

# Remote Attestation on Legacy Operating Systems With Trusted Platform Modules<sup>1</sup>

Dries Schellekens Brecht Wyseur Bart Preneel

*Katholieke Universiteit Leuven  
Department ESAT/SCD-COSIC  
Kasteelpark Arenberg 10  
B-3001 Heverlee, Belgium*

---

## Abstract

A lot of progress has been made to secure network communication, e.g., through the use of cryptographic algorithms. However, this offers only a partial solution as long as the communicating end points still suffer from security problems. A number of applications require remote verification of software executing on an untrusted platform. Trusted computing solutions propose to solve this problem through software and hardware changes, typically a secure operating system and the addition of a secure coprocessor respectively. On the other hand, timed execution of code checksum calculations aims for a solution on legacy platforms, but can not provide strong security assurance. We present a mixed solution by using the trusted computing hardware, namely the time stamping functionality of the trusted platform module, in combination with a timing based remote code integrity verification mechanism. In this way, we do not require a secure operating system, but at the same time the overall security of the timed execution scheme can be improved.

*Keywords:* remote software authentication, attestation, trusted platform module, timed execution

---

## 1 Introduction

In the past decades, most software designers assumed that their software is not a target of tampering and fraud. Or that, even in the case of tampering, this would only be limited to some individual cases, without any harmful impact to the business model of the software vendor. However, today's software is becoming more and more mobile, and their tasks become increasingly critical. For instance, banking applications become a commodity, in online gaming real money can be earned or lost (e.g., World of Warcraft, Second Life, online casino games).

---

<sup>1</sup> This work was supported in part by the Concerted Research Action (GOA) Ambiorics 2005/11 of the Flemish Government, by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), in part by the European Commission through the IST Programme under Contract IST-027635 OPEN-TC and IST-021186 RE-TRUST, and in part by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

For all these applications, it is clear that only legitimate, untampered client applications should be granted access to a service. Hence an authorized entity wants to be able to verify if client software is running untampered on a remote untrusted platform. If tampering is detected, this verifier will want to disconnect the client from the network, stop the service to this particular client, or even force that client application to halt its execution.

A verification entity is able to assure execution of software, using attestations (proofs) sent from the untrusted execution platform. We define the problem of *remote code integrity verification* as the act of delivering such attestations to a verification entity that guarantees code executes untampered on a remote untrusted computing platform. On such a platform, an adversary has administrative privileges and can tamper with all the software on the untrusted platform including the operating system. Remote code integrity verification can be seen as an extension of local integrity verification, in which the software execution fails when tampering of its code is detected; commonly this is referred to as *tamper resistant software* [1]. However, it is difficult to do a secure *tamper response* [22]. In the case of remote verification, it is sufficient that tampering is detected.

So far, establishing a trusted execution environment on an untrusted platform has been an open research challenge. An adversary having complete control over an untrusted platform, implies that he also has control over its input and output traffic. This makes it difficult for a verifier to be assured of communicating with a particular environment on a untrusted platform. Even more: to be guaranteed software is actually running in that environment. For example, how can we detect if the software is running directly on the OS of the platform? Techniques like simulation, emulation, virtualization, or misdirection, are available to an adversary.

### 1.1 Related work

These issues were addressed by Kennell *et al.* [11] who developed so called genuinity tests, to verify if the hardware is real, and certain software is actually running. These tests leverage detailed knowledge about the processor of the untrusted platform, and are slow to execute on other processors or to simulate in virtual machines. In practice however, the proposed solution turns out to be not sufficient [20].

Other propositions try to verify computations performed on the untrusted host, e.g., by embedding trace gathering code in the original program and locally cross checking the trace [12] or by verifying certain assertions.

Eventually, when attestation systems are unable to guarantee reliable execution of software, one can move critical code away from untrusted platforms. Techniques such as program slicing split software into non-critical and critical code slices. Only the non-critical code is run on the untrusted platform, guaranteeing that the critical slices can not be tampered [4,5,25]. This is an example of server side execution.

A completely different approach is to introduce hardware, tailored specifically to provide assurance on an untrusted platform. Using a trusted platform module, a trusted computing platform can be created. This offers good security, but on the downside, the operating systems needs to be adapted heavily, and there are

deployment issues.

The opposite angle is a pure software approach. Pioneer [17,18] is one of these systems that tries to establish this. It works on legacy systems, but it relies on some (unrealistic) strong assumptions.

## 1.2 Outline of paper

In this paper, we address the problem of remote code integrity verification based on the latter two approaches. Since recent, a lot of TPM enabled computers are sold on the market. Therefore, we want to use them to address the problems of the pure software solutions, without the deployment of heavily adapted operating systems. In section 2, we focus on trusted computing platform based attestation. The opposite angle, purely software attestation techniques, is discussed in section 3. The new mixed attestation technique is presented in section 4. Section 5 concludes our results and presents future work.

# 2 Remote attestation on trusted computing platforms

Trusted computing initiatives propose to solve some of today's security problems of the underlying computing platforms through hardware and software changes. The two main initiatives for a new generation of computing platforms are the *Trusted Computing Group* (TCG) [2], a consortium of most major IT companies, and Microsoft's *Next-Generation Secure Computing Base* (NGSCB) [6,13]. We will solely focus on TCG technology, as these specifications are public and TCG enabled computers are commercially available.

## 2.1 TCG overview

The TCG sees itself as a standard body only. Neither does it provide any infrastructure to fully utilize the technology, nor does it perform certification of any kind. The TCG specifications define three components that form a Trusted Platform<sup>2</sup>.

The core is called the *Trusted Platform Module* (TPM) which usually is implemented by a smartcard-like chip bound to the platform.

The second component is called *Core Root of Trust for Measurement* (CRTM), and is the first code the TCG compliant platform executes when it is booted. In a personal computer, the CRTM is the first part of the BIOS (Basic I/O System), which can not be flashed or otherwise be modified.

To compensate for the lack of functionality in the TPM, the TCG specifies a *TCG Software Stack* (TSS), which facilitates some of the complex, but non-critical functionality and provides standard interfaces for high level applications.

---

<sup>2</sup> All TCG specifications are available on <https://www.trustedcomputinggroup.org>.

### 2.1.1 Trusted Platform Module

The TPM is the main component of a TCG platform and offers a physical true random number generator, cryptographic functions (i.e., SHA-1, HMAC, RSA encryption/decryption, signatures and key generation), and tamper resistant non-volatile memory (mainly used for persistent key storage). Remark that no symmetric encryption algorithm is provided.

The TPM offers a set of *Platform Configuration Registers* (PCRs) that are used to store measurements (i.e., hash values) about the platform configuration. The content of these registers can only be modified using the *extending* operation<sup>3</sup>:  $PCR_{new} \leftarrow SHA-1(PCR_{old} || M)$  with  $PCR_{old}$  the previous register value,  $PCR_{new}$  the new value,  $M$  a new measurement and  $||$  denoting the concatenation of values.

### 2.1.2 Integrity measurement

The initial platform state is measured by computing cryptographic hashes of all software components loaded during the boot process. The task of the CRTM is to measure (i.e., compute a hash of) the code and parameters of the BIOS and extend the first PCR register with this measurement. Next, the BIOS will measure the binary image of the bootloader before transferring control to the bootloader, which in its turn measures the operating system<sup>4</sup>. In this way a *chain of trust* is established from the CRTM to the operating system and potentially even to individual applications.

### 2.1.3 Integrity reporting

The TCG *attestation* allows to report the current platform configuration ( $PCR_0, \dots, PCR_n$ ) to a remote party. It is a challenge-response protocol, where the platform configuration and an anti-replay challenge provided by the remote party are digitally signed with an *Attestation Identity Key* (AIK). If needed, a *Stored Measurement Log* (SML), describing the measurements that lead to a particular PCR value, can be reported as well. A trusted third party called *Privacy Certification Authority* (Privacy CA) is used to certify the AIKs. Version 1.2 of the TCG specification defines a cryptographic protocol called *Direct Anonymous Attestation* (DAA) [3] to eliminate the need for a Privacy CA, as it can potentially link different AIKs of the same TPM.

## 2.2 Application level attestation

TCG attestation is designed to provide remote verification of the complete platform configuration, i.e., all software loaded since startup of the platform. Establishing a chain of trust to individual programs is not straightforward.

<sup>3</sup> The version 1.2 specification contains the notion of dynamic root of trust for measurement, where a number of PCRs can be reset by higher privileged (determined by locality) code.

<sup>4</sup> TrustedGRUB ([https://prosec.trust.rub.de/trusted\\_grub.html](https://prosec.trust.rub.de/trusted_grub.html)) is an example of an open source bootloader that is enhanced to measure the operating kernel system. The OSLO bootloader [10], on the other hand, uses the AMD SKINIT instruction to create a dynamic root of trust for measurement; this has the advantage that the – potentially untrusted – BIOS is not included in the chain of trust.

### 2.2.1 Operating system requirements

The operating system needs to measure the integrity of all privileged code it loads (i.e., kernel modules), because it can be used to subvert the integrity of the kernel; traditionally loadable kernel modules are used to inject kernel backdoors. However, legacy operating system are monolithic, too big and too complex to provide a sufficiently small *trusted computing base* and hence they are prone to security vulnerabilities. As legacy operating system can not guarantee a chain of trust beyond the bootloader, trusted computing initiatives opt for a microkernel or hypervisor in combination with virtualization to achieve both security and backward compatibility.

### 2.2.2 Load-time binary attestation

A first approach to attest individual program is to directly apply the TCG (i.e., load-time binary) attestation on all userland components [16]. On the creation of user level processes, the kernel measures the executable code loaded into the process (i.e., the original executable and shared libraries) and this code can measure subsequent security sensitive input its loads (e.g., arguments, configuration files, shell scripts). All these measurements are stored in some PCR register and the Stored Measurement Log.

In its basic form TCG attestation has some shortcomings. First, a huge number of possible configurations exist, because every new version of a component will have a different binary and hence produces a different hash value.

Lastly, load-time attestation provides no run-time assurance as there can be a big time difference between integrity measurement (i.e., startup) and integrity reporting. The platform could be compromised since it has been booted.

### 2.2.3 Hybrid attestation schemes

To overcoming some of the shortcomings of binary attestation, a number of more flexible attestation mechanisms have been proposed.

BIND [21] tries to provide fine grained attestation by not verifying the complete memory content of an application, but only the piece of the code that will be executed. On top of that it allows to include the data that the code produces in the attestation data. The solution requires the attestation service to run in a more privileged execution environment and the integrity of the service is measured using the TPM.

In [9] the concept of semantic remote attestation is proposed. This is also a hybrid attestation scheme, where a virtual machine is attested by the TPM and the trusted virtual machine will certify certain semantic properties of the running program.

Property based attestation [15] takes a similar approach where properties of the platform and/or applications are reported instead of hash values of the binary images. One practical proposal is to use delegation based property attestation: a certification agency certifies a mapping between properties and configurations and publishes these property certificates.

All these solutions require the attestation service to run in a secure execution environment. As a consequence they can not easily be implemented in legacy operating systems.

### 3 Remote attestation on legacy platforms

#### 3.1 Checksum functions

A widely implemented technique in software tamper resistance is the use of checksum functions (e.g., in software guards) [1]. These functions read (a part of) the software code as input. If the output does not correspond to a pre-computed value, tampering is detected. However, using the memory copy attack by Wurster *et al.* [23,24], these checks can be easily circumvented. An adversary can distinguish if code instructions are interpreted or read (e.g., as input to a checksum function). Hence, tamper detection can be fooled when reading of code is redirected to an untampered copy, although a tampered copy is executed.

Two techniques to detect memory copy attack have been proposed. A first approach is the measurement of the execution time of the verification function. Memory copy attacks introduce some levels of indirection, which imply extra computations that slow down the execution, and this behavior can be detected.

A second option is the usage of self modifying code to detect a memory copy attack [8]. If the verification function modifies itself, only the clean (i.e., untampered) memory copy, where memory reads/writes are pointed to, will be updated. Doing so, a verifier can notice that the execution, i.e., running the unmodified tampered copy, has not been changed, and thus detect the attack.

#### 3.2 Pioneer

In [19] Seshadri *et al.* describe a remote attestation solution for embedded devices, without the need for a trusted platform module. Latter, they proposed a remote code integrity verification solution for legacy systems, called Pioneer [17,18]. It consists of a two-stage challenge-response protocol. First, the verifier obtains an assurance that a verification agent is present on the untrusted host. Next, this verification agent reports the integrity of the executable the verifier is interested in. The detailed steps in the protocol are depicted in Figure 2.

- (i) The verifier invokes a verification agent  $V$  on the untrusted host by sending a challenge  $n$ , and starts timing its execution:  $t_1 \leftarrow t_{current}$ .
- (ii) This challenge is used as a seed for a pseudo-random walk through the memory of the verification agent. Based on this walk, a checksum is computed:  $c \leftarrow cksum(n, V)$ .
- (iii) The verification agent reports the checksum  $c$  to the verifier. The verifier can now check the integrity of verification agent by verifying that two conditions are satisfied:
  - (a) the fingerprint of the verification agent is delivered in time ( $t_2 \leftarrow t_{current}$ ),

i.e., the verifier knows an upper bound on the expected execution time of the checksum calculation:  $t_2 - t_1 < \Delta t_{expected} = \Delta t_{cksum} + \Delta t_{network} + \delta t$ , with  $\Delta t_{cksum}$  the expected execution time of the checksum function,  $\Delta t_{network}$  the network delay, and  $\delta t$  some margin; and

- (b) the checksum should correspond with the value that the verifier has calculated on its own local copy of the verification agent.
- (iv) The verification agent computes a cryptographic hash of the executable  $E$  as a function of the original nonce:  $h \leftarrow hash(n, E)$ .
- (v) This hash is sent to and verified by the verifier. Again the verifier needs to independently perform the same computation on a local copy of the executable.
- (vi) The verification agent invokes the application  $E$  and transfer control to it.

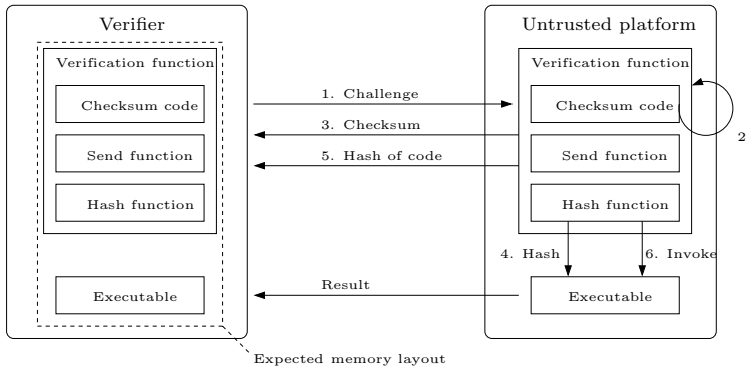


Fig. 1. Overview of the challenge-response system of Pioneer

When an adversary attempts to produce a correct checksum while running tampered code, this should be detectable due to an execution slowdown. In Pioneer, when a memory copy attack is deployed, an execution slowdown is caused by incorporating the Program Counter value and/or the Data Pointer value into the checksum computation. Because an adversary needs to forge these values as well, this will lead to an increase in execution time.

However, the design of the checksum function  $cksum()$  is subject to several constraints:

- The checksum function should be execution time optimal. If an adversary would be able to optimize the checksum function, he would gain time to perform malicious actions.
- To maximize the adversary's overhead, the checksum function will read the memory in a pseudo-random traversal. This prevents the adversary from predicting the memory reads. The challenge  $n$  seeds the pseudo-random traversal.
- The execution time of the checksum function must be predictable. Hence, Pioneer needs to run in supervisor mode and with interrupts disabled.

Pioneer comes with a checksum function that is compliant with these constraints. However, three important assumptions need to be introduced.

First, the verifier needs to know the exact hardware configuration of the untrusted platform, including the CPU model, clock speed and memory latency, in order to compute the expected untampered execution time. If an adversary is able to replace or overclock the CPU, he could influence the execution time. Hence in the Pioneer system, it is assumed that the hardware configuration is known by the verification entity and cannot be changed.

Secondly, an adversary could act as a proxy, and ask a faster computing device to compute the checksum on his behalf. We call these *proxy attacks*. To avoid this, in the Pioneer protocol, it is assumed that there is an authenticated communication channel between the verification entity and the untrusted execution platform.

Finally, a general problem that remains is the network latency. Hence Pioneer assumes the verification entity to be located closely to the untrusted execution platform.

### 3.3 Timed Execution Agent Systems

Garay and Huelsbergen also rely in the time execution of a verification agent, in their Timed Execution Agent Systems (TEAS) [7]. Contrary to Pioneer, TEAS issues a challenge that is an obfuscated executable program potentially computing any function. As such, the verification function is mobile in TEAS, while Pioneer uses a single fixed verification function invoked by a random challenge. Hence the line of defense is not the time-optimal behavior of the verification function and the unpredictable memory traversal. Instead, for each challenge (verification agent) sent, the attacker has to reverse-engineer this agent (i.e., gain information of the checksum function used) within the expected time, to fool the verification entity.

This way, TEAS tries to address some of the shortcomings of Pioneer. Adversaries are delayed due to difficulties in reverse engineering, and the unpredictability of the verification agent. The verification entity still keeps track of execution time to detect (hardware assisted) memory copy attacks.

## 4 Remote attestation on legacy operating systems with trusted platform modules

Pure software approaches for remote attestation, relying on timed execution of a checksum function, impose a number of limitations. It is impossible to uniquely identify the platform, creating an opportunity for proxy attacks. To determine the expected execution time of the checksum computation, detailed knowledge about the processor of the untrusted platform is needed. The adversary will be tempted to replace the processor with a faster one such that the extra computing cycles can be used to tamper with the checksum function. The expected execution time can be unreliable because the verifier has to make a worst case assumption on the network latency, which can be rather unpredictable on the Internet.

Meanwhile, a lot of TCG enabled computers are sold. Trusted computing technology promises to solve the remote attestation problem by measuring integrity



(i.e., task of the CRTM) in the boot process and later reporting the measured configuration to an external verifier. The authenticity of this integrity report is guaranteed by a trustworthy hardware module, namely the TPM. To offer a solid solution trusted computing platforms require a secure operating system, typically based on a hypervisor or microkernel. If legacy operating system are used, the chain of trust can be subverted; e.g., by loading a malicious device driver or by exploiting a kernel level security vulnerability.

In this section, we proposed a mixed solution, to obtain the advantages of both software based attestation and TCG attestation techniques. We present a practical solution to remotely verify software, based on software-only solutions like Pioneer and TEAS, combined with limited support of trusted platform modules. As such, we can invalidate the strong assumptions (which are unrealistic in some deployment scenarios), but avoid the need to deploy an adapted bootloader and secure operating system.

#### 4.1 Local execution time measurement with TPMs

##### 4.1.1 TPM time stamping

Time stamping is one of the new features in version 1.2 of the TPM specification. The TPM provides the `TPM_TickStampBlob` command to create a time stamp on a blob:  $TS \leftarrow \text{Sign}_{SK}(\text{blob} || t || TSN)$  with  $SK$  a signature key,  $\text{blob}$  the digest to stamp,  $t$  the current time and  $TSN$  a nonce determined by the TPM. The time stamp  $TS$  does not include an actual universal time clock (UTC) value, but rather the number of timer ticks the TPM has counted since startup of the platform; therefore the functionality is sometimes called *tick stamping*. It is the responsibility of the caller to associate the ticks to an actual UTC time, which can be done in a similar way as online clock synchronization protocols.

The TPM counts ticks from the start of a timing session, which is identified with the Tick Session Nonce  $TSN$ . On a PC, the TPM may use the clock of the Low Pin Count (LPC) bus as timing source, but it may also have a separate clock circuit (e.g., with an internal crystal). At the beginning of a tick session the tick counter is reset to 0 and the session nonce  $TSN$  is randomly generated by the TPM. The beginning of a timing session is platform dependent. In laptops, the clock of the LPC bus, normally running at 33 MHz, can be stopped to save power, which could imply that the tick counter is stopped as well. Consequently it depends on the platform whether the TPM will have the ability to maintain the tick counter across power cycles or in different power modes on a platform.

According to the specification the tick counter will have a maximum resolution of 1  $\mu\text{s}$ , and the minimum resolution should be 1 ms. Initial experiments show that the Infineon 1.2 TPM has a resolution 1 ms and that the Atmel TPM clearly violates the TCG specification. Subsequential invocations of the `TPM_GetTicks` command give a tick count value that is incremented with 1; effectively the tick counter in the Atmel TPM behaves as a monotonic counter and not as a clock<sup>5</sup>! This is not the

<sup>5</sup> This behavior is valid in an older revision (64) of the 1.2 specification, where the TPM only needs to guarantee “that the clock value will increment at least once prior to the execution of any command”. Sending

first instance of non-compliance of TPM chips with the TCG specification [14].

#### 4.1.2 Improved Pioneer protocol

The Pioneer protocol can be improved by employing the tick stamping functionality of trusted platform modules.

- (i) The verifier sends a challenge  $n$  to the verification agent.
- (ii) The verification agent uses the TPM to create a tick stamp on this nonce:  $TS_1 \leftarrow \text{Sign}_{SK}(n || t_1 || TSN_1)$ . The result  $TS_1$  is sent to the verifier.
- (iii) The verification agent uses  $TS_1$  as seed for the pseudo-random walk through its memory, resulting in a fingerprint:  $c \leftarrow \text{cksum}(TS_1, V)$ .
- (iv) The calculated checksum gets time stamped by the TPM as well:  $TS_2 \leftarrow \text{Sign}_{SK}(c || t_2 || TSN_2)$ . This result  $TS_2$  gets reported to the verifier.
- (v) The verifier can now verify the integrity of the verification agent by performing the following steps:
  - (a) verify the two signatures  $TS_1$  and  $TS_2$  (at this stage the untrusted platform can be uniquely identified);
  - (b) check if  $TSN_1 = TSN_2$  (i.e., whether the TPM has been reset by a platform reboot or a hardware attack [10]);
  - (c) extract  $t_2 - t_1$  from the time stamps and check whether it corresponds with the expected execution time of the checksum function:  $t_2 - t_1 < \Delta t_{\text{expected}} = \Delta t_{\text{cksum}} + \Delta t_{\text{sign}} + \delta t$ , with  $\Delta t_{\text{cksum}}$  the expected execution time of the checksum function,  $\Delta t_{\text{sign}}$  the TPM signing duration, and  $\delta t$  the latency between the operations and bounds for TPM tick rounding.
  - (d) Check whether the checksum  $c$  corresponds with the value that the verifier has calculated on its own local copy of the verification agent.

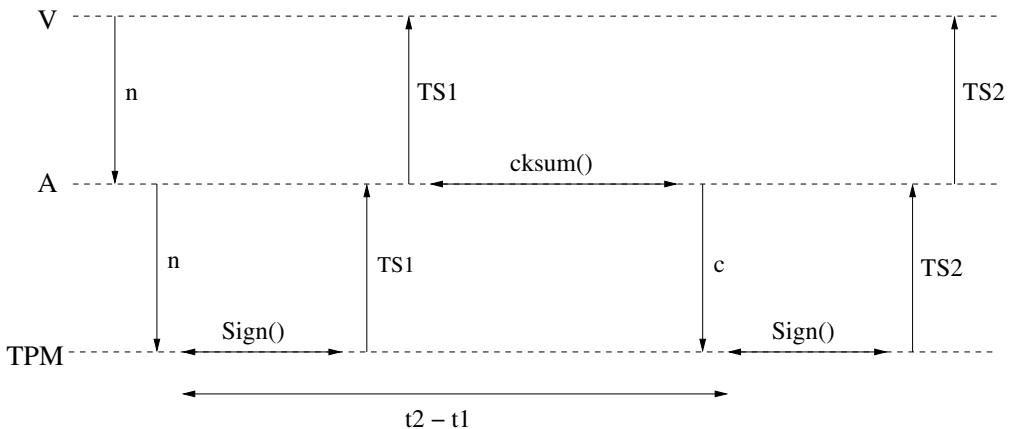


Fig. 2. Time Overview of the Improved Pioneer Protocol

---

other commands between two `TPM_GetTicks` requests, confirms that this is the tick counter increment on every command.

The advantage of this improved Pioneer protocol is that the timing is moved from the verifier to the verification agent on the untrusted platform. Consequently, the verifier does no longer need to take into account the (non-deterministic) network latency. Hence, the expected checksum computation time becomes more accurate. Because each TPM signs with its unique key, an authenticated channel can be established. If a verifier holds a database with the link between the TPM signing key, and the CPU specifications, he can take this into account to calculate the expected execution time of the checksum function. However, an adversary is still able to replace the CPU or install faster memory.

In order to deploy this system, only a TPM and driver (available for Linux, Mac, and Windows) need to be installed on the untrusted platform. There is no need for an adapted bootloader or adapted operating system, because it does not rely on TCG attestation.

#### *4.2 Configuration identification with trusted bootloader*

The solution can be further improved, if the TPM is used to report the processor specification. In this way some hardware attacks, where the processor or/and the memory get replaced by faster ones, can be detected during attestation. To achieve this extra feature, we propose to modify the bootloader. Bootloaders tend to be a lot smaller, and hence more trustworthy, than legacy operating systems: the OSLO bootloader [10] for instance is around 1000 lines of code, while a Linux 2.6 kernel contains more than 6 million lines of code. The integrity of the enhanced bootloader can be reported using standard TCG functionality. We still rely on timed execution to detect the compromise of legacy operating systems, given that the correct processor specification is known.

##### *4.2.1 Processor identification*

A first approach is to enhance the trusted bootloader to report the processor identifier to the TPM. Pentium class processors for instance have a CPUID instruction which returns the vendor ID (e.g., Intel or AMD), stepping, model, and family information, cache size, clock frequency, presence of features (like MMX/SSE), etc. All this information needs to be stored in the Stored Measurement Log and its hash should be extended to one of the Platform Configuration Registers. Before the improved Pioneer protocol is performed, the TPM will attest that the trusted bootloader is loaded correctly (i.e., its hash is stored in a certain PCR) and identifies the processor by digitally signing the PCR register containing the hashed processor identifier.

This mechanism allows to detect processor replacement and simulation, because the expected execution time will depend on the processor identification. On the other hand, this scheme can not cope with memory replacement (i.e., upgrading RAM with lower latency).

#### 4.2.2 Run-time checksum performance measurement

Another strategy is to run some performance measurement code during the startup of the platform. The bootloader could be adapted to run the Pioneer checksum function with a locally generated challenge (i.e., produced by the TPM RNG) and measure the required execution time. This timing can be measured accurately with the CPU cycle counter (e.g., RDTSC instruction in case of Pentium class CPUs) or with lower precision using the TPM time stamping mechanism described earlier. The trusted bootloader will report this performance measurement to the TPM, which later can sign the recorded value; again stored in a PCR register and logged in the SML.

This technique can provide the verifier a very accurate expectation of the checksum function's execution time. During the attestation phase, the verifier can rely on the timing information determined by trusted bootloader. Both processor and memory changes can be successfully and efficiently detected in this way.

#### 4.2.3 Proxy attacks

Although this protocol addresses a great deal of the issues raised in Pioneer, it still remains vulnerable to a proxy attack. A slow computer with TPM can send its timestamp  $TS_1$  to a fast computer that computes the checksum results. This result  $c$  is sent back to the slow machine that provides a signed attestation  $TS_2$  to the verifier. The network delay is captured by the computation profit. We provide two possible strategies to address this attack.

In the original protocol, a checksum is computed over the memory of the verification function, which includes the send function. The verification agent can be modified to only accept messages from the verifier, based on the IP or MAC address. However, these addresses can be spoofed.

Similarly, the verification agent also contains a function to communicate with the TPM. If the checksum function is computed over this function too, then there is a guarantee that there is only one way to invoke the verification agent.

## 5 Conclusion

At the moment commercially available operating system only offer limited trusted computing support. At most they provide a TPM device driver, a TCG Software Stack and/or a TPM-aware bootloader. This however is insufficient to achieve remote attestation of individual applications. In the meantime, pure software based attestation schemes have been proposed for legacy platforms. They rely on the timed execution of a checksum function, that computes an application fingerprint. The execution time is measured remotely by the verifier, imposing heavy assumptions that are difficult to achieve in practice.

In this work, we have proposed improvements for these software based attestation protocols. By using the time stamping functionality of a TPM, the execution time of the fingerprint computation can be measured locally. This also allows to uniquely identify the platform that is being verified. The solution can be further

strengthen with a trusted bootloader. This bootloader can identify the processor specification of the untrusted platform and provide accurate timing information about the checksum function.

## References

- [1] Aucsmith, D., *Tamper Resistant Software: An Implementation*, in: R. J. Anderson, editor, *Information Hiding, First International Workshop, Cambridge, U.K., May 30 - June 1, 1996, Proceedings*, Lecture Notes in Computer Science **1174** (1996), pp. 317–333.
- [2] Balacheff, B., L. Chen, S. Pearson, D. Plaquin and G. Proudler, “Trusted Computing Platforms: TCPA Technology in Context,” Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.
- [3] Brickell, E. F., J. Camenisch and L. Chen, *Direct anonymous attestation*, in: V. Atluri, B. Pfitzmann and P. D. McDaniel, editors, *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004* (2004), pp. 132–145.
- [4] Ceccato, M., M. D. Preda, J. Nagra, C. Collberg and P. Tonella, *Barrier Slicing for Remote Software Trusting*, in: *7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM’07), September 30 - October 1, Paris, France, 2007*.
- [5] Dvir, O., M. Herlihy and N. N. Shavit, *Virtual Leashing: Internet-Based Software Piracy Protection*, in: *ICDCS’05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS’05)* (2005), pp. 283–292.
- [6] England, P., B. W. Lampson, J. Manferdelli, M. Peinado and B. Willman, *A Trusted Open Platform*, *IEEE Computer* **36** (2003), pp. 55–62.
- [7] Garay, J. A. and L. Huelsbergen, *Software Integrity Protection Using Timed Executable Agents*, in: F.-C. Lin, D.-T. Lee, B.-S. Lin, S. Shieh and S. Jajodia, editors, *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2006, Taipei, Taiwan, March 21-24, 2006* (2006), pp. 189–200.
- [8] Giffin, J. T., M. Christodorescu and L. Kruger, *Strengthening software self-checksumming via self-modifying code*, in: *21st Annual Computer Security Applications Conference (ACSAC 2005), 5-9 December 2005, Tucson, AZ, USA* (2005), pp. 23–32.
- [9] Haldar, V., D. Chandra and M. Franz, *Semantic Remote Attestation – Virtual Machine Directed Approach to Trusted Computing*, in: *Proceedings of the 3rd Virtual Machine Research and Technology Symposium, May 6-7, 2004, San Jose, CA, USA* (2004), pp. 29–41.
- [10] Kauer, B., *OSLO: Improving the Security of Trusted Computing*, in: *Proceedings of the 16th USENIX Security Symposium, August 6-10, 2007, Boston, MA, USA* (2007).
- [11] Kennell, R. and L. H. Jamieson, *Establishing the Genuinity of Remote Computer Systems*, in: *Proceedings of the 12th USENIX Security Symposium, August 4-8, 2003, Washington, DC, USA* (2003), pp. 295–308.
- [12] Monroe, F., P. Wyckoff and A. D. Rubin, *Distributed Execution with Remote Audit*, in: *Proceedings of the Network and Distributed System Security Symposium, NDSS 1999, San Diego, California, USA* (1999), pp. 103–113.
- [13] Peinado, M., Y. Chen, P. England and J. Manferdelli, *NGSCB: A Trusted Open System*, in: H. Wang, J. Pieprzyk and V. Varadharajan, editors, *Information Security and Privacy: 9th Australasian Conference, ACISP 2004, Sydney, Australia, July 13-15, 2004. Proceedings*, Lecture Notes in Computer Science **3108** (2004), pp. 86–97.
- [14] Sadeghi, A.-R., M. Selhorst, C. Stübke, C. Wachsmann and M. Winandy, *TCG inside?: A Note on TPM Specification Compliance*, in: *STC’06: Proceedings of the first ACM workshop on Scalable trusted computing* (2006), pp. 47–56.
- [15] Sadeghi, A.-R. and C. Stübke, *Property-based attestation for computing platforms: caring about properties, not mechanisms.*, in: C. Hempelmann and V. Raskin, editors, *Proceedings of the New Security Paradigms Workshop 2004, September 20-23, 2004, Nova Scotia, Canada* (2004), pp. 67–77.
- [16] Sailer, R., X. Zhang, T. Jaeger and L. van Doorn, *Design and Implementation of a TCG-based Integrity Measurement Architecture*, in: *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA* (2004), pp. 223–238.

- [17] Seshadri, A., M. Luk, A. Perrig, L. van Doorn and P. K. Khosla, *Externally Verifiable Code Execution*, *Communications of the ACM* **49** (2006), pp. 45–49.
- [18] Seshadri, A., M. Luk, E. Shi, A. Perrig, L. van Doorn and P. K. Khosla, *Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems*, in: A. Herbert and K. P. Birman, editors, *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSOP 2005, Brighton, UK, October 23-26, 2005* (2005), pp. 1–16.
- [19] Seshadri, A., A. Perrig, L. van Doorn and P. K. Khosla, *SWATT: SoftWare-based ATTestation for Embedded Devices*, in: *2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA* (2004), pp. 272–282.
- [20] Shankar, U., M. Chew and J. D. Tygar, *Side Effects Are Not Sufficient to Authenticate Software*, in: *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA* (2004), pp. 89–102.
- [21] Shi, E., A. Perrig and L. van Doorn, *BIND: A Fine-Grained Attestation Service for Secure Distributed Systems*, in: *2005 IEEE Symposium on Security and Privacy (S&P 2005), 8-11 May 2005, Oakland, CA, USA* (2005), pp. 154–168.
- [22] Tan, G., Y. Chen and M. H. Jakubowski, *Delayed and Controlled Failures in Tamper-Resistant Systems*, in: *8th Information Hiding, Lecture Notes in Computer Science* **4437** (2006), pp. 216–231.
- [23] van Oorschot, P. C., A. Somayaji and G. Wurster, *Hardware-Assisted Circumvention of Self-Hashing Software Tamper Resistance*, *IEEE Transactions on Dependable and Secure Computing* **2** (2005), pp. 82–92.
- [24] Wurster, G., P. C. van Oorschot and A. Somayaji, *A Generic Attack on Checksumming-Based Software Tamper Resistance*, in: *2005 IEEE Symposium on Security and Privacy (S&P 2005), 8-11 May 2005, Oakland, CA, USA* (2005), pp. 127–138.
- [25] Zhang, X. and R. Gupta, *Hiding Program Slices for Software Security*, in: *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003), 23-26 March 2003, San Francisco, CA, USA* (2003), pp. 325–336.