Systematic Treatment of Remote Attestation

Aurelien Francillon
Networking and Security Group
Eurecom Instutute
aurelien.francillon@eurecom.fr

Quan Nguyen   Kasper B. Rasmussen,   Gene Tsudik
Department of Computer Science
University of California, Irvine
{quann1,kbrasmus,gene.tsudik}@uci.edu

***Abstract*—Embedded computing devices (such as actuators, controllers and sensors of various sizes) increasingly permeate many aspects of modern life: from medical to automotive, from building and factory automation to weapons, from critical infrastructures to home entertainment. Despite their specialized nature as well as limited resources and connectivity, these devices are now becoming increasingly popular and attractive targets for various attacks, especially, remote malware infestations. There has been a number of research proposals to detect and/or mitigate such attacks. They vary greatly in terms of application generality and underlying assumptions. However, one common theme is the need for *Remote Attestation*, a distinct security service that allows a trusted party (verifier) to check the internal state of a remote untrusted embedded device (prover).**

This paper provides a systematic treatment of Remote Attestation, starting with a precise definition of the desired service and proceeding to its systematic deconstruction into necessary and sufficient properties. These properties are, in turn, mapped into a minimal collection of hardware and software components that results in secure Remote Attestation. One distinguishing feature of this line of research is the need to prove (or, at least argue) architectural minimality; this is rarely encountered in security research. This work also offers some insights into vulnerabilities of certain prior techniques and provides a promising platform for attaining more advanced security services and guarantees.

***Keywords*-Remote Attestation, Embedded Device, Architectural Minimality.**

I. Introduction

"Embedded system" is a broad notion that encompasses many kinds of specialized computing devices, such as controllers, actuators, sensors, RFID tags, peripherals and all possible hybrids thereof. Embedded systems vary greatly in terms of resources and intended purposes. Unlike general-purpose computers that, for decades, have been the primary attack victims, embedded systems have been targeted only relatively recently. The Stuxnet [10] incident pointedly demonstrated the impressive scope and impact of malware on embedded devices. Stuxnet specifically targeted Programmable Logic Controllers (PLC) in industrial control systems and by modifying PLC control parameters, ostensibly caused some serious physical damage.

Stuxnet should be viewed as both an example and a warning of things to come. It epitomizes the power and the amplification factor of *remote* attacks, i.e., those not requiring direct access to victim devices. With the growing presence and proliferation of embedded devices into many spheres of life (e.g., automation of homes, factories, office buildings as well as automotive, aerospace and public utilities sectors), remote attacks have become a clear and present danger. This motivates the necessity for countermeasures, a number of which have been proposed by the research community and some have been implemented by manufacturers. One common theme among them is the need for **Remote Attestation** – a security service that involves verification of internal state of a remote embedded device. Remote Attestation is not a panacea; however, it should ideally allow for efficient and accurate detection of remote malware attacks.

Prior research results have underscored the difficulty of the problem. We believe that, although ad hoc or specialized solutions might work in the near term or for a narrow range of devices, only systematic approaches to Remote Attestation that offer concrete security guarantees are likely to prove effective in the long run. This assertion forms the starting point for the work described in this paper.

Throughout this paper, the term *Remote Attestation* denotes *attestation performed across a network*. In this scenario, software attestation [15], [18], [19], [30]–[34] is not applicable. Indeed, software attestation only works if the verifier communicates directly to the prover, with no intermediate hops. We outline a number of issues with software attestation in Appendix A. While software attestation cannot be applied, the use of secure hardware components, such as TPMs [38] or secure co-processors [36], represent a significant cost barrier for low-end embedded devices. Instead, we believe that the promise lies in a careful analysis of Remote Attestation, including a systematic identification of its necessary and sufficient components. This should, in turn, ultimately result in the design of a generic and practical embedded system architecture for Remote Attestation.

In this paper, we follow the above path: starting with the definition of Remote Attestation, we derive the properties needed to attain it. These properties are then translated into architectural components. Next, we map those into a small set of hardware features that collectively achieve the required properties. We argue that the set of identified features form the minimal generic architecture for Remote Attestation. In the process, we remain agnostic with respect to the underlying

hardware by making as few as possible assumptions about specific devices. We believe that the outcome of this effort is valuable as it represents the first attempt to systematically explore the notion of Remote Attestation and to produce a light-weight blueprint that can be realized on wide range of devices, with minimal modifications.

The rest of this paper is organized as follows: Section II overviews related work. Section III defines a security notion for Remote Attestation and presents our system and adversary models. Section IV presents the properties required for our notion of security, and Section V shows how these properties can be realized with concrete hardware features. Next, Section VI illustrates a remote attestation protocol and Section VII discusses some attacks on one recent (ostensibly minimal) remote attestation proposal. Section VIII contains a brief discussion and the paper concludes with a summary in Section IX.

II. Related Work

Software-Based Attestation. There have been several proposals for software-based attestation. One early example is Pioneer [33] that provides device attestation without relying on a secure co-processor or any specialized hardware. It computes a checksum of device memory using a function that includes side-effects (e.g., status registers) in its computation, such that any emulation of this function incurs a timing overhead that is sufficient to detect cheating. Attestation that relies on time-based checksums has also been adapted to embedded devices in [15], [18], [19], [30]–[32], [34]. However, some assumptions that form the basis for these solutions have been challenged [35] and several attacks on these (and similar) schemes have been proposed [6]. Moreover, Kovah et al. [16] showed that some time-based attestation schemes may be vulnerable to Time Of Check, Time Of Use(TOCTOU) attack.

Alternative (non-time-based) approaches (e.g., [26]) rely on filling the entire memory of the prover with random data to ensure absence of malicious code. Although timing is not essential here, this approach is still limited to one-hop attestation since it lacks the means to authenticate a remote prover.

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

Hardware-Based Attestation. An early example of this approach is Secure Boot [4]. 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. Nowadays, trusted platform modules (TPMs) [38] are present in many commercial systems, from phones to laptops. A TPM can compute an integrity checksum over the memory at boot time and send this checksum to be validated by a remote verifier. TPMs can also protect a limited amount of data against a compromised operating system, e.g., hide an encryption key unless the Platform Configuration Registers (PCRs) are in a specific state. A TPM can store integrity measurements in PCRs in protected memory. Overall, security is based on two properties: (1) PCRs are accessible only via an API provided by the TPM and (2) measurements in the PCRs can only be extended, each new extension is computed using a cryptographic hash of the previous PCR value and the new measurement. The root of trust is the BIOS that performs the very first extension upon boot. Several concrete architectures have been proposed that rely on a TPM as a foundation [15], [25].

Dynamic Root of Trust. This is a new mechanism recently added to TPM specifications [38]. It has been implemented by major vendors, e.g., Intel TXT [12] and AMD SVM [1]. Dynamic root of trust provides 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. Several hardware protection mechanisms, e.g., disabling DMA or debugging and resetting the TPM PCRs, are included to prevent fraudulent attestation. Flicker [21] is an architecture that establishes dynamic root of trust on commodity computers. It takes advantage of recent advances by Intel and AMD by executing a piece of application logic (PAL) on the prover. Execution of PAL is guaranteed even if the platform's BIOS, OS and DMA are all compromised. 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 dynamic root of trust primitive. There are several other proposals that deal with establishment of trust on remote systems [14], [22]–[24], [39]. Underlying platforms range from Web servers to embedded systems.

Other Hardware-Based Techniques. A recent hardware-based mechanism for process isolation is called SPM [37]. 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 recent result is SMART [9] – 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 protection features. SMART requires small changes to the MCUs but no additional hardware. In [8] Datta et al. present a logic for secure systems, and use it to describe attestation protocols

standardized by the TCG, without providing a definition of attestation. The work in [8] relies on the presence of a secure TPM device. (In contrast, we want to describe the requirements at a lower level, without assuming the presence of such a trusted device.)

III. Remote Attestation

We use the term *Remote Attestation* to denote a protocol whereby a challenger (Chal) verifies the internal state of a device called a prover (Prov). This protocol is performed remotely, i.e., over a network, such as the Internet. The goal of a Remote Attestation protocol is to allow honest (not compromised) Prov to create an authentication token, that convinces Chal that the former is in some well-defined (expected) state. Whereas, if Prov has been compromised by the adversary (that has modified the Prov's state) the authentication token must reflect this. We will first define an attestation protocol.

Definition 1 (Attestation Protocol): An attestation protocol $\mathcal{P}$ is comprised of the following components:

- $\mathsf{Setup}(\cdot)$ – a probabilistic algorithm that, given a security parameter 1^κ, outputs a long-term key k;
- $\mathsf{Attest}(k, \cdot)$ – a deterministic algorithm that, given a key k and device state s, outputs an attestation token α;
- $\mathsf{Verify}(k, \cdot, \cdot)$ – a deterministic algorithm that, given a key k, a device state s and an attestation token α, outputs 1 iff α corresponds to s, i.e., iff $\mathsf{Attest}(k, s) = \alpha$, and outputs 0 otherwise.

At the time of attestation, Prov's state $s = (s^{\mathsf{Chal}}, s^{\mathsf{Prov}})$ consists of two parts: (1) s^{Chal} provided by Chal, e.g., a nonce, and (2) s^{Prov} that reflects the rest of Prov's state.

Next we define a game between Chal and Prov that will lead to the definition of security for attestation protocols.

Game 1 ($\mathsf{Att\text{-}Forgery}_{\mathsf{Chal,Prov}}(\kappa)$): Chal running $\mathcal{P}$ interacts with Prov as follows:

1) Chal runs $k \leftarrow \mathsf{Setup}(1^\kappa)$ and outputs s^{Chal} to Prov.
2) Prov is given oracle access to Attest. Specifically, Prov is allowed to adaptively submit q device states $\{s_1, \ldots, s_q\}$. For each $s_i \neq (s^{\mathsf{Chal}}, s^{\mathsf{Prov}})$, Prov receives the corresponding token α_i.
3) Eventually, Prov outputs α; the game outputs 1 iff $\mathsf{Verify}(k, s, \alpha) = 1$, i.e., iff α corresponds to $s = (s^{\mathsf{Chal}}, s^{\mathsf{Prov}})$.

An honest Prov can trivially create α using $\mathsf{Attest}(k, s)$. Whereas, if Prov has been compromised, its s^{Prov} has changed and it must attempt to simulate the operation of Attest. This security game bears some resemblance to a MAC-Forge game [5]. In Section VIII-A we discuss the relationship between remote attestation and MACs.

We now define our security notion, based on Game 1.

Definition 2 (Att-Forgery security): An attestation protocol $\mathcal{P} = (\mathsf{Setup}, \mathsf{Attest}, \mathsf{Verify})$ is Att-Forgery-secure if there exist a negligible function $negl$, such that, for any probabilistic polynomial time prover Prov and sufficiently large κ, it holds that: $Pr[\mathsf{Att\text{-}Forgery}_{\mathsf{Chal,Prov}}(\kappa) = 1] \leq negl(\kappa)$

To simplify our notation we say that $\mathcal{P}$ is a *secure remote attestation protocol* if $\mathcal{P}$ is Att-Forgery-secure. In Section IV, we identify the properties that Attest must have for remote attestation to be possible.

A. System Model

The central goal of any attestation protocol is to verify Prov's state. Successful execution of the attestation protocol does not guarantee that Prov's entire system can be trusted or that the adversary can not modify Prov's state after attestation is completed.

We assume that Prov is a low-end embedded device with a single thread of execution, limited storage capacity and general complexity. Although our definition of Att-Forgery-security is valid for any device, its motivation is strongest for low-cost platforms where adding secure hardware components (e.g, a TPM [38]) would be too costly. In the rest of this paper, Prov is considered to have the following characteristics:

1) Single memory space, i.e., no separation between "kernel" and "user" memory.
2) Single thread of execution, with the exception of interrupts. Note that this implies lack of Direct Memory Access (DMA). [1]
3) Ability to disable interrupts and force a region of code to execute atomically.
4) Availability of read-only memory (ROM).
5) Ability to securely cleanup (erase) memory upon device reset.
6) A hardware-based control mechanism to prevent unauthorized access to certain memory locations. (See Section V for details.)

We make no assumptions about Chal. Indeed, a malicious Chal can perform a denial-of-service (DoS) attack by forcing Prov to take part in the remote attestation protocol at will. Our security model is focused on a possibly malicious Prov and protection of Prov against DoS attacks is not a primary goal. (However, for the sake of completeness, we discuss Chal authentication, as well as replay and DoS mitigation in Section VI-A.). Also, malicious Chal does not learn any new information about an honest Prov by performing remote attestation, since Chal must already know the desired state of Prov in order to verify the attestation token. In the rest of this paper, we assume that Chal is honest. Note that s^{Chal} (the part of Prov's state sent by Chal) can contain any information

[1] If the device offers DMA, we assume that it can be securely disabled during attestation.

that Chal wants to be included in the computation of α, e.g., a nonce, a sequence number or a timestamp.

We assume a reliable communication channel between Chal and Prov. We make no assumptions about its security, latency, packet routing or any other properties.

Remark: In the research literature, the term *challenger* is often used in more theoretical (e.g., cryptographic) settings, whereas, *verifier* is the term typically encountered in attestation papers. We use them interchangeably and synonymously.

B. Adversary Model

We do not specify how Prov might be compromised; we assume that the adversary can do so at any time. Once Prov is compromised, we use the term *prover* to mean the device itself and the term *adversary* to reflect the adversary's presence on the prover. The distinction is relevant because, in order to implement remote attestation securely, there must exist some secret quantity (i.e., a key) that the adversary can not access, even though it is in full control of Prov. We discus the necessary properties for this in Section IV and practical considerations in Section V.

Once Prov is compromised, the adversary has full control over the CPU. It can schedule interrupts at will, read all readable storage (including ROM) and write to all writable storage. The only behavioral restrictions are those imposed by the hardware, e.g., the adversary can not write to ROM, or force an interrupt if interrupts are disabled. We also assume that the adversary can not perform any hardware modifications to the prover, e.g., install a different CPU or more memory. We also assume that no hardware side channels are available to the adversary, e.g., it can not measure power consumption and use it to deduce the key. Fault-based attacks (e.g., glitches on power or lines) that could lead the processor to execute instructions incorrectly, or skip some instructions, are also out of the scope of this paper. Finally, we assume that there is a mechanism to protect Attest from *software* side-channel attacks, e.g., a software-only time channel attack. Since architectures of low-end embedded devices tend to be simple (e.g., there is no cache), Attest can be made resistant to such attacks. In any case, this subject is beyond the scope of this paper.

IV. Properties Required For Remote Attestation

In this section, we describe and justify the necessary security properties of Attest. We start by assuming the existence of a secure algorithm to compute α based on prover's state s and prover-specific secret key k. Techniques that could be used for this purpose, e.g., HMAC, has been extensively studied in cryptographic literature.

According to the definition in Section III, Attest must satisfy the following security properties: (1) only Attest (using the correct key) can compute a valid token α, and (2) α accurately captures s, i.e., for any two states $s' \neq s$, $\mathsf{Attest}(k, s) = \mathsf{Attest}(k, s')$ with negligible probability. In other words, there are two ways to attack remote attestation:

Attack type 1: The adversary simulates Attest and correctly computes α.

Attack type 2: The returned α does not correctly reflect s, i.e., the adversary escapes detection.

The latter might seem unlikely. However, we believe that it can occur if execution of Attest is not *atomic*, as in our definition. We now show some attacks (of both types) along with security properties that prevent them. The end-goal is a complete set of security properties that conform to our definition in Section III.

Since k is the only secret held by Prov, access to k allows the adversary to simulate Attest, i.e., perform a type 1 attack by computing α without invoking the actual Attest. Therefore, we need the following property:

Exclusive Access: Attest has exclusive access to k.

On the other hand, exclusive access to k does not imply that the adversary can not learn some intermediate value that leaks information about k. Suppose that $\mathsf{Attest}(k, s) = \mathrm{HMAC}(k, s) = \mathrm{H}(k \oplus \mathit{opad}, \mathrm{H}(k \oplus \mathit{ipad}, s))$ and $k \oplus \mathit{ipad}$ is somehow leaked, e.g., it remains in memory after computation of α. Then, the adversary can learn k, and use it to compute α. This prompts the need for the following property:

No Leaks: Attest leaks no function of k other than α.

Another way of stating this property is that, after Attest completes, the entire state of Prov (except for α and k itself) is *statistically independent* from k. Furthermore, if Attest code is not protected, the adversary can modify it, e.g., by forcing it to output k to an unprotected memory location and then use k to compute α. For this reason, an additional property is required:

Immutability: Attest (i.e., its code) is immutable.

We stress that this security property requires code to be *executed in-place* from immutable memory. This is not always the case, e.g., when code is loaded from low-speed storage (e.g., FLASH) to high-speed memory, such as RAM or cache before execution. The adversary could modify Attest after it is loaded into RAM, but before it is executed [29]. This is an instance of the well-known time-of-check-to-time-of-use (TOCTTOU) attack, that can be prevented by a hardware signature check of the code, as in [1], [12].

The three aforementioned properties together are not sufficient to protect Attest. Consider a scenario where Attest sequentially checks a certain memory range $[a, a + n)$. Suppose the adversary resides in the range $[a + n/2, a + n)$, i.e., the 2nd half of the interval. When Attest completes its pass over the first half – $[a, a + n/2)$, the malware interrupts Attest, moves itself from $[a + n/2, a + n)$ to $[a, a + n/2)$ and restores the contents of $[a + n/2, a + n)$ to its expected state. When Attest resumes, it eventually produces a correct (valid) α, while the adversary escapes detection. This is an example of a type 2 attack.

Note that checking memory in a pseudo-random fashion, as in [33], [34], does not solve the problem, since the adversary can schedule an interrupt every time the next address is computed. Then, if the next address falls into the memory range occupied by malware, it moves its code fragment elsewhere and restores memory to its expected state. To prevent such attacks, another property we need is:

Uninterruptibility: Execution of Attest must be uninterruptible.

There is still a potential attack, despite all security properties described so far. The adversary can start execution in the middle of Attest, e.g., via return-oriented programming (ROP) [7], [17], [28]. Suppose that, in the beginning of Attest, there is an instruction to enforce uninterruptible execution environment. Then, if the adversary can start execution of Attest just after that instruction, the remainder of Attest would run in an *interruptible* manner, which leads attacks of types 1 and 2, as shown earlier.

Assuming uninterruptibility of Attest, it might seem that if the very first instruction of Attest loads the secret key k, then, even if the adversary invokes Attest in the middle, no information derived from k can be learned and a valid α can not be computed. However, this argument is incorrect, for the following reasons:

First, in some processor instruction sets (e.g, Intel x86 [13]), the adversary can jump into the middle of individual instructions, which totally changes the semantics of Attest in an unintended manner. Naturally, we prefer not to rely on features of a specific instruction set in stating general security properties. **Second**, the stated argument assumes that invoking Attest in the middle precludes the adversary from reading k. However, the adversary may be able to mount a fairly sophisticated variant of the ROP attack, as follows:

> Jump into the beginning of some function within Attest. Since the $jump$ instruction does not push a return address onto the stack, the stack will be "de-synchronized", i.e., the value specified in the stack by the adversary will become the return address of the function. When the function returns, it will jump to the address chosen by the adversary. This way, the adversary can cause Attest to jump anywhere within Attest code.

In general, the adversary can influence control flow of Attest and alter its behavior, e.g., induce Attest to leak k by reading it from restricted memory and failing to erase it later. We describe a concrete attack of this type in Section VII. To prevent all such attacks, we need one final property:

Invocation from Start: Attest must only be invoked at its very beginning.

In summary, the first three properties: *exclusive access*, *immutability* and *no leaks*, are necessary (but not sufficient) to prevent type 1 attacks. Whereas, the other two (*uninterruptibility* and *invocation from start*) together enforce atomic execution of Attest. Although they also prevent some attacks of type 1, they mainly prevent type 2 attacks.

Under the assumptions made above, we claim that any correctly implemented attestation protocol, that has all five properties listed in this section is Att-Forgery-secure.

A. Minimality of Properties

We now argue that removing any of the postulated five properties, leads to an insecure Attest. Note that each property is largely independent and eliminating any of them will make Attest vulnerable to the attack(s) described just above that property in the previous section. Specifically, if we were to omit:

- *Exclusive Access to k:* the adversary would easily learn k.
- *No Leaks:* the adversary would learn information about k that could lead to an advantage in computing a valid α.
- *Immutability:* the adversary can change the code to move k to unprotected memory.
- *Uninterruptibility:* the adversary can move malware around during attestation, which helps escape detection.
- *Invocation from Start:* the adversary can invoke Attest anywhere, which might cause it to be interruptible and/or skip sanity checks on input parameters.

It thus becomes clear that any proper subset of the five properties is insufficient for secure remote attestation. However, we do not claim that this particular set is the smallest possible set of any possible properties.

V. Deriving Features from Properties

In this section, we describe a combination of platform *features* that achieve the five security properties presented in the previous section. Our goal is a set of features that are both necessary and sufficient for remote attestation. We examine each property and identify features needed to attain it.

Exclusive Access to k. This is the most difficult property to impose on a low-end embedded device. There is no way to achieve it without some hardware support. If the underlying processor supports multiple privilege modes and a full-blown separation of memory for each process, we could use a privileged process to handle all computations that involve k. However, low-end processors generally do not offer such features.

Our solution is to add a small hardware-based check that monitors the address bus and program counter (PC) and enforces that k is only accessible when PC is within Attest. We believe that this "custom" hardware check is unavoidable.

No Leaks. To make sure that no information related to (or derived from) k is accessible when Attest completes we need a way to erase all intermediate values that depend on k, except the attestation token α, when they are no longer needed.

Immutability. In order to make Attest immutable we place it in ROM, which is available on most platforms. We consider ROM to be an inexpensive way to enforce immutability. Attest needs to execute in-place from ROM.

Uninterruptibility. On a platform with a single thread of execution, the adversary can still regain control after invoking Attest by scheduling an interrupt. To enforce uninterruptibility, we need a way to disable (and enable) interrupts such that Attest will run from beginning to end. Moreover, the instruction to disable interrupts must itself be *atomic*. Otherwise, the adversary could interrupt this instruction and violate atomicity of Attest. We show a concrete attack on *non-atomic* interrupt deactivation in Section VII.

Invocation from Start. As discussed earlier, we must enforce exclusive invocation of Attest from its very first instruction. Since there is no OS or protected CPU mode that can enforce this on low-end devices, custom hardware is needed. As before we use a small piece of custom hardware that enforces the following logic: If the program counter (PC) is an address within the Attest code, other than the first instruction address, then the previous instruction must also be within Attest.

Although this property precludes the adversary from jumping to the middle of Attest, in practice, there is no way to enforce this in an embedded system without OS support. The only option is to monitor Attest region and reset the device if illegal behavior is detected. Therefore, the ability to reset the device (in case of an error) is necessary. In addition, all sensitive memory regions must be erased immediately after the device is reset.

Features described in this section form a set that is necessary and sufficient to support the security properties described in Section IV. We summarize them as:

- Custom hardware to enforce exclusive access to k.
- Reliable and secure memory erasure.
- Read-only-memory (ROM).
- Enable-interrupts and atomic disable-interrupts instructions.
- Custom hardware to enforce Attest being invokable only at the first instruction.
- Secure reset mechanism.

VI. Attestation Protocol

We now describe a generic attestation protocol. We assume that both the verifier and the prover engage in a Att-Forgery-secure remote protocol $\mathcal{P}$ = (Setup, Attest, Verify), with all properties described in Section IV. We show that, if the prover has access to Attest, the resulting protocol can be very simple.

Figure 1 shows the protocol. It starts with the verifier challenging the prover with a fresh nonce. The nonce must be chosen uniformly at random to prevent guessing attacks.

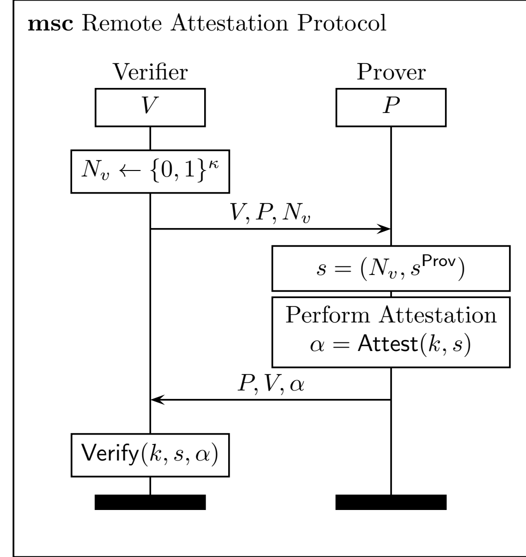

Figure 1. Remote attestation protocol. The prover and the verifier are assumed to share a secret key k beforehand.

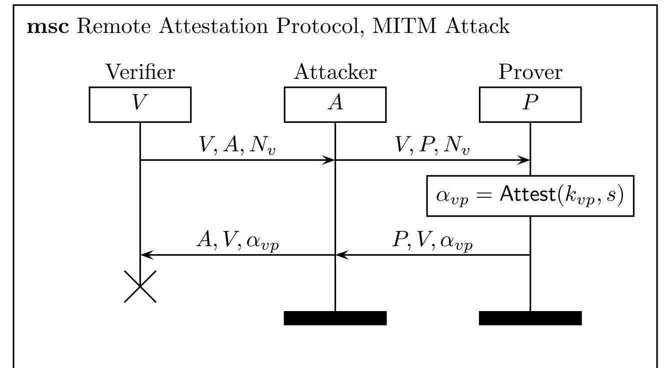

Figure 2. Failed MITM attack on the remote attestation protocol. Some details are left out for clarity.

For the same reason, it must also be chosen from a sufficiently large pool, i.e., $\kappa \geq 160$. Upon receipt of the challenge, the nonce becomes part of prover's state. The prover computes α via Attest and returns it to the verifier. The verifier runs $\mathsf{Verify}(k, s, \alpha)$ and accepts α if verification succeeds.

This protocol's main goal is to provide security guaranties to the verifier. To subvert it, the adversary must falsify α. However, since α is the output of Attest, which is secure against Att-Forgery (defined in Section III), the adversary can prevail only with negligible probability. It can not succeed in a man-in-the-middle attack either, since k used to create α is a secret shared between the prover and the verifier, as illustrated in Figure 2.

Replay attacks on the response message from the *prover* are prevented since the verifier picks a fresh nonce for each protocol instance and only accepts responses computed using the same nonce.

A. Verifier Authentication: Replay Detection and DoS Mitigation

Verifier authentication is needed to prevent unauthorized invocation of Attest (including replay attacks) on the prover. However, recall that according to our model in Section III-B, the adversary is assumed capable of compromising the prover at will. Thus, verifier authentication is, strictly speaking, beyond the scope of remote attestation. Nevertheless, we believe that it is important for overall security since attestation does not operate in a vacuum. In this section, we sketch out some means of verifier authentication, starting with verifier impersonation prevention and moving to replay and DoS attack mitigation.

Verifier Impersonation. Preventing verifier impersonation is straightforward. Since the two parties already share k, the verifier computes and includes a message authentication code (MAC) as part of the challenge. The prover verifies this MAC and proceeds to execute Attest only if the challenge is authentic. However, this does not detect replays of stale challenges and DoS attack as discussed below. Note that, if verifier authentication is needed, then the MAC computed by the verifier must be structurally distinct from that computed by the prover, in order to prevent certain well-known protocol attacks, e.g., reflection [3]. This is easily done by using direction and/or message identifiers in each party's MAC computation.

Denial Of Service (DoS). Preventing all DoS attacks on the prover is impossible. If the verifier's challenge is not authenticated, the adversary can impersonate the verifier and force the prover to run Attest using arbitrary unauthenticated challenges. Otherwise, if we require verifier's challenge to be authenticated, the adversary can force the prover to verify (re-compute) a MAC for an arbitrary challenge. It might seem that both choices are bad, since in either case the prover can be forced to compute a MAC. (Recall that MAC computation represents the dominant cost of running Attest.) However, computing a MAC over a short challenge is significantly faster then doing so over possibly very large amounts of prover's state that can include RAM and disk. Therefore, allowing the adversary to force the prover to verify MACs on arbitrary challenges is the "lesser of two evils".

Replay Mitigation. Replay attacks on the prover are applicable only if our protocol provides verifier authentication, i.e., the verifier includes MAC with the challenge and the prover verifies this MAC before executing Attest. As is well-known, replay attacks can be mitigated only if the victim (the prover) can establish not only authenticity of the challenge but also its *freshness*. This is a non-trivial requirement that leaves us with several options:

Nonce History: the verifier maintains a history of all challenge nonces. This is clearly unscalable unless a limited number of attestation protocol instances are expected.

Counters: a prover's challenge includes the protocol instance sequence number. The verifier keeps the last valid (non-wrapping and monotonically increasing) sequence number it received. This allows the verifier to detect out-of-sequence (but not necessarily delayed) challenges. The verifier can also determine whenever (and how many) challenges might have been lost.

Timestamps: a prover's challenge includes a timestamp. The verifier keeps the last valid timestamp it received. Assuming a sufficiently fine grained verifier clock, no two challenges carry the same timestamp. This allows the verifier to detect delayed[2] and out-of-sequence challenges.

The subject of replay detection has been extensively studied in the security protocols literature. As is well-known, the best protection can be obtained by the combination of sequence numbers and timestamps:

- At prover initialization (installation) time, a dummy initial challenge c_0 – that includes a sequence number s_0 and a timestamp t_0 – is stored on the prover. The prover sets $c_p = c_0$, $s_p = s_0$ and $t_p = t_0$.
- At some later time, the prover receives a challenge c_i $(i > 0)$ with s_i and t_i. It checks whether the $s_i > s_p$ and $t_i > t_p$. If not, it aborts execution. Otherwise, provided that the MAC of c_i verifies, the prover sets $c_p = c_i$, $s_p = s_i$ and $t_p = t_i$. Then, it proceeds to execute Attest.
  Remark: Optionally, as part of the last step, if the prover has its own clock, it can also check if t_p is "recent", i.e., within a certain allowable skew from its current time. This allows it to detect *delayed* challenges.

Note that the prover does not need a clock for these countermeasures. The only advantage of having one is the ability to detect (perhaps maliciously) delayed challenges.

The above measures would suffice only in the presence of an external adversary, i.e., the kind that never compromises the verifier and modifies the stored sequence number/timestamp values. We now consider a more powerful "roaming" adversary. This adversary can do everything an external adversary can, including eavesdropping. It can also compromise the prover and modify any of its internal state.

It turns out that the above countermeasures still allow a roaming adversary to mount a limited replay (DoS) attack on the prover. The attack involves three stages: (1) eavesdropping, (2) prover compromise, and (3) replay. Although this kind of adversarial behavior might seem far-fetched, we believe that it is appropriate for settings where the adversary can only compromise the prover within certain limited time interval. In more detail, the attack proceeds as follows:

(1) The adversary observes and records n authentic challenges $c_1, c_2, \ldots, c_n$ with timestamps $t_1, t_2, \ldots, t_n$ respectively.

[2] Only if the prover maintains a reliable clock of its own.

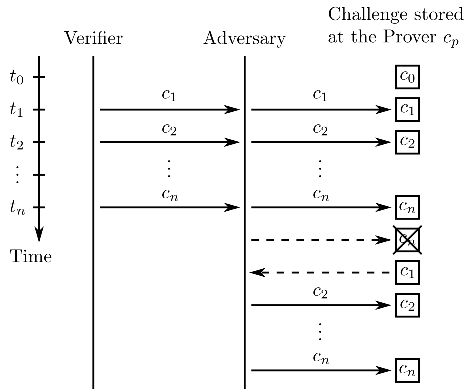

Figure 3. "Roaming" adversary. The adversary eavesdrops on a number of challenges sent by the verifier. The adversary compromises the prover and replaces the stored challenge with a previously captured one, then leaves the device and mounts a replay attack.

(2) At some time after t_n, but before the verifier issues the next challenge (c_{n+1}), the adversary compromises the prover. It changes the prover's stored challenge from c_n to c_1 and leaves the device, cleaning up after itself, i.e., the only remaining trace of adversary's former presence is c_1.

(3) At any time until the verifier issues the next challenge, the adversary can mount a series of replay attacks by using the authentic challenges $c_2, c_3, \ldots, c_n$ as shown in Figure 3.

This attack is clearly not germane to the attestation protocol which remains secure. We believe that preventing it requires additional hardware components, e.g., a secure clock or a small amount of writable secure memory (accessible only from ROM) to store the last challenge.

In conclusion, we stress that replay and DoS attacks discussed here are only relevant in very specialized scenarios where the "roaming" adversary model is appropriate. We include it here for completeness. In our main security model (which is geared purely for remote attestation) the adversary is allowed to compromise the prover at will. Thus, replay attacks are not relevant to our definition of security of remote attestation. For this reason, we omit a full treatment of all possible DoS attacks. Another reason is that is an interesting subject in its own right which deserves to be discussed separately. In fact, other variants of the roaming adversary attacks are possible and, as it turns out, easily preventable. We defer further discussion to future work.

B. *Asymmetric Cryptography*

In terms of cryptographic primitives, our discussion has been focused on symmetric techniques. One interesting question is whether there are any benefits in using public key cryptography, i.e., digital signatures.

At the first glance, digital signatures would significantly complicate Attest code in terms of both size and execution speed. (Incidentally, the latter would increase the impact of DoS attacks.) Also, instead of a shared k, the prover would need to store its private key sk (in a secure location). However, none of this prompts the need for additional security features or components. On the other hand, as far as the verifier is concerned, α produced using a MAC is no less and no more secure than a digital signature computed over the same state. (Recall that our adversary model allows prover's compromise but not hardware attacks that could extract k or sk.) We believe that the only potential advantage of digital signatures can be obtained if the application requirements of Remote Attestation include public verifiability of attestation tokens.

VII. Weaknesses in Current Designs

Among several existing proposals for remote attestation, SMART [9] is the state-of-the-art design and is the closest to the work presented in this paper. However, SMART, while sensible, lacks a systematic analysis of security properties and security features.

Using the security properties derived from our definition of secure remote attestation, we manually verified whether they hold for SMART, as described in [9]. In the process, we found three possible violations of our security properties. We believe that, if combined, these violations can lead to actual attacks on SMART. We stress that our "attacks" on SMART are on the design of the architecture, not on a particular implementation. These attacks show that it is possible to realize SMART in an insecure way, while violating none of the security assumptions of SMART.

SMART code resides in ROM; it is referred to as ROM Code (RC). SMART takes as input a memory range $[a, b]$, a challenge $nonce$, an output address out, a jump flag x_{flag}, a jump address x, and an optional input in. SMART then performs the following steps:

1) Disable interrupts.
2) Check validity of out.
3) Compute HMAC of range $[a, b]$ and write the result to address out.
4) Clean/erase all working memory.
5) Check whether x_{flag} is set. If so, jump to address x (with input in). Otherwise, enable interrupts and stop execution.

SMART includes several security checks, including the following:

- If the program counter (PC) is in RC but differs from the start of RC, then the previous instruction must also be in RC.
- If PC is outside RC, then the previous instruction must be outside RC, except for the last instruction of RC.

Finally, the result (attestation token) is placed in location out.

A. *Manipulating Input Parameters*

All parameters provided to SMART are assumed to be under adversary's control. While some of them are

```
1 [...]
2 memcopy:
3         pop r0         ; load to ptr
4         pop r1         ; load from ptr
5         pop r2         ; load len
6 loop:   load r4, r0++  ; read word at @r1
7         store r4, r1++ ; write word at @r0
8         dec r2         ; decrement len
9         bne loop       ; (branch if len > 0)
10        ret
11 [...]
12 pop r0
13 ret
14 [...]
15 jmp r0 ; the last instruction of ROM code
16        ; that will either jump to x or
17        ; return to the calling site of SMART.
```

Figure 4. Assumed contents of SMART ROM code (pseudo-code).

checked (e.g., *out*), memory address x is not. The adversary can therefore freely choose where to execute at SMART termination point. One way to abuse this feature is to call SMART by specifying x to be inside SMART code itself. By doing so, the adversary can control SMART execution. This will not trigger a fault since execution does not leave SMART code. Providing x as the last instruction of SMART itself will cause an infinite loop. Therefore, the adversary can cause a very effective denial-of-service attack without actually controlling the device.

By making some simple assumptions about the SMART implementation we can make the attack even more damaging. Assume that SMART code contains a `memcopy(*to, *from, len)` function. For simplicity, lets assume that this function takes arguments from the stack. Here is how the adversary could potentially extract the key:

> First, prepare a stack layout to perform a "borrowed code chunks" attack [17] or an "ROP programming" attack if a Turing-complete gadget set is available [7], [28]. Next, invoke SMART with x set to the entry point of the `memcopy` function.

To better illustrate exposition the attack, we assume that the RC contains the code chunks shown in Figure 4.

When RC is about to terminate, the jump instruction will execute code from the `memcopy` function. Because the jump is not pushing a return address on the stack, the stack will be "de-synchronized". Another possibility would be to find a sufficiently long sequence of `pop` instructions followed by a return instruction. The `memcopy` function will take its arguments from the adversary-controlled contents of the stack. The adversary controls `*to`,`*from` and `len`. It uses this to copy the key by providing on the stack the address of the key, the address where to copy the key and the key length. The destination will be chosen to be some adversary-controlled memory region, i.e., a region that is not erased after SMART termination. Finally, the `memcopy` function returns, fetching the return address from the adversary's controlled stack. The goal of the adversary from now on is to terminate SMART execution from the last instruction of SMART. Not doing so would lead to a violation of checks on PC and lead to a reset and full memory erase. To do so, it needs to find a chunk of code that allows him to fetch r0 from the stack (with the `pop r0/ret` chunk of code). He then returns to the jump instruction to exit ROM code. At this point, the key value is available in adversary-controlled memory.

While actual implementations of SMART might not contain the code chunks assumed above, SMART should be resistant to such an attack. A simple counter-measure is to check, at the beginning of SMART, that x does not point to the ROM code itself.

B. Control Flow Hijacking by Manipulating Interrupt Vectors

In the design of SMART [9] it is claimed that, if Attest is forced to start execution at the first – and to stop execution at the last — instruction of Attest, then disabling interrupts is redundant. While not disabling interrupts is stated as a suggestion in the SMART paper, we examine its possible consequences.

This assumption relies on the fact that, on many platforms, interrupt vectors are memory locations that contains instructions to be executed if an interrupt occurs. Those instructions usually contain a jump to the respective interrupt service routine. Therefore, to process interrupts, the processor will first move PC to the interrupt vector to execute the instruction located there. Because interrupt vectors are not part of SMART RC, this will violate the control flow checks that enforce exit from the last instruction of RC.

While this holds for some processors (e.g., AVR and ARM) it does not hold for others, where interrupt vectors contains addresses to the interrupt service routines. This is the case with MSP430 – one of the commodity processors mentioned in the SMART paper.

The adversary can change the interrupt vectors such that their entries (i.e., addresses of interrupt service routines) point to the adversary's chosen address inside RC. After that, the adversary schedules an interrupt to occur when PC is within RC. When the interrupt occurs, the processor will automatically fetch the address from the corresponding interrupt vector and jump to adversary's chosen address inside RC. This attack does not violate the two security properties stated above since the previous and next instructions are both inside RC. Essentially, whenever PC is in RC, the adversary can execute the next instruction at anywhere inside RC by carefully manipulating the interrupt vector. For example, it can schedule an interrupt so that RC skips the code that erases its working memory. Therefore, if RC does not cleans up its working memory, the adversary can learn the secret key, thus violating SMART security.

VIII. Discussion

In this section we discuss some issues that were not fully addressed earlier.

A. Comparison with MAC

Our definition of Remote Attestation functionality shares some features with the well-known and well-studied Message Authentication Code (MAC) primitive. Suppose that the legitimate prover has some secure hardware that can compute both MACs and attestation tokens. Furthermore, assume that the adversary can interrupt secure hardware execution. Whenever the verifier sends the challenge that includes a, n and $nonce$, along with expected memory contents in memory range $[a, a + n)$, the prover sends back to the verifier: the challenge, its MAC and α. Suppose that adversarial code resides in memory region $[a + n/2, a + n)$. When MAC (or Attest) finishes computation for $[a, a + n/2)$, the adversary interrupts the secure hardware, moves outside that range and restores all memory $[a + n/2, a + n)$ to original contents. In this scenario, both MAC and α are computed correctly.

- The verifier believes that MAC is computed by the genuine prover.
- The verifier can not assert absence of adversarial code in memory range $[a, a + n)$ at attestation time.

This situation illustrates that uninterruptibility is not essential for MAC computation, whereas, it is essential to the security of remote attestation.

B. Comparison with Secure Hardware

While deconstructing our definition of remote attestation into properties, and mapping them to features, we described a mixed hardware-software system. Another option would be to design a purely hardware component that computes Attest atomically. Would a design based on a single piece of secure hardware require fewer security properties?

First, we only claim minimality of security properties that are quite independent of the specific architecture, rather than minimality of security features that are architecture-specific.

Second, we need to consider what security properties must be satisfied by the secure hardware component itself. In particular, this component would still have to satisfy all five security properties described earlier.

C. Untampered Execution Environment

Attest does not automatically set up an untampered execution environment. However, it runs uninterrupted and authenticity of α guarantees absence of adversarial code in state s, at attestation time. We can take advantage of these properties to set up an untampered execution environment as follows.

Suppose that Attest disables interrupts during execution, as in Section V. First, the verifier sends the challenge that includes $nonce$, the code to be executed, and (optionally) the expected dynamic environment state, i.e., stack memory location, global configuration variables, etc. Then, prover runs Attest uninterrupted: it computes α, returns it to verifier, checks that the start address of the code is outside Attest and, if so, Attest immediately hands over control to the received code. Thus, when verifier receives a valid α, it learns that the code it sent was executed uninterrupted in an untampered execution environment. This approach is similar to SMART [9].

IX. Conclusion

This paper provided an in-depth systematic treatment of Remote Attestation and defined a new security notion for remote attestation protocols. Using this notion, we identified the necessary and sufficient properties needed for a device to support secure remote attestation. We then mapped these properties into a minimal collection of hardware and software components that collectively yield a secure attestation primitive. We also presented a protocol that uses the this primitive to achieve secure remote attestation, over a network, such as the Internet. We showed that such protocols can be made both simple and efficient. Finally, we used the set of properties derived from our security notion to analyze a recent remote attestation proposal SMART [9] and identified some surprising vulnerabilities.

This work represents the first step towards a systematic study of Remote Attestation. There remain some important issues and questions for future work. Although we argued that the identified properties and derived components that collectively represent a minimal architecture for Remote Attestation, there could well be other sets of components that also achieve minimality. We plan to further investigate this and implement the proposed architecture on several commodity platforms, possibility using public key digital signature as an alternative to symmetric MAC constructs. Finally, our future work will include the development of methods for automated verification of such properties on actual implementations.

References

[1] Advanced Micro Devices. AMD, Secure Virtual Machine Architecture Reference Manual. Publication No. 33047, Revision 3.01, May 2005.

[2] Alteon Networks, Inc. Gigabit ethernet/pci network interface card. host/nic software interface definition, June 1999. Revision 12.3.11.

[3] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2008.

[4] W. A. Arbaugh, D. J. Farbert, and J. M. Smith. A secure and reliable bootstrap architecture. *IEEE Symposium on Security and Privacy (IEEE S&P)*, 1997.

[5] M. Bellare, J. Kilian, and P. Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System*, 839:1–36, 2000.

[6] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*, 2009.

[7] S. Checkoway, L. Davi, A. Dmitrienko, and A.-r. Sadeghi. Return-Oriented Programming without Returns. *Proceedings of the 17th ACM conference on Computer and communications security (CCS)*, 2010.

[8] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *30th IEEE Symposium on Security and Privacy (IEEE S&P)*. Carnegie Mellon University, 2009.

[9] K. E. Defrawy, A. Francillon, D. Perito, and G. Tsudik. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *The Network & Distributed System Security Conference*, 2012.

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

[11] R. Gardner, S. Garera, and A. Rubin. On the difficulty of validating voting machine software with software. In *Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, 2007.

[12] Intel Corporation. *Intel Trusted Execution Technology (Intel TXT) – Software Development Guide*, December 2009. Document Number: 315168-006.

[13] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, March 2012.

[14] R. Kennell and L. H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[15] 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. *IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009.

[16] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New Results for Timing-Based Attestation. In *IEEE Symposium on Security and Privacy*. The MITRE Corporation Bedford, MA, USA, 2011.

[17] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. Technical report, suse, September 2005.

[18] Y. Li, J. M. McCune, and A. Perrig. SBAP: Software-Based Attestation for Peripherals. *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (TRUST)*, 2010.

[19] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the Integrity of PERipherals Firmware. In *Proceedings of the 18th ACM conference on Computer and communications security (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 Symposium on Security and Privacy (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 *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 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. *Technology*, 42(2), 2008.

[23] C. Nie. Dynamic root of trust in trusted computing. *TKK T1105290 Seminar on Network Security*, 2007.

[24] B. J. Parno, J. M. McCune, and A. Perrig. Bootstrapping Trust in Commodity Computers. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2010.

[25] S. Pearson, M. C. Mont, and S. Crane. Persistent and Dynamic Trust: Analysis and the Related Impact of Trusted Platforms. *Security*, 3477, 2005.

[26] D. Perito and G. Tsudik. Secure Code Update for Embedded Devices via Proofs of Secure Erasure. In *Proceedings of the 15th European conference on Research in computer security (ESORICS)*, 2010.

[27] A. Perrig and L. v. Doorn. Refutation of "On the Difculty of Software-Based Attestation of Embedded Devices", 2009.

[28] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *Manuscript*, V, 2009.

[29] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium*, 2004.

[30] A. Seshadri, M. Luk, and A. Perrig. SAKE: Software attestation for key establishment in sensor networks. *Ad Hoc Networks*, 9(6), 2008.

[31] A. Seshadri, M. Luk, A. Perrig, L. V. Doorn, and P. Khosla. SCUBA: Secure Code Update By Attestation in sensor networks. In *Proceedings of the 5th ACM workshop on Wireless security (WiSec)*, 2006.

[32] 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 December, DTIC Document, 2004.

[33] 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. *ACM SIGOPS Operating Systems Review*, 39(5), 2005.

[34] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. SWATT: software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy(IEEE S&P)*, 2004.

[35] U. Shankar, M. Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. *13th USENIX Security Symposium*, 8(3), May 2004.

[36] S. W. Smith. Outbound Authentication for Programmable Secure Coprocessors. *International Journal of Information Security*, 3(1), 2004.

[37] R. Strackx, F. Piessens, and B. Preneel. Efficient Isolation of Trusted Subsystems in Embedded Systems. *Security and Privacy in Communication Networks*, 2010.

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

[39] Q. Yan, J. Han, Y. Li, and R. Deng. A software-based root-of-trust primitive on multicore platforms. In *Proceedings of 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

Appendix A. Issues with Time-Based Attestation

As shown in Section II, time-based attestation techniques are not applicable for attestation performed over a network. Moreover, some assumptions that form the basis for time-based attestation have been challenged [6], [16], [35]. In this section, we underscore the difficulty of *secure* time-based attestation by discussing some new issues in time-based attestation.

A. Partial Precomputation in VIPER

The central security assumption in time-based attestation is:

> "The code design of the checksum algorithm must require precisely the *smallest* number of cycles to complete" [19].

We show how this security assumption does not hold in VIPER [19].

In time-based attestation, a verifier sends a challenge to the prover, which normally includes a nonce. It is used as part of input to the checksum function to prevent pre-computation or replay attacks. However, this does not prevent the adversary from executing a part of the checksum code in advance. An adversary could reduce the time for computing a checksum during the challenge phase. Our attack uses this observation, which we call *partial precomputation*. The code in Figure 5 illustrates a fragment of VIPER checksum code, borrowed from Figure 10 of [19].

We can see that these 9 instructions are independent of $nonce$, which allows the adversary to pre-execute them and reduce the run-time of the checksum function during the challenge phase. This clearly violates the main security assumption. On the other hand, the checksum function iterates 300 times. Whereas, pre-computation is only applicable to the first iteration. In practice, this limited "real estate" is likely insufficient to execute any effective malware. However, it shows that a checksum function must ideally not allow any pre-computation.

Note that the above is different from a parallel computation attack [39] which works only with multi-core processors. Our pre-computation works for a single-core system – the target platform for VIPER.

Defense. To prevent partial precompution, we recommend the following:

- Careful attention paid to how $nonce$ is used. The above would not work if every instruction depended on $nonce$.
- Clear distinction between (and enumeration of) parameters that are in known-good state and parameters that are randomized.

B. Hardware Assumptions in VIPER

VIPER was implemented for the PCI bus setting, where devices might collude to cheat on attestation requests. Therefore, to attest a device, all faster devices on the same system need to be attested. Quoting from [19]:

> "[...] The solution for verifying a device with a particular level of computational capability is that all devices with greater capabilities must be verified first. [...] After the attestation of a faster peripheral, the verification function on the faster peripheral continues running until all peripherals have been verified."

In other words, because an attested device continues running until all devices are attested, it is running trusted code. Therefore, it cannot be used as a proxy to help another device. This assumption requires that only the CPU can stop the device once all devices are attested

However, if a slower device can interrupt a faster device that *continues running after attestation*, then this would violate the fundamental security property. The GA620 network card, on which VIPER was developed, is based on the MIPS-based TIGON 2 chip [2]. This device has two cores that are attested one after the other. According to its documentation a *CPU State* register per core is exported to the PCI configuration space. The devices can be interrupted and controlled using those registers:

> "Alteon Networks uses these bits a great deal to implement a firmware debugger. In general, a driver writer should not need to adjust these bits directly. On the Tigon 2 ASIC there are two CPUs (CPU A and CPU B). There is a CPU State Register for each CPU" [2].

With peer-to-peer DMA, it would therefore be possible for a slower device to interrupt a faster device and use it as a proxy, which violates VIPER design assumptions.

Finally, the above demonstrates that, while software-based attestation is attractive for legacy embedded systems, there are many abstruse hardware features that needs to be considered. Only hardware mechanisms designed for attestation can

```
1 xor r31,r4,r1          ;addr=memory_base^offset
2 lw r1,0(r31)           ;memory read
3 xor r1,r5,r1           ;tmp1=r5^mem[addr]
4 add r31,r31,r          ;tmp2=addr+tmp1
5 andi r1,r31,0x1ffc     ;offset=tmp2&mask
6 xor r1,r3,r1           ;addr=memory_base^offset
7 lw r1,0(r1)            ;memory read
8 xor r1,r7,r1           ;tmp3=r7^mem[addr]
9 add r1,r3,r1           ;tmp3=PC+tmp3
```

Figure 5. VIPER code.

guaranty minimal required security properties with a high level of confidence.

C. Reduced Program Size in Pioneer

Security of Pioneer [33] relies not only on the speed of the checksum function (assumed to be optimal) but also the minimal size of the code for that function. If the adversary can implement the same checksum within smaller binary code, a *memory copy attack* might be possible, using x86 CPU segment registers [33].

We now show how to design an equivalent, yet smaller-sized, checksum. The following code fragment is taken from the checksum implementation in Figure 11 of [33]:

```
or 0x5,rax ;binary code 0x4883c805
```

Note that only the lower 8 bits of register rax are modified. This can be rewritten as:

```
or 0x5,al ;binary code 0x0c05
```

Note that *"or 0x5, rax"* translates into 4 bytes of binary code, while *"or 0x5, al"* – to only 2 bytes. This already shows a reduction in program size of 2 bytes. Similarly, we can reduce instruction size for the following code:

```
and 0x1,rdi
add rdi,rdx
add rdi,rdi ;binary code 0x4801ff
```

After the first instruction executes, only the least significant bit (LSB) of rdi is preserved. Therefore , in the third instruction: *"add rdi, rdi"*, we can use an equivalent instruction: *"add edi, edi"* which is only 2 bytes in binary code *"0x01ff"*.

These examples illustrate two ways to reduce checksum code size in Pioneer. Such optimizations could be collectively used to implement a memory copy attack.

D. Time-Based Attestation in Multicore Systems

Time-based attestation approaches have been primarily developed for single-core processor systems. More recently, there has been a proposal for multicore processor systems [39]. This scheme uses time-based attestation for single-core processor (e.g. Pioneer) as a building block. The main argument is the following:

> "[...] reduce the time-optimal problem on multicore platforms to the time-optimal problem on each computing unit, which is the same as the problem that has been solved in the time-based schemes for uniprocessor platforms."

We consider this argument to be problematic. The reason is that secure time-based attestation schemes are not available for all single-core processors. Schemes have been proposed for specific CPUs, such as Intel Pentium IV Xeon – the Pioneer's checksum implementation is designed to be optimal only for that processor. Pioneer checksum code might not be optimal when used on other processors. Each single-core processor model might share its instruction set with other models, might have different performance characteristics due to different type of cache, pipelining implementation and branch prediction algorithms. Therefore, the assumption that Pioneer code is optimal is not applicable for a single computing unit in a multi-core processor system.

The fact that time-based attestation is not portable due to the dependency of security of time-based attestation and specific processor models has been already observed by others, including Pioneer [33], SWATT [27], [34], and [11]. However, this issue has not been considered in the proposal for multicore platforms [39].