Remote Attestation for Low-End Embedded Devices: the Prover's Perspective

Ferdinand Brasser
Technische Universität Darmstadt
f.brasser@trust.tu-darmstadt.de

Kasper B. Rasmussen
University of Oxford
kasper.rasmussen@cs.ox.ac.uk

Ahmad-Reza Sadeghi
Technische Universität Darmstadt
a.sadeghi@trust.tu-darmstadt.de

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

Abstract

Security of embedded devices is a timely and important issue, due to the proliferation of these devices into numerous and diverse settings, as well as their growing popularity as attack targets, especially, via remote malware infestations. One important defense mechanism is *remote attestation*, whereby a trusted, and possibly remote, party (verifier) checks the internal state of an untrusted, and potentially compromised, device (prover).

Despite much prior work, remote attestation remains a vibrant research topic. However, most attestation schemes naturally focus on the scenario where the verifier is trusted and the prover is not. The opposite setting—where the prover is benign, and the verifier is malicious—has been side-stepped. To this end, this paper considers the issue of prover security, including: verifier impersonation, denial-of-service (DoS) and replay attacks, all of which result in unauthorized invocation of attestation functionality on the prover. We argue that protection of the prover from these attacks must be treated as an important component of any remote attestation method. We formulate a new *roaming* adversary model for this scenario and present the trade-offs involved in countering this threat. We also identify new features and methods needed to protect the prover with minimal additional requirements.

1. INTRODUCTION

Security of embedded devices is a popular and important research topic and this will likely remain the case for the foreseeable future. One contributing factor is the constantly increasing integration and introduction of such devices into many spheres of everyday life, including: automotive, avionics, factory automations, household, medical, and public utilities. At the same time, growing presence of computerized components in previously non-electronic (mechanical or simply analog) objects and tasks represents an attractive set of new and exciting attack surface for nefarious individuals and organizations.

Well-known incidents such as Stuxnet [9] and Duqu [36] are prominent examples of the impressive scale and penetration capabilities of remote malware infestations. They are also a preview of *future attractions*. Although these attacks did not target low-end devices, they (particularly, Stuxnet) allegedly succeeded in their mission of infecting numerous specialized industrial controllers. Meanwhile, researchers have also demonstrated numerous attacks on various embedded systems, including: automotive components [23], home appliances, and medical devices [27].

The research community recognized the danger posed by insecure embedded devices and responded with countermeasures. One important countermeasure is a distinct security service referred to as *attestation*: a means for a trusted party (*verifier*) to obtain the state of the, possibly remote, untrusted device (*prover*). Attestation is also an important building block, useful for constructing more specialized services, such as secure code update and secure memory erasure [30].

All attestation techniques involve a protocol between a trusted verifier and a potentially compromised prover. Even though this focus is both natural and sensible, it has overshadowed another important issue—attacks on the prover that can be launched through the attestation protocol itself. As we argue in this paper, such attacks that maliciously invoke attestation functionality on the prover pose a real threat, and any comprehensive attestation method must include a means to resist them. Attacks that involve verifier impersonation (e.g., via replay, reorder or delay) are particularly dangerous, since they amount to an effective denial-of-service (DoS). Such attacks can waste energy (deplete batteries) and take the targeted device away from performing its primary tasks, such as control, sensing, or actuation.

Goals & Contributions. In this paper, we identify and analyze DoS attacks that target provers running on low-end embedded devices. As the first step, we show how attestation protocols can be secured against a simple external adversary using well-known techniques. Then, we investigate a more sophisticated *roaming* adversary that compromises the prover and manipulates it in a way that is undetectable by standard attestation methods. Such attacks are particularly dangerous since the roaming adversary may erase all traces of its presence and remain stealthy. Next, we demonstrate—through two implementations—how the roaming adversary can be mitigated by extended attestation techniques for low-end embedded systems with minimal hardware assumptions. We believe that countermeasures developed in this paper represent a significant improvement and an advantage over current attestation techniques.

2. RELATED WORK

This section overviews prior results in device attestation. Given familiarity with the topic, it can be skipped with no loss in continuity.

Software-Based Attestation. There are many software-based attestation techniques. One early example is Pioneer [32]. It computes a checksum of the device memory using a function with side-effects (e.g., status registers) in its computation, such that any emulation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '16, June 05-09, 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-4236-0/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2897937.2898083

of this function incurs an additional timing overhead (delay) that is sufficient to detect cheating. Attestation that relies on time-based checksums has also been adapted to embedded devices in [30, 33, 14, 18, 19, 29, 31]. However, some basic assumptions that underlie these techniques are uncertain [34] and several attacks[1] on software-based attestation schemes have been demonstrated, e.g., [6, 16].

In general, all current software-only techniques rely on strong assumptions about adversarial capabilities, and only work if the verifier communicates directly to the prover, with no intermediate hops. While applicable to very specific settings (e.g., attestation of computer peripherals), this general approach is not viable for attestation performed over a network.

Secure Hardware-Based Attestation. An early example of this approach is Secure Boot [2]. In it, system integrity is verified at boot time: the *root of trust* is a small bootloader which computes a hash of the content loaded into memory, and compares this to a signed hash stored in secure ROM. A device is only allowed to boot if the two hashes match. Trusted platform modules (TPMs) [37] are present in many modern commercial systems, from smart-phones to laptops. A TPM can store an integrity checksum computed over the memory at boot time in protected memory called platform configuration registers (PCRs). The stored checksum can be sent to a remote verifier for validation. TPMs can also protect data against a compromised operating system, e.g., protect encryption keys from misuse. The root of trust is the TPM plus the BIOS that performs the very first extension upon boot. Several concrete architectures have been proposed that rely on a TPM as a foundation [26, 14].

Datta et al. [7] present a logic for secure systems, and use it to describe attestation protocols standardized by the trusted computing group (TCG), without providing a definition of attestation. [7] relies on the presence of a secure TPM device.

Dynamic Root of Trust (DRT). This is an extended mechanism added to TPM specifications [37]. It has been implemented by major vendors, e.g., Intel TXT [12] and AMD SVM [1]. Basically, DRT is a way to perform attestation *dynamically*, i.e., after boot. This is accomplished by allowing a specific CPU instruction to reset the state of some PCRs, isolate a memory region, hash and atomically execute its content. Flicker [21] is an architecture that establishes DRT on commodity computers. It takes advantage of Intel TXT and AMD SVM by executing a piece of application logic (PAL) on the prover. This architecture was extended by TrustVisor [20] that provides a dynamic root of trust for PALs from a minimal hypervisor. TrustVisor significantly improves performance of the DRT primitive. There are several other proposals that deal with establishment of trust on remote systems [13, 24, 25, 22, 38]. Underlying platforms range from Web servers to embedded systems.

Hybrid Techniques. SPM is a hardware-based mechanism for process isolation [35]. It relies on a special *vault* module bootstrapped from a static root of trust. This vault bootstraps the SPM-protected programs, which gains exclusive control over the protection of their own memory pages. Another proposal from 2011 is SMART [8]—a hardware-based scheme for establishing a dynamic root of trust in embedded devices. Its focus is on low-end microcontrollers (MCUs) that lack sophisticated features, such as specialized memory management or TPMs. SMART requires no additional hardware – only a few small changes to the MCUs. SPM and SMART share some key features, such as the use of program counters to restrict access to secret key storage, and code entry point enforcement. However, unlike SMART, SPM does not provides a dynamic root of trust. It also involves a larger TCB and is generally oriented towards higher-end embedded systems with an MMU or an MPU. Furthermore, SPM requires the addition of custom instructions to the core. Finally, its feasibility, i.e., the effort needed to implement SPM on a real hardware platform, and its overall *footprint* remain unclear.

[1]For example, vulnerabilities to Time-Of-Check-Time-Of-Use (TOCTOU) attacks identified in [16].

A refinement of SMART was recently presented in [10], where more precise specification of minimal architectural features needed to support attestation were derived.

Another follow-on is the TrustLite architecture for embedded systems [15] which augments SMART with support for so-called *trustlets*. TrustLite enables running arbitrary code (trustlets), isolated from the rest of the system. Such isolated code chunks are called *trustlets*. Similar to SMART, an *execution-aware memory protection unit* (EA-MPU) ensures that the data of a trustlet can be accessed only by the code of the trustlet to which the data belongs. Furthermore, EA-MPU can be used to control access to hardware components such as peripherals. Authenticity and confidentiality of the code and data of trustlets is ensured by means of secure boot. TrustLite can be seen as an extension of SMART. The main difference to SMART is that the memory access control rules of the EA-MPU in TrustLite can be programmed as required by trustlets. In contrast, memory access control rules of SMART are static. Also, TrustLite supports interrupt handling for trustlets, while the security-critical code in ROM of SMART cannot be interrupted during execution.

3. SYSTEM & ADVERSARY MODEL

As discussed earlier, an attestation protocol is an interaction between a prover (Prv) and a verifier (Vrf). Vrf needs to determine whether Prv is in a known (and therefore trusted) state. Vrf invokes the attestation protocol by sending a request (attreq) to Prv. We assume that Prv has a trust anchor responsible for measuring Prv's state and sending the result back to Vrf. To authenticate the attestation result, the trust anchor of Prv uses a key ($\mathsf{K}_{\mathsf{Attest}}$) shared with Vrf.

3.1 Attestation as Denial-of-Service

Remote attestation techniques typically assume that Vrf is trusted while Prv is not. However, Prv has no assurance whether it is interacting with a *real verifier*. Without authentication, the adversary can trivially impersonate the verifier by sending bogus attestation requests to the prover. Believing that a fake attestation request is genuine, Prv invokes its local attestation functionality, which results in a waste of energy (by depleting batteries) and takes Prv away from performing its primary tasks, such as control, sensing, or actuation.

If the adversary impersonates Vrf and causes Prv to perform attestation, some particular costs are incurred. First, executing attestation to compute a response typically involves computing a message authentication code (MAC) over the prover's entire writable memory. A MAC is usually implemented as either a CBC-based function based on a block cipher (such as AES) or a keyed hash function (such as SHA1-HMAC [17]). To illustrate MAC costs, Table 1 shows the time for computing a SHA1-HMAC on variable input sizes, using the *Intel Siskiyou Peak* embedded processor as the hardware platform [28]. Hashing its $512\,\mathrm{KB}$ of RAM takes $(512\,\mathrm{KB}/64\,\mathrm{B})\cdot 0.340\,\mathrm{ms} + 0.120\,\mathrm{ms} = 754.032\,\mathrm{ms}$

Furthermore, current low-end device attestation techniques assume that attestation runs without interruption [8, 35]. Thus, gratuitous (malicious) invocation of attestation can be detrimental to the execution of prover's main (even critical) functions. Techniques that perform attestation in a manner compliant with real-time op-

SHA1-HMAC [17]		AES-128 (CBC)			Speck 64/128 (CBC)			ECC (secp160r1)	
			per block			per block			
Fix	per block	Key exp.	Enc	Dec	Key exp.	Enc	Dec	Sign	Verify
0.340	0.092	0.074	0.288	0.570	0.016	0.017	0.015	183.464	170.907

Table 1: Performance (in milliseconds) of cryptographic primitives on Intel Siskiyou Peak at $24\,\mathrm{MHz}$.

eration [5] do not fully address this problem, since they require a managing software layer, e.g., an operating system. This makes them inapplicable to low-end embedded devices.

The main reason for the ease of such DoS attacks is that the prover's work-load is much higher than that of the verifier. This asymmetry is not limited to the sheer *amount* of work performed by each party; it also occurs because the verifier is generally much more powerful than the prover, which might be a low-end MCU.

3.2 Adversaries

We now define two types of adversaries envisaged in the context of verifier impersonation and DoS attacks on the prover. Neither type is capable of physical attacks.

External Adversary $\mathcal{A}dv_{ext}$. We first consider the *external* adversary ($\mathcal{A}dv_{ext}$) that can control all communication between Prv and Vrf. $\mathcal{A}dv_{ext}$ can drop, insert and delay messages, following the well-known Dolev-Yao model. However, being strictly external, $\mathcal{A}dv_{ext}$ cannot directly manipulate any internal state of Prv.

Roaming Adversary $\mathcal{A}dv_{roam}$. A stronger and more sophisticated adversary can, in addition to $\mathcal{A}dv_{ext}$'s capabilities, infect Prv with malware and later *cover its tracks* by erasing that malware. We call it a *roaming* adversary ($\mathcal{A}dv_{roam}$). As with $\mathcal{A}dv_{ext}$, we assume that $\mathcal{A}dv_{roam}$'s primary goal is DoS on Prv. $\mathcal{A}dv_{roam}$ operates in three phases:

- Phase I: eavesdrops on genuine Vrf-Prv attestation requests.
- Phase II: compromises Prv by introducing malware, changes local state, and leaves Prv, i.e., erases all traces of its presence.
- Phase III: replays previously recorded attestation requests.

Note that, in Phase II, $\mathcal{A}dv_{roam}$ only changes dynamic data on Prv. This is not detectable by subsequent attestation. Also, $\mathcal{A}dv_{roam}$ can extract other information from Prv, e.g., authentication key $\mathsf{K}_{\mathsf{Attest}}$.

In the following section we discuss mitigation techniques against these adversaries. In the process, we also identify the requirements they pose on underlying attestation protocols and device hardware features.

4. MITIGATING $\mathcal{A}dv_{ext}$

We now turn to $\mathcal{A}dv_{ext}$ mitigation strategies, starting with request authentication and proceeding to replay (and related) attack countermeasures.

4.1 Authenticating Verifier Requests

It is quite evident that, in order to mitigate bogus attestation requests (or, equivalently, verifier impersonation attacks) verifier must authenticate itself to the prover. This can be done via either public key or symmetric cryptography. With the former, verifier signs its attestation request and the prover authenticates it with the verifier's public key, which must reside in some non-malleable memory on the prover. With the latter, assuming the two parties share a secret key, the verifier appends a MAC to the attestation request and the prover recomputes the same on its side, thus authenticating the request.

As can be expected, public key cryptography is expensive for low-end MCU-s. Table 1 shows that even relatively efficient elliptic curve cryptography (ECC) incurs a computational cost for the prover $170\,\mathrm{ms}$. Thus, with the use of ECC, simply authenticating verifier's authentication request can be viewed as a kind of DoS. In other words, we have a paradoxical situation where a supposed way of preventing DoS attacks can itself result in DoS. Consequently, we rule out the use of public key cryptography in this context.

Using symmetric cryptography to secure authenticated attestation requests yields significantly better performance: a SHA-1-based HMAC can be validated in $0.430\,\mathrm{ms}$. Standard block ciphers such as AES perform slightly better. Using lightweight block ciphers such as Speck [4] reduces the cost even further, to $0.015\,\mathrm{ms}$, if key expansion is done in advance. Messages are assumed to fit into one block for each cryptographic primitive (in bits): ECC: 160, AES: 256, Speck: 64; and HMAC: 512.

Finally, we note that the use of symmetric cryptography to compute a MAC and the requirement to protect this key by access-restricted hardware-protected key storage (e.g., in ROM) is not new. It is already mandated by recent attestation architectures such as SMART, SPM and TrustLite, for the purposes of the prover computing an authenticated response, i.e., a challenge-based MAC over prover's memory.

4.2 Handling Replay, Reorder & Delay

Unfortunately, mere authentication of attestation requests is insufficient to mitigate DoS attacks. $\mathcal{A}dv_{ext}$ can simply eavesdrop on genuine attestation requests and later replay them. Alternatively, $\mathcal{A}dv_{ext}$ can intercept and arbitrarily delay and/or reorder genuine requests. There are several standard ways of detecting replay, reordering and delay attacks:

- *Nonces:* If each attestation request includes a nonce (i.e., a unique value) provided by the verifier, the prover can keep a complete nonce history of previously received (and authenticated) attestation requests. A replayed request is thus detectable.
- *Counters:* If each attestation request includes a monotonically increasing counter, the prover accepts a new request only if its counter is strictly greater than the last one received and processed. The new counter then replaces the previous one. Requests bearing out-of-order or duplicate counters are rejected.
- *Timestamps:* Assuming synchronized clocks among both parties and sufficiently inter-spaced genuine attestation requests, a verifier's timestamp included in the attestation request allows the prover to detect replayed, reordered and delayed messages.

It is easy to see that using nonces is problematic, for two reasons. First, keeping a complete nonce history requires a lot of non-volatile memory on the prover. Second, it only protects against replays, while reordered or delayed requests cannot be detected. Therefore, in the rest of the paper we rule out the use of the nonce history.

Keeping a counter (i.e., a sequence number) also requires the prover to have non-volatile memory, although only a small and fixed amount thereof. Note that, assuming that non-volatile memory is already available on the underlying prover MCU, keeping a counter requires no new architectural features on top of those identified in [8] as necessary and sufficient to support secure remote attestation in the trusted-verifier model. On the other hand, a counter does not protect against delayed request attacks.

Attack:	Feature: Nonces	Counter	Timestamps
Replay	✓	✓	✓
Reorder	-	✓	✓
Delay	-	-	✓

Table 2: Summary of DoS attack mitigation features.

Timestamps offer the best security, under aforementioned assumptions. However the major requirement imposed by timestamps is availability of a reliable real-time clock on the prover – a feature not previously identified as necessary for attestation.

Table 2 summarizes security features attainable with each approach.

5. MITIGATING $\mathcal{A}dv_{roam}$

We first show that $\mathcal{A}dv_{ext}$ countermeasures discussed above can be easily defeated by $\mathcal{A}dv_{roam}$ and then develop mitigation techniques to protect against $\mathcal{A}dv_{roam}$.

- $\mathcal{A}dv_{roam}$ and Counters: We assume that: (1) each attestation request contains a monotonically increasing counter, and (2) the prover stores the counter from the last genuine attestation request in non-volatile memory. Without loss of generality, we assume that, in Phase I, $\mathcal{A}dv_{roam}$ records just one genuine attestation request $\mathsf{attreq}(i)$ where i denotes the counter. In Phase II, $\mathcal{A}dv_{roam}$ modifies the counter stored by the prover from i to $i-1$. It then leaves the prover, and after waiting arbitrary length of time, replays $\mathsf{attreq}(i)$. After checking its stored (modified) last counter, the prover accepts $\mathsf{attreq}(i)$ as fresh and performs attestation. The prover's counter is changed to i.
- $\mathcal{A}dv_{roam}$ and Timestamps: We assume that: (1) each attestation request is timestamped by the verifier, (2) the prover has a clock, and (3) prover's and verifier's clocks are synchronized. Again, we also assume that, in Phase I (see Section 3.2), $\mathcal{A}dv_{roam}$ records just one genuine attestation request $\mathsf{attreq}(t_i)$ where t_i denotes the timestamp. In Phase II, $\mathcal{A}dv_{roam}$ re-sets the prover's clock to time $t_i - \delta$. It then leaves the prover, and after waiting for δ time units, replays $\mathsf{attreq}(t_i)$. After consulting its (modified) clock, the prover accepts it as timely and performs attestation. The prover's clock remains behind.

Although the DoS attack succeeds in both cases, there are two subtle differences: First, $\mathcal{A}dv_{roam}$ is more constrained with timestamps, since it is bound to δ wait time before replay in Phase III. Second, resetting the prover's clock in Phase II leaves some evidence of the attack since the prover's clock remains behind. In contrast, resetting the counter allows $\mathcal{A}dv_{roam}$ to bring the prover back to its expected state. In other words, the DoS attack is undetectable after the fact.

Protecting Keys, Counters & Clocks: In Phase II, $\mathcal{A}dv_{roam}$ compromises the prover. At this time, it can take certain actions to prepare for the actual DoS attack in subsequent Phase III. For example, $\mathcal{A}dv_{roam}$ could extract Prv's $\mathsf{K_{Attest}}$ which would allow it to generate authentic attreq-s. Hence, $\mathsf{K_{Attest}}$ must be protected from read access, except by the trusted attestation code $\mathsf{Code_{Attest}}$. Note that this is impossible in software-based attestation (see Section 2). Similarly, $\mathsf{K_{Attest}}$ must be write-protected; otherwise, $\mathcal{A}dv_{roam}$ could overwrite it with any key it chooses and achieve the same result. The counter in the last authentic (processed) attreq as well as Prv's local clock state must not be modifiable by $\mathcal{A}dv_{roam}$, in order to prevent replay, delay and reorder attacks described in Section 4.2. At the same time, the last counter must be writable only by immutable $\mathsf{Code_{Attest}}$ on Prv.

In summary, $\mathsf{K_{Attest}}$ must be confidential and non-malleable, i.e., read-only and readable only by $\mathsf{Code_{Attest}}$. Meanwhile, the counter and the clock must be write-protected. This protection can be achieved with minimal hardware security mechanisms. Low-end device attestation techniques already rely on hardware means to protect $\mathsf{Code_{Attest}}$ and $\mathsf{K_{Attest}}$ against software attacks. The same means can be used to protect the counter and the clock, as described in the next section.

6. IMPLEMENTATION

We now describe a prototype of proposed countermeasures. First, we overview hardware protection mechanisms from previous attestation techniques. Then, we describe how to protect against $\mathcal{A}dv_{roam}$ using those mechanisms. Finally, we provide evaluation results showing the costs of our countermeasures.

6.1 Background

The primitive used to restrict access to critical components of the system (i.e., $\mathsf{K_{Attest}}$, the counter and the clock) is called execution-aware memory access control (EA-MAC). It has been used in several prior proposals [8, 15], as discussed in Section 2. The main idea of EA-MAC is to limit read and/or write memory access depending on currently executing code. For example, in case of $\mathsf{K_{Attest}}$, it means that only $\mathsf{Code_{Attest}}$ can read it. Hence, if all code (except $\mathsf{Code_{Attest}}$) is compromised, $\mathsf{K_{Attest}}$ remains protected. $\mathsf{Code_{Attest}}$ itself is non-malleable, e.g., in SMART [8] it is resident ROM. TrustLite [15] and TyTAN [5] use secure boot and EA-MAC-based isolation to maintain $\mathsf{Code_{Attest}}$ integrity.

The basic operation of EA-MAC is roughly the same in all attestation architectures: the CPU allows a particular memory access based on the value of the current program counter (PC). However, specific implementations differ. For example, SMART has one memory element ($\mathsf{K_{Attest}}$) protected with a hard-wired EA-MAC. In contrast, TrustLite allows flexible configuration of protected memory regions and associated access policies at runtime, by software. An extended memory protection unit (MPU) specifies which code region has access to which data region. Notably, controlled-access memory regions can also contain the memory-mapped configuration registers of peripheral devices.

6.2 Implementation Details

The prototype of $\mathcal{A}dv_{roam}$ countermeasures is based on three components: ROM, EA-MAC and secure boot. Figure 1 shows two prototype versions: Figure 1a is the basic version while Figure 1b shows a more advanced one, which requires no new hardware features at the price of more complex software implementation. Both versions are based on the TrustLite attestation architecture. The same countermeasures are easily adaptable to other attestation techniques, such as SMART or TyTAN. However, due to space limitations, we omit them.

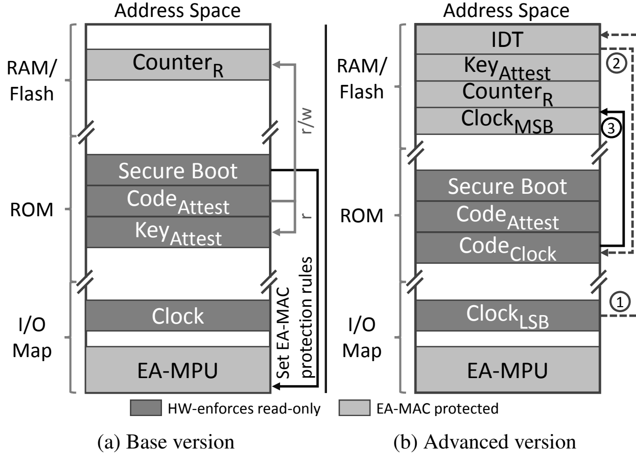
As discussed earlier, to mitigate $\mathcal{A}dv_{roam}$, three components must be protected: $\mathsf{K_{Attest}}$, the counter, and the real-time clock.

Secure Boot. Protection of critical system components is realized by setting appropriate memory access rules in EA-MPU. However, if the adversary controls systems software, it could change those rules and disable protection. For this reason, the system is started via secure boot, i.e., at boot time it verifies that correct software is loaded. This initial software sets up memory protection rules in the EA-MPU and locks it down to preclude further changes. This can be done by the EA-MPU itself, via memory-mapped configuration registers. Setting the EA-MPU-s configuration registers as read-only protects EA-MPU rules from being changed, as in Figure 1a.

Keys & Counters. The secret attestation key must be both read- and write-protected. In ROM, it is inherently write-protected. Otherwise,

	Siskiyou Peak	EA-MPU (TrustLite)	Attest-Key	Counter	64 bit clock	32 bit clock	SW-clock
EA-MPU rules	0	1	1	1	0	0	2
Register	5528	$278 + (116 \cdot \#r)$	0	0	64	32	0
Look-up Table	14361	$417 + (182 \cdot \#r)$	0	0	64	32	0

Table 3: Hardware cost per component ($\#r$ is the number of protection rules configurable in EA-MPU).



(a) Base version (b) Advanced version

Figure 1: *(a) Base version of $\mathcal{A}dv_{roam}$ mitigation;* $\mathsf{K}_{\mathsf{Attest}}$ *and* $\mathsf{counter}_{\mathsf{R}}$ *are only accessible by* $\mathsf{Code}_{\mathsf{Attest}}$. *Access control is enforced by EA-MPU set up at system start by a secure boot mechanism. (b) Advanced version for a common low-end MCU clock design;* $\mathsf{Clock}_{\mathsf{LSB}}$ *is a short-term counter which issues an interrupt when it wraps around ①. The immutable interrupt handling engine ensures that* $\mathsf{Code}_{\mathsf{Clock}}$ *serves the interrupt ②;* $\mathsf{Code}_{\mathsf{Clock}}$ *maintains a software counter (*$\mathsf{Clock}_{\mathsf{MSB}}$*) such that* $\mathsf{Clock}_{\mathsf{MSB}} + \mathsf{Clock}_{\mathsf{LSB}}$ *form a real-time clock ③.*

if it is stored in writable memory (e.g., RAM or Flash), it must be write-protected by a dedicated EA-MAC rule. In both cases, an EA-MAC rule is required to ensure that only $\mathsf{Code}_{\mathsf{Attest}}$ can read $\mathsf{K}_{\mathsf{Attest}}$. Hence, $\mathcal{A}dv_{roam}$ that controls all software (except $\mathsf{Code}_{\mathsf{Attest}}$) can neither read nor write the $\mathsf{K}_{\mathsf{Attest}}$ location. At the same time, $\mathcal{A}dv_{roam}$ cannot modify $\mathsf{Code}_{\mathsf{Attest}}$, which is write-protected (in ROM). Runtime attacks on $\mathsf{Code}_{\mathsf{Attest}}$ can be addressed, e.g., by limiting code entry points, or using control-flow integrity (CFI) [3]. Similarly, $\mathsf{counter}_{\mathsf{R}}$ is protected by an EA-MAC rule that allows it to be writable by $\mathsf{Code}_{\mathsf{Attest}}$ only.

Real-Time Clock. A real-time clock is needed if protection against delay attacks is required. Recall that a counter is enough to mitigate replay and reorder attacks. Obviously, the clock must be write-protected. In the simplest case, the clock counter is sufficiently large bit-wise (see Section 6.3) so that it does not wrap around within the expected lifetime of the prover, as in Figure 1a.

Figure 1b shows a set-up with a short-term counter ($\mathsf{Clock}_{\mathsf{LSB}}$) which causes an interrupt at wrap-around. The interrupt is handled by the trusted and integrity-protected $\mathsf{Code}_{\mathsf{Clock}}$. This code maintains higher-order bits of the clock in writable memory ($\mathsf{Clock}_{\mathsf{MSB}}$). Again, the hardware counter must be read-only and $\mathsf{Clock}_{\mathsf{MSB}}$ memory must be protected with an EA-MAC rule, so it is writable only by $\mathsf{Code}_{\mathsf{Clock}}$. Also, system interrupt handling must itself be protected. For example, if $\mathcal{A}dv_{roam}$ manipulates the interrupt descriptor table (IDT), it could preclude $\mathsf{Code}_{\mathsf{Clock}}$ being invoked upon a wrap-around of $\mathsf{Clock}_{\mathsf{LSB}}$, thus effectively stopping the real-time clock. To prevent this, IDT can be locked down similar to the EA-MPU, by setting up a read-only access rule for the memory region storing IDT. Depending on the underlying MCU platform, disabling the timer interrupt must also be prevented. Moreover, the location of the IDT itself must be immutable.

6.3 Evaluation

In this section, we evaluate the costs of prototyped mechanisms. The cost of protecting $\mathsf{K}_{\mathsf{Attest}}$ and $\mathsf{counter}_{\mathsf{R}}$ is the same in all variants, i.e., one EA-MAC policy for each. This is independent of the storage location of $\mathsf{K}_{\mathsf{Attest}}$, since the cost of read protection (in ROM) is identical to the cost of read/write protection in RAM or Flash.

Clock Implementations. We considered two alternatives. The first involves a dedicated counter register that does not wrap around within the lifetime of the prover. For example, a $64\,\mathrm{bit}$ register incremented every clock cycle wraps around after $24{,}372.6\,\mathrm{years}$ on a $24\,\mathrm{Mhz}$ CPU. Table 3 shows the hardware cost for a counter register of this size as well as combinational logic for incrementing it. Register size can be decreased by changing the clocking frequency. For example, given a $32\,\mathrm{bit}$ register, the wrap-around time is about $3\,\mathrm{minutes}$ at $24\,\mathrm{Mhz}$. By dividing the clock by $2^{20} = 1{,}048{,}576$ (i.e., incrementing it every one millionth cycle), wrap-around can be increased to $6\,\mathrm{years}$ while keeping clock resolution at $42\,\mathrm{ms}$.

The second implementation is based on a clock design in Intel Siskiyou Peak and other popular low-end MCUs, e.g., TI MSP430 [11]. Hence, the clock involves no additional hardware cost. Protecting this type of a clock incurs the costs of: (1) protecting interrupt handling, and (2) storage of software-maintained share of the current clock value. Table 3 shows the cost of this "software clock" (SW-clock) which consists of two EA-MPU protection rules: one to set IDT as immutable, and another—to protect the location of the high-order bits of the clock value, $\mathsf{Clock}_{\mathsf{MSB}}$ in Figure 1b.

Overhead. We compare our implementations against a base-line system that supports attestation without protection against $\mathcal{A}dv_{ext}$ or $\mathcal{A}dv_{roam}$. The base-line needs an EA-MPU with at least two rules: one to lock down the EA-MPU itself, and the other – to protect $\mathsf{K}_{\mathsf{Attest}}$. The total cost of the base-line system is $5528 + 278 + (116 \cdot 2) = 6038$ registers and $14361 + 417 + (182 \cdot 2) = 15142$ LUTs; see Table 3 columns "Siskiyou Peak" and "EA-MPU". For the $64\,\mathrm{bit}$ clock implementation, we need an additional EA-MPU rule, plus the direct cost of the clock: $116 + 64 = 180$ registers and $182 + 64 = 246$ LUTs, which is 2.98% and 1.62% of the overall cost, respectively. For the $32\,\mathrm{bit}$ clock with a divider, the cost is $116 + 32 = 148$ registers and $182 + 32 = 214$ LUTs, which is 2.45% and 1.41%, respectively.

The SW-clock implementation requires three new EA-MPU rules: $116 \cdot 3 = 348\,\mathrm{Registers}$ and $182 \cdot 3 = 546\,\mathrm{LUTs}$ which is 5.76% and 3.61% of the overall cost, respectively.

7. CONCLUSIONS & FUTURE WORK

Prior attestation methods assume a trusted verifier and an untrusted prover. We have shown that, verifier impersonation and prover-bound DoS are serious threats, largely overlooked by prior work. We formulated a new roaming adversary model and developed and evaluated techniques to protect the prover from attacks by

this powerful adversary. We also showed that desired protection can be achieved with low additional hardware cost. For future work we plan to:

1. Trial-deploy proposed methods in the context of connected devices, such as Internet of Things (IoT).
2. Develop mechanisms for secure and reliable synchronization of verifier's and prover's clocks.
3. Generalize proposed techniques to other network protocols (beyond attestation) to mitigate DoS attacks on other security services on embedded devices.

Acknowledgments

We thank the anonymous reviewers. This work was funded in part by: (1) German Science Foundation project S2, CRC 1119 CROSSING, (2) EU Horizon 2020 Research and Innovation Program, grant No. 643964 SUPERCLOUD, (3) Intel CRI for Secure Computing, (4) National Security Agency grant No. H98230-15-1-0276, and (5) Department of Homeland Security (under subcontract from HRL Laboratories) contract No. 14089-503866-DS.

8. REFERENCES

[1] Advanced Micro Devices. AMD, Secure Virtual Machine Architecture Reference Manual, 2005.

[2] W. A. Arbaugh, D. J. Farbert, and J. M. Smith. A Secure and Reliable Bootstrap Architecture . *IEEE S&P*, 1997.

[3] O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan. HAFIX: Hardware-Assisted Flow Integrity Extension. In *ACM DAC*, 2015.

[4] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404, 2013.

[5] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. TyTAN: Tiny Trust Anchor for Tiny Devices. In *ACM DAC*, 2015.

[6] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the Difficulty of Software-Based Attestation of Embedded Devices. In *ACM CCS*, 2009.

[7] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A Logic of Secure Systems and its Application to Trusted Computing. In *IEEE S&P*, 2009.

[8] K. E. Defrawy, A. Francillon, D. Perito, and G. Tsudik. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *NDSS*, 2012.

[9] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. *Symantec*, 2010.

[10] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A Minimalist Approach to Remote Attestation. In *DATE*, 2014.

[11] Girard, Olivier. openMSP430. http://opencores.org/project,openmsp430.

[12] Intel Corporation. *Intel Trusted Execution Technology (Intel TXT) – Software Development Guide*, 2009.

[13] R. Kennell and L. H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *USENIX Security*, 2003.

[14] 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 *IEEE/IFIP DSN*, 2009.

[15] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. TrustLite: A Security Architecture for Tiny Embedded Devices. In *ACM EuroSys*, 2014.

[16] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New Results for Timing-Based Attestation. In *IEEE S&P*, 2011.

[17] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, 1997.

[18] Y. Li, J. M. McCune, and A. Perrig. SBAP: Software-Based Attestation for Peripherals. In *TRUST*. Springer, 2010.

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

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

[21] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *ACM EuroSys*, 2008.

[22] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. How Low Can You Go? Recommendations for Hardware-Supported Minimal TCB Code Execution. *ACM ASPLOS*, 2008.

[23] C. Miller and C. Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. In *Blackhat USA*, 2015.

[24] C. Nie. Dynamic Root of Trust in Trusted Computing. *TKK T1105290 Seminar on Network Security*, 2007.

[25] B. J. Parno, J. M. McCune, and A. Perrig. Bootstrapping Trust in Commodity Computers. In *IEEE S&P*, 2010.

[26] S. Pearson, M. C. Mont, and S. Crane. Persistent and Dynamic Trust: Analysis and the Related Impact of Trusted Platforms. *iTrust*, 2005.

[27] J. Radcliffe. Hacking Medical Devices for Fun and Insulin: Breaking the Human SCADA System. In *Blackhat USA*, 2011.

[28] J. Rattner. *Extreme Ccale Computing*.

[29] A. Seshadri, M. Luk, and A. Perrig. SAKE: Software Attestation for Key Establishment in Sensor Networks. In *IEEE DCOSS*. Elsevier, 2008.

[30] A. Seshadri, M. Luk, A. Perrig, L. V. Doorn, and P. Khosla. SCUBA: Secure Code Update By Attestation in Sensor Networks. In *ACM WiSec*, 2006.

[31] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Using FIRE & ICE for Detecting and Recovering Compromised Nodes in Sensor Networks. Technical report, DTIC Document, 2004.

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

[33] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *IEEE S&P*, 2004.

[34] U. Shankar, M. Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *USENIX Security*, 2004.

[35] R. Strackx, F. Piessens, and B. Preneel. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *SecureComm*. Springer, 2010.

[36] Symantec. W32.Duqu - The precursor to the next Stuxnet, 2011.

[37] Trusted Computing Group. *TPM Main Specification Level 2 Version 1.2*.

[38] Q. Yan, J. Han, Y. Li, and R. Deng. A Software-Based Root-of-Trust Primitive on Multicore Platforms. In *ACM ASIACCS*, 2011.