could be used to hide the malicious code. To implement this attack, the adversary simply needs to check the highest bit, and if it is one, sets the second highest bit to zero. This can be implemented in an AVR ISA with two instructions "*sbrs r31, 7*" and "*cbr r31, 6*" which takes two cycles; thus, the overall number of added cycles will be three which is only one cycle more than the actual checksum loop (recall that the adversary does not need to read from the memory so that gives him an extra two cycles).

To further show that adding even an extra cycle to the main loop would shift the frequency of the loop, Figure 4 shows the frequency shift for adding a simple one cycle ALU instruction where we added a *CBR* (clearing a single bit in a register) instruction to the main loop. This confirms that the adversary *cannot add any instructions* to the main loop of the code since any instruction in AVR ISA adds at least one cycle of overhead hence can be detected by EMMA, a very strong security guarantee. Furthermore, similar to memory-copy attack, we used 10 trials for this attack and showed that its frequency is well below the actual frequency of the main loop (see Figure 4). Similar to memory-copy, here we also got 0% false positive and 100% true-positive rate accuracy.

To further compare EMMA with prior work, Figure 5 shows the detection probability for Memory-Shadow attack as a function of number of checksum loop iterations. Intuitively, higher number of iterations magnifies the overhead of adding extra instructions (cycles) to the loop thus it gets easier to detect. However, to limit runtime variations, number of iterations is limited by the size of L1 and/or instruction cache. Also, more iterations requires longer computation time which, in turn, increases the overhead (power, device availability, etc.). Thus, ideally the detection framework should be able to detect attacks with small number of iterations to minimize these overheads and the increase accuracy. As shown in Figure 5, EMMA detects attacks that involve as few as 100 iterations, which is 20x smaller than what can be detected by EDDIE [31], and more than two orders of magnitude smaller than the fastest timing-based approach. The main reason for this dramatic improvement in sensitivity, compared to EDDIE, is that EMMA leverages Pearson correlation as a distance metric instead of using non-parametric tests used in EDDIE [31]. Compared to the timing-based methods, EMMA can provide fine-grain per-iteration monitoring which enables it to detect small changes. Note that given the attacker knows that EMMA requires about 100 iterations to detect the change, the attacker can design a new attack to *selectively* run the memory-shadow attack in some iterations. While the attacker can modify the code in that way, achieving that functionality itself needs adding a *branch-and-compare* instruction (i.e., to check the iteration number), which, in turn, causes an overhead and hence will be detected by EMMA.

EP-1: Rootkit-Based Attack: Another class of attacks leaves the original checksum loop unchanged, but adds work before and/or

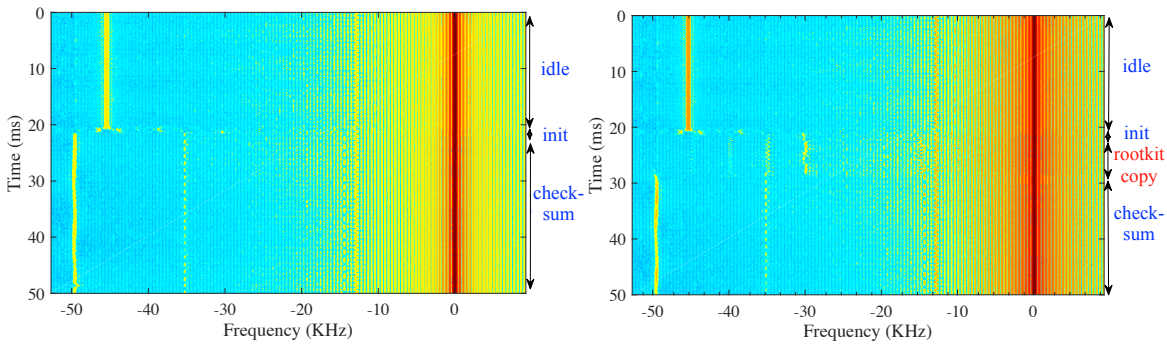

Figure 6: Spectrogram of the attestation code in normal (left) and rootkit-attack (right) runs. Note that the slight differences in colors between the two spectrograms correspond to variations in signal magnitude which are caused by different positioning of the antenna. Such variation is common in practice and has almost no effect on EMMA's functionality because EMMA was designed to be robust to such variation.

after it, e.g., by hiding/removing the modified code before the checksum computation and restoring it after the (unmodified) checksum computation. These types of attacks are extremely difficult to detect for existing timing-based schemes since the time overhead introduced by these attacks are typically hundreds of microseconds which is less than 1% of the overall execution time of the attestation procedure. Moreover, frequency-only detection methods (e.g., EDDIE [31], Zeus [20]), are also unable to efficiently detect these attacks if the changes is smaller than the minimum time-resolution of the method (typically around 1 ms). However, for EMMA, these attacks are not difficult to detect, as they add many cycles of work between when the checksum computation is supposed to start and when it actually starts, thus a proper time-domain analysis such as the one used in this paper can be applied to detect potential attacks.

As an example of such attacks, we implement the Rootkit-Based Attack [11], which leverages Return-Oriented Programming (ROP) technique [39]. In this attack, a hook (jump) replaces the first instruction in the attestation. The initiation of attestation results in a jump to the malware's hiding functionality, which deletes the attacker's code (including itself) from program memory, but leaves a series of ROP gadgets so that, after the (unmodified) attestation code sends its response, the malware is re-installed on the device.

The *deleting* procedure is the most time-consuming part of the attack, where the adversary needs to copy the malware hiding functionality and the modified checksum loop to the data-memory, and replace them with the original code. Figure 6 shows the spectrogram of the attestation procedure with and without the rootkit-based attack. As can be seen in the figure, for the normal behavior of the attestation code, initialization takes about 500μs which includes receiving the challenge and invoking `attest()`. Note that based on the initialization time, we set the *threshold* for `challengeTimer` to 2ms or 8 samples (i.e. the maximum delay between sending the challenge and starting the checksum main loop is smaller than 2ms). As illustrated in Figure 6, in the presence of rootkit attack though, there is an extra phase between the initialization and the start of the main checksum loop that takes about 8ms, which is larger than the timer's threshold and thus triggers an error caused by checking the `challengeTimer`. Moreover, we found that even without using the timer, the time-domain analysis can successfully detect the attack since the time-domain signatures for the rootkit were very different than that of in receiving the challenge.

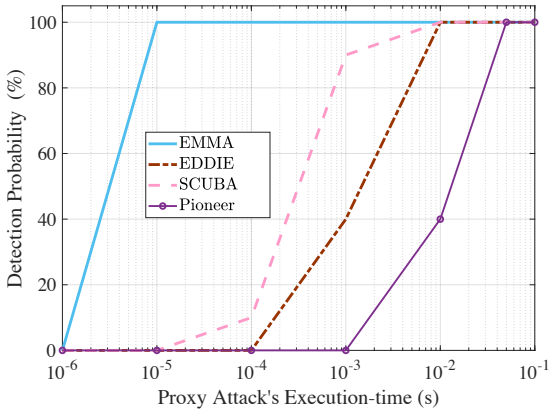

Figure 7: The probability of detecting the proxy attack for EMMA and state-of-the-art.

To evaluate EMMA against this attack, similar to previous attacks, we used 10 trial runs for the attack, and found that EMMA can successfully detect all the instances of the attack.

EP-2: Proxy Attack: In a proxy attack, instead of calculating the checksum on its own system, the prover contacts another often faster device (the proxy) to compute the correct checksum within the time limit, which enables malware on the device to go undetected. In this case, similar to a rootkit-based attack, the adversary needs some time after receiving the challenge to properly set up the attack. For proxy, this time is used for sending (forwarding) the challenge and any other necessary information (e.g., nonce) required to correctly compute the checksum in the proxy.

The major limitation in the existing methods for detecting proxy attacks is that the adversary can simply hide this attack if $t_{send} << t_{threshold}$. EMMA, however, is not limited by the overall attestation time and can distinguish the initialization phase from checksum calculation very accurately.

Figure 7 shows the probability of detecting this attack (as a function of t_{send}) and compares it with other existing methods. As can

be seen in this figure, EMMA can detect up to 2 orders of magnitude smaller attacks compared to existing work. This is particularly important in scenarios such as IoT devices where the device is connected to the network and the attacker can leverage proxy attacks very easily. Our evaluations show that for low-end devices such as Arduino, the fastest time the attacker can achieve to forward the challenge to another (colluding) device is about 800 μs (using a WiFi module) which is well within the detectable range of EMMA To evaluate EMMA against this attack, we used 10 trial runs for the proxy attack, and found that EMMA can successfully detect all the instances of the attack while having no false positives (while for other methods the accuracy is well below 80%).

5 FURTHER SENSITIVITY RESULTS AND ANALYSIS

Scalability to Other Platforms. To show EMMA is applicable to other systems with a different processor and/or architecture, and can be used at different frequency ranges, we tested our checksum main loop, `Checksum()` (an un-optimized form), on three other embedded systems: a TI MSP430 Launchpad with a processor clocked at 16MHz, an STM32 ARM Cortex-M Nucleo Board also clocked at 16MHz, and an Altera's Nios-II soft-core implemented on a Startix FPGA board clocked at 60MHz. The criteria for choosing these boards were to pick the embedded systems that are popular and widely used, and have different architectures than Arduino.

Running the same attestation code on these boards, we then confirmed that by using the same setup used in § 4, EM-Mon receives EM signals similar to that of for Arduino board (i.e., spikes at the frequency of the loop), and further, we confirmed that our detection algorithm can successfully detect when this loop starts and when it ends by adding the training information (i.e., the position and the number of spikes) for each board to our framework. Finally, to further show that even single added instruction to the loop's code is detectable by EMMA, we added a single-cycle "ADD" instruction to the checksum's assembly code for each of the mentioned boards (i.e., similar to *memory-shadow attack*)). We then used EMMA, to label malware-free and malware-afflicted (i.e., runs with the extra instruction) checksum runs (10 each) for all the three boards. Our results showed that, in all cases, EMMA successfully detected the malicious runs.

Scalability to More Complex Systems. We tested our framework on a more complex embedded system, A13-OLinuXino Single-Board-Computer [32]. This board has an ARM A8 in-order core clocked at 1GHz, with 2 level of caches, a branch predictor, and a prefetcher. It also runs a Debian Linux OS. We ran our checksum loop on this board and measured the beginning/end time and the loop's per-iteration time.

Our measurements showed that while having caches, a branch predictor, and a prefetcher introduces some variation in the per-iteration execution time of the loop, in practice this variation is not significant. For the cache, since the checksum code is small, it fits completely inside the CPU's L1 instruction cache. Furthermore, the memory region containing the verification function is small enough to fit inside the CPU's L1 data cache. Thus, once the CPU caches are warmed up, no more cache misses occur. The time taken to warm up the CPU caches is a very small fraction of the total execution time. As a result, the variance in the execution time caused by cache misses during the cache warm-up period is negligible.

For the branch predictor, we observed that in our code, the branch mis-prediction only happens in the last iteration (recall that the inner loop was enrolled), when the checksum computation is finished; thus, it does not have any impact on the execution time of the checksum loop. Furthermore, due to the memory's random access pattern in the checksum loop, the prefetcher's accuracy is inevitably low.

To further analyze the effect of having cache, branch predictor, and prefetcher on the timing, we used gem5 [6] simulator, to simulate the checksum code on an in-order ARM core machine with similar configurations to that of A13-OLinuXino board. Our results showed that for a 1.2KB size checksum code, our code accessed the cache about 7000 times, out of which only 21 accesses were L1 miss (i.e., >99.5% hit-rate), and only about 800 more cycles (mostly due to L2 misses) were added to the overall execution time (i.e., <0.01%). Note that this extra delay only happens inside the checksum loop, and does not affect the delay for the proxy and/or rootkit attacks since those attacks happen *before* the beginning of the checksum. Furthermore, our results showed no mis-prediction for the branch predictor, and <1% prefetching accuracy for the checksum loop.

To evaluate EMMA, we ran the same experiment (adding an extra "ADD" instruction) discussed in the previous section, and found that our detection algorithm successfully detected all instances of the attack with no false positive.

Overall, the goal of evaluating EMMA on multiple different boards was showing that the ability to use EM signals for monitoring the attestation procedure is a result of a fundamental connection between repetitive program behavior and the spectra of resulting side-channel signals, and is not dependent to a specific architecture. Furthermore, these experiments confirmed that EMMA is scalable to other platforms.

Scalability on Monitoring Multiple Devices. To show the ability of EMMA on monitoring multiple devices at the same, we used EMMA to monitor eight Arduino devices. For each device, we used a hand-made coil (cost around $3) as the measurement probe and taped it to the board (on the center of the board). We then used SMA cables and 2-way channel splitter/combiner (cost around $4), to connect all the probes to a single SDR (cost $30). The SDR is connected to a computer where EMMA was implemented. We then repeated the measurements in the previous section, by attesting these devices one-by-one. In all the experiments, we saw no significant degradation in the accuracy, and in all cases we were able to detect the attack with perfect accuracy which validates that the ability of EMMA to monitor multiple devices at the same time. Note that we further increased the devices to 16, but observed a much higher noise and significantly lower SNR, which prevented us from successfully perform the attestation.

Based on these experiments, with the current setup, EMMA is able to monitor up to 8 devices with no performance loss. The entire setup cost for monitoring is about $80 (i.e., $10 per device), which makes the system a practical approach for monitoring security-critical devices. Furthermore, multiple EMMA setups can be used to monitor a large group of devices where a C&C server can manage

the whole process and send and receive the results from individual EMMA setups.

Robustness and Variations Over Time. To test the robustness of our algorithm over time and against environment variations (e.g., temperature, interference, etc.), we repeat the attestation procedure at one-hour intervals, over a period of 24 hours, while keeping the Arduino board and the receiver active throughout the experiment, to observe how the emanated signals vary over time as device temperature (and room temperature) and external radio interference such as WiFi and cellular signals change during the day and due to the day/night transition. At each hour we ran the attestation once (without any malicious behavior). The training data was collected before the first hour of the experiment. The goal was to show how the false positive (FP) rate changes over time. We observed a significant increase in FP rate after hour 2 when we were not using the clock-adjustment feature (see § 3). However, adding this feature, EMMA achieved perfect accuracy (i.e., 0% FP). The major reason for this dramatic degradation in accuracy was due to the fact that the clock-rate for Arduino began to drift after about one hour of continuous usage. Without associating this drift to our algorithm, EMMA was unable to correctly predict that the shift in the frequency of the checksum main loop was due to the clock drift, and not because of a potential malicious activity.

6 RELATED WORK

Remote Attestation schemes can be categorized into three methods: software-based, hardware-based, and hybrid.

Software-based approaches require no hardware overhead on the prover. There is a large body of work in software-based attestation schemes. Seshadri et al. [38] introduced SWATT to verify the software of an embedded device. SWATT relies on a checksum function that computes a checksum over the entire memory contents and is constructed to force an attacker to induce overhead to compute the correct checksum. Furthermore, Seshadri et al. proposed an extension to SWATT by enabling verification of a small amount of memory on a sensor (ICE/SCUBA [36]). Li et al. [27] proposed a similar approach to verify the integrity of peripherals' firmware and showed that they can detect several known attacks, especially proxy attacks. However, this scheme also suffers from unwanted delays in communication between the prover and verifier and requires a low-latency low-contention network.

Castelluccia et al. [11] pointed out weaknesses in the specific SWATT and ICE-based schemes, and showed that all of these schemes are vulnerable to rootkit-based attacks, which also implemented and analyzed in this paper. We showed that using EM side-channel signals, we are able to not only precisely monitor the per-iteration behavior of checksum loop but also find when this loop begins and ends which enables us to detect the attacks shown in [11]. However, all the existing SW methods are *not* able to detect the exact begin/end time and/or per-iteration time of the checksum loop which makes them vulnerable to the attacks proposed in [11].

Hardware-based approach relies on a secure hardware (e.g., Trusted Platform Module [21, 25, 29, 40], SGX [14], TrustZone [4], etc.), which often present in powerful devices such as desktops, laptops and smartphones, and it is often impractical for medium- and low-end embedded systems due to costs and complexity overheads.

Hybrid-based techniques [9, 18, 23, 24, 44] are based on hardware/software co-design, and aims to reduce the hardware and cost overhead on the prover, however, they still incur some overhead to the system. Compared to these schemes, EMMA incurs no overhead to the system while providing a strong security guarantee.

Also related to this work, are schemes that use **side-channel signals** (e.g., EM or power) to profile or monitor a system [3, 8, 13, 20, 28, 31, 34]. Most notably, EDDIE [31] leveraged EM signals for malware detection in IoT devices, using a classifier based on a statistical distribution test. Similarly, ZEUS [20] used spectral components of the signal as a feature, and a neural network as a classifier, to detect a control-flow deviation in a PLC. While these two methods also used EM side-channel for malware detection there are several key differences between our approach and these schemes. The major difference between this method and the existing methods is that, in addition to leverage spectra to detect anomaly, EMMA also leverages time-domain analysis. As shown in Figure 7, this is particularly important to detect stealthy attacks such as proxy. Furthermore, instead of detecting deviations from normal behavior in an arbitrary application, the target in our design is a purpose-designed piece of code (checksum computation) that has very stable timing, and thus a sharp spectral signature, so the anomaly detection can be much more sensitive (comparing to EDDIE and/or ZEUS). Another difference is that we are also using the loop's per-iteration time and overall duration to estimate its iteration count, and deviation in this iteration count is also used for anomaly detection in EMMA. And of course here the EM monitoring is used in combination to a challenge-response attestation rather than as a stand-alone mechanism for detecting malware.

7 CONCLUSIONS

This paper proposed EMMA, a novel method for hardware/software attestation on embedded systems. Unlike existing approaches, EMMA uses electromagnetic side-channel signals generated by the embedded system as the communication channel between the verifier and prover which provides a more accurate scheme to monitor the prover during the attestation and completely eliminates the timing tight constraints in the existing methods. We described our attestation framework which can attest an embedded system in real-time. To evaluate our system, we implemented EMMA on a popular embedded system, Arduino UNO, and evaluated our system with a wide range of known attacks. Our evaluations showed that EMMA can provide an excellent security guarantee in the presence of these attacks while prior work failed to detect some (or all) these attacks with high accuracy. Further, we showed how EMMA can be scaled to attest multiple devices and support other embedded systems platforms, and demonstrated its robustness against different sources of variability.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable feedbacks. This work has been supported, in part, by NSF grant 1563991 and DARPA LADS contract FA8650-16-C-7620. The views and findings in this paper are those of the authors and do not necessarily reflect the views of NSF and DARPA.

REFERENCES

[1] AARONIA. 2016 (accessed Nov. 6, 2018). Datasheet: RF Near Field Probe Set DC to 9GHz. http://www.aaronia.com/Datasheets/Antennas/RF-Near-Field-Probe-Set.pdf.

[2] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 743–754. https://doi.org/10.1145/2976749.2978358

[3] Carlos R. Aguayo González and Jeffrey H. Reed. 2011. Power Fingerprinting in SDR Integrity Assessment for Security and Regulatory Compliance. *Analog Integr. Circuits Signal Process.* 69, 2-3 (Dec. 2011), 307–327. https://doi.org/10.1007/s10470-011-9777-4

[4] ARM. 2009. ARM Security Technology- Building a Secure System using TrustZone Technology. *ARM Technical White Paper* 2009 (2009).

[5] Frederik Armknecht, AhmadReza Sadeghi, Steffen Schulz, and Christian Wachsmann. 2013. A Security Framework for the Analysis and Design of Software Attestation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2508859.2516650

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[7] Ferdinand Brasser, Kasper B. Rasmussen, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. Remote Attestation for Low-end Embedded Devices: The Prover's Perspective. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, Article 91, 6 pages. https://doi.org/10.1145/2897937.2898083

[8] Robert Callan, Farnaz Behrang, Alenka Zajic, Milos Prvulovic, and Alessandro Orso. 2016. Zero-overhead Profiling via EM Emanations. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 401–412. https://doi.org/10.1145/2931037.2931065

[9] Xavier Carpent, Karim ElDefrawy, Norrathep Rattanavipanon, and Gene Tsudik. 2017. Lightweight Swarm Attestation: A Tale of Two LISA-s. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*. ACM, New York, NY, USA, 86–100. https://doi.org/10.1145/3052973.3053010

[10] X. Carpent, G. Tsudik, and N. Rattanavipanon. 2018. ERASMUS: Efficient remote attestation via self-measurement for unattended settings. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1191–1194. https://doi.org/10.23919/DATE.2018.8342195

[11] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. 2009. On the Difficulty of Software-based Attestation of Embedded Devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*. ACM, New York, NY, USA, 400–409. https://doi.org/10.1145/1653662.1653711

[12] Binbin Chen, Xinshu Dong, Guangdong Bai, Sumeet Jauhar, and Yueqiang Cheng. 2017. Secure and Efficient Software-based Attestation for Industrial Control Devices with ARM Processors. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)*. ACM, New York, NY, USA, 425–436. https://doi.org/10.1145/3134600.3134621

[13] Shane S. Clark, Benjamin Ransford, Amir Rahmati, Shane Guineau, Jacob Sorber, Kevin Fu, and Wenyuan Xu. 2013. WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices. In *Proceedings of the 2013 USENIX Conference on Safety, Security, Privacy and Interoperability of Health Information Technologies (HealthTech'13)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=2696523.2696532

[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.

[15] R. d. Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. d. Bosschere, B. Preneel, B. d. Sutter, and I. Verbauwhede. 2016. SOFIA: Software and control flow integrity architecture. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1172–1177.

[16] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A. Sadeghi. 2017. LO-FAT: Low-Overhead control Flow ATtestation in hardware. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/3061639.3062276

[17] Thomas Eisenbarth, Christof Paar, and Björn Weghenkel. 2010. Building a Side Channel Based Disassembler. In *Transactions on Computational Science X*, Marina L. Gavrilova, C. J. Kenneth Tan, and Edward David Moreno (Eds.). Springer-Verlag, Berlin, Heidelberg, 78–99. http://dl.acm.org/citation.cfm?id=1985581.1985585

[18] Karim Eldefrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik. 2012. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA*. San Diego, USA. http://www.eurecom.fr/publication/3536

[19] R. W. Gardner, S. Garera, and A. D. Rubin. 2009. Detecting Code Alteration by Creating a Temporary Memory Bottleneck. *IEEE Transactions on Information Forensics and Security* 4, 4 (Dec 2009), 638–650. https://doi.org/10.1109/TIFS.2009.2033231

[20] Yi Han, Sriharsha Etigowni, Hua Liu, Saman Zonouz, and Athina Petropulu. 2017. Watch Me, but Don'T Touch Me! Contactless Control Flow Monitoring via Electromagnetic Emanations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 1095–1108. https://doi.org/10.1145/3133956.3134081

[21] Rick Kennell and Leah H. Jamieson. 2003. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1251353.1251374

[22] Alexander Klimov and Adi Shamir. 2004. New Cryptographic Primitives Based on Multiword T-Functions. In *Fast Software Encryption*, Bimal Roy and Willi Meier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–15.

[23] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: A Security Architecture for Tiny Embedded Devices. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 10, 14 pages. https://doi.org/10.1145/2592798.2592824

[24] J. Kong, F. Koushanfar, P. K. Pendyala, A. R. Sadeghi, and C. Wachsmann. 2014. PUFatt: Embedded platform attestation based on novel processor-based PUFs. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2593069.2593192

[25] Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, and John Butterworth. 2012. New Results for Timing-Based Attestation. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 239–253. https://doi.org/10.1109/SP.2012.45

[26] Li Li, Hong Hu, Jun Sun, Yang Liu, and Jin Song Dong. 2014. Practical Analysis Framework for Software-Based Attestation Scheme. In *Formal Methods and Software Engineering*, Stephan Merz and Jun Pang (Eds.). Springer International Publishing, Cham, 284–299.

[27] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. 2011. VIPER: Verifying the Integrity of PERipherals' Firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, New York, NY, USA, 3–16. https://doi.org/10.1145/2046707.2046711

[28] Yannan Liu, Lingxiao Wei, Zhe Zhou, Kehuan Zhang, Wenyuan Xu, and Qiang Xu. 2016. On Code Execution Tracking via Power Side-Channel. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1019–1031. https://doi.org/10.1145/2976749.2978299

[29] Mohammad Mannan, Beom Heyn Kim, Afshar Ganjali, and David Lie. 2011. Unicorn: Two-factor Attestation for Data Security. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, New York, NY, USA, 17–28. https://doi.org/10.1145/2046707.2046712

[30] Mehari Msgna, Konstantinos Markantonakis, and Keith Mayes. 2013. The B-Side of Side Channel Leakage: Control Flow Security in Embedded Systems. In *Security and Privacy in Communication Networks*, Tanveer Zia, Albert Zomaya, Vijay Varadharajan, and Morley Mao (Eds.). Springer International Publishing, Cham, 288–304.

[31] Alireza Nazari, Nader Sehatbakhsh, Monjur Alam, Alenka Zajic, and Milos Prvulovic. 2017. EDDIE: EM-Based Detection of Deviations in Program Execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 333–346. https://doi.org/10.1145/3079856.3080223

[32] Olimex. 2016 (accessed Dec. 1, 2018). A13-OLinuXino-MICRO User Manual. https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/open-source-hardware.

[33] RTL-SDR. 2016 (accessed April 2019). v3. https://www.rtl-sdr.com/rtl-sdr-quick-start-guide/.

[34] Nader Sehatbakhsh, A. Nazari, A. Zajic, and M. Prvulovic. 2016. Spectral profiling: Observer-effect-free profiling by monitoring EM emanations. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–11. https://doi.org/10.1109/MICRO.2016.7783762

[35] Arvind Seshadri, Mark Luk, and Adrian Perrig. 2011. SAKE: Software Attestation for Key Establishment in Sensor Networks. *Ad Hoc Netw.* 9, 6 (Aug. 2011), 1059–1067. https://doi.org/10.1016/j.adhoc.2010.08.011

[36] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. 2006. SCUBA: Secure Code Update By Attestation in Sensor Networks. In *Proceedings of the 5th ACM Workshop on Wireless Security (WiSe '06)*. ACM, New York, NY, USA, 85–94. https://doi.org/10.1145/1161289.1161306

[37] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. 2005. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/1095810.1095812

[38] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. 2004. SWATT: softWare-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*. 272–282. https://doi.org/10.1109/SECPRI.2004.1301329

[39] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[40] Amit Vasudevan, Jonathan McCune, James Newsome, Adrian Perrig, and Leendert van Doorn. 2012. CARMA: A Hardware Tamper-resistant Isolated Execution Environment on Commodity x86 Platforms. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)*. ACM, New York, NY, USA, 48–49. https://doi.org/10.1145/2414456.2414484

[41] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. 2004. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. http://eprint.iacr.org/2004/199 no lai-xj@cs.sjtu.edu.cn 12647 received 16 Aug 2004, last revised 17 Aug 2004.

[42] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. 2005. Finding Collisions in the Full SHA-1. In *Advances in Cryptology – CRYPTO 2005*, Victor Shoup (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 17–36.

[43] Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao. 2007. Distributed Software-based Attestation for Node Compromise Detection in Sensor Networks. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS '07)*. IEEE Computer Society, Washington, DC, USA, 219–230. http://dl.acm.org/citation.cfm?id=1308172.1308237

[44] T. Zhang and R. B. Lee. 2015. CloudMonatt: An architecture for security health monitoring and attestation of virtual machines in cloud computing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 362–374. https://doi.org/10.1145/2749469.2750422