# A Minimalist Approach to Remote Attestation

***Abstract*—Embedded computing devices 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 an increasingly popular and attractive target for attacks, especially, malware infections. A number of research proposals have been made 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 for) architectural minimality, which is rarely encountered in security research. This work also provides a promising platform for attaining more advanced security services and guarantees.**

## I. Introduction

"Embedded systems" is a broad notion that encompasses many kinds of specialized computing devices that 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 [9] incident pointedly demonstrated the impressive scope and impact of malware on embedded devices. Stuxnet specifically targeted Programmable Logic Controllers (PLC) in industrial control systems. By modifying PLC control parameters, it ostensibly caused some serious physical damage.

Stuxnet should be viewed as both an example, a warning and a preview of coming attractions. It epitomizes the power and amplification factor of *remote* attacks, i.e., those not requiring direct access to victim devices. With growing presence and proliferation of networked embedded devices into many spheres of life, remote software attacks have become a clear and present danger. This motivates the need 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 **Remote Attestation** – a security service that involves verification of internal state of a remote embedded device. Although Remote Attestation is not a panacea, it should ideally allow for efficient and accurate detection of remote software 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.

We use the term *Remote Attestation* to denote *attestation performed across a network*. In this setting, software attestation techniques [13], [16], [17], [25]–[29] are not applicable since software attestation is secure only if the verifier communicates directly to the prover, with no intermediate hops [5]. At the other end of the solution spectrum are techniques based on secure hardware components, such as TPMs [33] or secure co-processors [31]. However, they represent a significant cost barrier for low-end embedded devices. We believe that the promise lies in a careful analysis of Remote Attestation as a distinct security service, including systematic identification of its necessary and sufficient components. This should ideally 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 exact properties needed to attain it. We then translate them into architectural components, which we then map into a small set of hardware features that collectively achieve all 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 fewest 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.

## II. Related Work

**Software-Based Attestation.** One early example of software-based attestation is Pioneer [28]. which does not rely on a secure co-processor or any specialized hardware. Pioneer 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 sufficiently long to detect cheating. Attestation that relies on time-based checksums has also been adapted to embedded devices in [13], [16], [17], [25]–[27], [29]. However, some assumptions that form the basis for these solutions have been challenged [30] and several attacks on these (and similar) schemes have been proposed [5]. Moreover, Kovah et al. [14] showed that time-based attestation schemes may be vulnerable to Time Of Check, Time Of Use (TOCTOU) attack.

In general, all current software-only solutions rely on strong assumptions on adversarial capabilities 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.

**Static Root of Trust.** An early example of a hardware-based mechanism is Secure Boot [3] which verifies system integrity at boot time. The *root of trust* is an immutable bootloader that is stored in ROM together with a public key. It verifies code signature and will execute it (i.e., boot) only when the signature is correct, ensuring the origin of the code.

Trusted Platform Modules (TPMs) [33] are now present in many commercial systems and are used in several concrete architectures [13], [22]. TPM security is based on two properties: (1) Platform Configuration Registers (PCRs) accessible only via a fixed API, and (2) PCRs that are reset only on boot, and each new measurement added (extended) to a PCR is added using a cryptographic hash of the previous PCR value and the new measurement. A TPM can sign a set PCRs that represent the load time state of the software that was executed on the computer. TPM is a static mechanism where the root of trust is the BIOS that performs the very first extension upon boot.

**Dynamic Root of Trust.** A dynamic root of trust provides a mechanism that can be used to perform an attestation *dynamically*, i.e., on the current state of the software. This requires additional features present in extended trusted computing specifications [33] as well as by CPU and chipset support by major vendors (e.g., Intel TXT [10] and AMD SVM [2]).

Flicker [18] is an architecture that establishes dynamic root of trust on commodity computers. It uses CPU extensions to execute 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.

There are several techniques for remote trust establishment [12], [19]–[21], [34] with Underlying platforms range from Web servers to embedded systems.

**Other Hardware-Based Techniques.** SPM [32] is a recent hardware-based mechanism for process isolation that uses a special *vault* module bootstrapped from a static root of trust. The vault bootstraps the SPM-protected programs, which gain exclusive control over the protection of their own memory pages. SMART [8] is 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 with a limited hardware impact. Datta et al. [7] proposed LS2, a logic for secure systems relying on a TPM. They use this to describe attestation protocols standardized by the TCG, without providing a definition of attestation. In contrast, we aim at describing the requirements at a lower level, without assuming the presence of a TPM or a similar device. This is because this would incur an unacceptable increase of complexity or cost of the low end embedded device.

## III. Remote Attestation

We use the term *Remote Attestation* to denote a protocol, whereby a challenger (Chal) verifies the internal state of another device called a prover (Prov). This protocol is performed remotely, i.e., over a network. The goal of the protocol is to allow an 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 and its state has been modified, the authentication token must reflect this. We begin by defining the "remote attestation protocol".

*Definition 1 (Remote Attestation Protocol):* A protocol $\mathcal{P}$ comprised of the following components:

- $\mathsf{Setup}(1^\kappa)$ – a probabilistic algorithm that, given a security parameter $1^\kappa$, outputs a long-term key $k$;
- $\mathsf{Attest}(k, s)$ – a deterministic algorithm that, given a key $k$ and device state $s$, outputs an attestation token $\alpha$;
- $\mathsf{Verify}(k, s, \alpha)$ – a deterministic algorithm that, given a key $k$, a device state $s$ and an attestation token $\alpha$, outputs 1 iff $\alpha$ 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) $s_{\mathsf{Chal}}$ provided by Chal, e.g., a nonce, and (2) $s_{\mathsf{Prov}}$ 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 remote attestation protocols.

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

1) Chal runs $k \leftarrow \mathsf{Setup}(1^\kappa)$ and outputs $s_{\mathsf{Chal}}$ 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 $\alpha_i$.
3) Eventually, Prov outputs $\alpha$; the game outputs 1 iff $\mathsf{Verify}(k, s, \alpha) = 1$, i.e., iff $\alpha$ corresponds to $s = (s_{\mathsf{Chal}}, s_{\mathsf{Prov}})$.

An honest Prov can trivially create $\alpha$ using $\mathsf{Attest}(k, s)$. Whereas, if Prov has been compromised, its $s_{\mathsf{Prov}}$ has changed and it must attempt to simulate the operation of Attest. This security game bears some resemblance to a MAC-Forge game [4]. Section VI-A discusses the relationship between remote attestation and MACs.

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

*Definition 2 (*Att-Forgery *security):* A remote 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 $\kappa$, it holds that: $Pr[\mathsf{Att\text{-}Forgery}_{\mathsf{Chal},\mathsf{Prov}}(\kappa) = 1] \leq negl(\kappa)$

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 remote attestation protocol is to verify Prov's state. Successful protocol execution 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 a low 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 [33]) would be too costly.

We make no assumptions about Chal. In particular, 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. In the rest of this paper, we assume that Chal is honest. Note that $s_{\mathsf{Chal}}$ (the part of Prov's state sent by Chal) can contain any information that Chal wants to be included in the computation of $\alpha$, 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.

### 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 (almost) 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., tamper with the digital logic, install a different CPU or add more memory.

Hardware side channels (e.g., measuring power consumption to infer bits of the key) or fault attacks (e.g., power glitches leading to incorrect execution of instructions) are very effective attacks on embedded systems. However, they are often dealt with by dedicated countermeasures (e.g., glitch sensors, dual rail logic...), and are therefore out of the scope of this paper. Similarly, we assume that Attest is free of software vulnerabilities, implementation flaws and *software* side-channels (e.g., a software-only time channel attack).

While those attacks are important to consider in a secure embedded system, they are beyond the scope of the current paper. Indeed, we first aim at getting the design of remote attestation right before implementing a complete hardened system.

## IV. Properties Required For Remote Attestation

In this section, we describe the necessary security properties that Attest must satisfy in order to be used for secure attestation.

As follows from definition 2, Attest must satisfy the following security properties: (1) only Attest can compute a valid token $\alpha$, and (2) $\alpha$ accurately captures the device state $s$, i.e., for any two states $s' \neq s$, $\mathsf{Attest}(k, s) = \mathsf{Attest}(k, s')$ with negligible probability. These observations leads us the following two attacks:

- **Attack type 1:** The adversary simulates Attest and correctly computes $\alpha$.
- **Attack type 2:** Returned $\alpha$ does not correctly reflect $s$, i.e., the adversary escapes detection.

We now derive the complete set of security properties that a function Attest must have in order to be Att-Forgery secure. Since the key $k$ 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 $\alpha$ without invoking the actual Attest. Therefore, we need the following property:

- **Exclusive Access:** Attest has exclusive access to $k$.

Exclusive access to $k$ does not imply that the adversary can not learn some intermediate value that leaks information about $k$. Suppose that $\mathsf{Attest}(k, s) = \mathrm{HMAC}(k, s) = \mathrm{H}(k \oplus opad, \mathrm{H}(k \oplus ipad, s))$ and $k \oplus ipad$ is somehow leaked, e.g., remains in memory after computation of $\alpha$. This will allow the adversary to learn $k$, and use it to compute $\alpha$. Therefore we need the following property:

- **No Leaks:** Attest leaks no function of $k$ other than $\alpha$.

Another way of stating this property is that, after Attest completes, the entire state of Prov (except for $\alpha$ and $k$ itself) is *statistically independent* from $k$.

If the code that makes up Attest is not protected, the adversary can modify it, for example by forcing it to output $k$ (i.e., violate the "exclusive access" property). 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 [24]. 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 [10].

Taken together, the three aforementioned properties are insufficient to protect Attest. Consider the case where a memory region is attested sequentially. When attestation reach the region containing malicious code, attestation is interrupted, malicious code moved to an already verified memory location and the original memory is restored before resuming attestation.

This allows the adversary to escape detection. This is an example of a type 2 attack.

Note that checking memory in a pseudo-random fashion, as in [28], [29], 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, we need another property:

- **Uninterruptibility:** Execution of Attest must be uninterruptible.

There remains a potential attack, despite all security properties described so far: the adversary can start execution in the middle of Attest skipping important parts of the code. 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 $\alpha$ can not be computed. However, this argument is incorrect, for the following reasons:

**First**, in some instruction sets (e.g, Intel x86 [11]), skipping the first byte(s) of an instruction can lead to decoding a different instruction. Therefore, the adversary can jump into carefully selected locations in Attest changing its semantics 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 perform a return oriented programming (ROP) [6], [15], [23] 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, if the adversary can influence control flow of Attest and alter its behavior it can induce Attest to leak $k$ by reading it from the restricted memory or avoid to erase it later. To prevent all such attacks, we need one final property:

- **Controlled Invocation:** Attest must only be invoked from an intended entry point[1].

[1]Although Control-Flow Integrity (CFI) [1] is a desirable property of Attest, it must not be confused with **Controlled Invocation** which is still required because the attacker is able to load its own code in the device which would allow him to skip some instructions of Attest despite the presence of CFI. We consider CFI to be part of the code correctness assumptions of Attest.

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 *controlled invocation*) together enforce a semantically 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 $\alpha$.
- *Immutability:* the adversary could change the code to move $k$ to unprotected memory.
- *Uninterruptibility:* the adversary could move malware around during attestation, which helps escape detection.
- *Controlled Invocation:* the adversary could 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.

## V. Deriving Features from Properties

In this section, we describe a combination of platform *features* that achieve the five security properties presented above. Our goal is to obtain 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 $k$ only being 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 $\alpha$, 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[2]. 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.

**Controlled Invocation.** 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.

**Secure Reset.** Although **Controlled Invocation** precludes the adversary from jumping to the middle of Attest, in practice, there is no way to prevent this from happening in an embedded system. The only option is to reset the device if this property, or any other one, is violated. Such a reset must be triggered in hardware and, to prevent any sensitive memory contents to be leaked, all memory 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.

### A. 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, $\alpha$ 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.

[2]Choices other than ROM may be possible as well (e.g., a latched EEPROM), but they would likely incur more complexity and thus detract from our goal of designing a minimalist mechanism.

## VI. Discussion

In this section we discuss some issues that were not adequately 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 a nonce, along with expected memory contents in memory range $[a, a+n)$, the prover sends back to the verifier: the challenge, its MAC and $\alpha$. 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 $\alpha$ 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 $\alpha$ 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 the 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 $\alpha$, 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 $\alpha$, it learns that the code it sent was executed uninterrupted in an untampered execution environment. This approach is similar to SMART [8].

## VII. 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.

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] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM TISSEC*, 13(1):4:1–4:40, Nov. 2009.

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

[3] W. A. Arbaugh, D. J. Farbert, and J. M. Smith. A secure and reliable bootstrap architecture. *IEEE S&P*, 1997.

[4] 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.

[5] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. *ACM CCS*, 2009.

[6] S. Checkoway, L. Davi, A. Dmitrienko, and A.-r. Sadeghi. Return-Oriented Programming without Returns. *ACM CCS*, 2010.

[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*, October 2010.

[10] Intel Corporation. *Intel Trusted Execution Technology (Intel TXT) – Software Development Guide*, 2009. doc: 315168-006.

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

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

[13] 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 DSN*, 2009.

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

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

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

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

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

[19] 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.

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

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

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

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

[24] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium*, 2004.

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

[26] 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.

[27] 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 2004, DTIC Document.

[28] 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 OSRw*, 39(5), 2005.

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

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

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

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

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

[34] Q. Yan, J. Han, Y. Li, and R. Deng. A software-based root-of-trust primitive on multicore platforms. In *ACM ASIACCS*, 2011.