Science of Computer Programming 74 (2008) 13–22

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Remote attestation on legacy operating systems with trusted platform modules☆

Dries Schellekens*, Brecht Wyseur, Bart Preneel

Katholieke Universiteit Leuven, Department ESAT/SCD-COSIC, Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium
Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium

ARTICLE INFO

Article history:
Received 1 April 2008
Accepted 3 September 2008
Available online 23 September 2008

Keywords:
Remote software authentication
Attestation
Trusted platform module
Timed execution

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 (TPM), in combination with a timing-based remote code integrity verification mechanism. In this way, the overall security of the timed execution scheme can be improved without requiring a secure operating system.

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, online 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).

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 guarantee 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 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

☆ This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), by the FWO project G.0300.07 (Security components for trusted computer systems), and in part by the European Commission through the IST Programme under Contract IST-027635 OPEN_TC and IST-021186 RE-TRUST.

* Corresponding author at: Katholieke Universiteit Leuven, Department ESAT/SCD-COSIC, Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
E-mail address: dries.schellekens@esat.kuleuven.be (D. Schellekens).

0167-6423/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2008.09.005

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* [26]. 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 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

So called genuinity tests [13] have been developed to verify if the hardware is real and if 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 [24].

The Pioneer system proposed in [21,22] establishes whether software on an untrusted host is untampered by calculating a checksum over its runtime memory image. If the resulting checksum is not reported within a defined time frame, the verifier assumes that the checksum function itself has been altered. This solution works on embedded systems [23] and legacy Personal Computer (PC) systems, but it relies on some strong assumptions.

Other proposals 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 [16] 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 [4,5,29]. This is a form of server side execution.

A completely different approach is to introduce hardware tailored specifically to provide assurance on an untrusted platform. The Trusted Computing Group (TCG) defines the addition of a Trusted Platform Module (TPM). A trusted computing platform reliably measures the software (i.e., BIOS, bootloader, operating system components and user applications) that get loaded during startup of the platform [20], and can later report its configuration to a remote entity with an attestation protocol.

The TCG attestation process has deployment issues [19] and requires a secure operating system. Virtualization technology like Intel Trusted Execution Technology (TXT) [10] can overcome some of these shortcomings [8,11,15].

1.2. Our contributions

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 today. To offer a solid solution, trusted computing platforms require a secure operating system. If legacy operating systems are used, the chain of trust can be easily subverted; e.g., by loading a malicious device driver or by exploiting a kernel level security vulnerability.

Given these two observations we propose to improve the Pioneer scheme by using the time stamping functionality of the TPM and additionally some minor changes to the bootloader. Our solution does not rely on a secure operating system or a trusted virtualization layer.

1.3. Outline of paper

In Section 2 we focus on the attestation functionality provided by trusted computing platforms. Purely software- based attestation techniques are discussed in Section 3. We describe how to enhance the latter schemes with the time stamping feature of a TPM and with a trusted bootloader, in Sections 4 and 5, respectively. Section 6 concludes our results.

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) [6,17]. 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[1].

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.

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[2]: $PCR_{new} \leftarrow Hash(PCR_{old}||M)$ 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. 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 ($PCR_0, \ldots, PCR_n$) 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.

TCG technology also has the concept of *sealing*, enabling certain data or keys to be cryptographically bound to a certain platform configuration. The TPM will only release this data if a given configuration is booted. This can be considered as an implicit form of attestation: the application that needs to be externally verified, can seal a secret in the TPM and consequently, if the application is able to unseal the secret, the platform is known to be in a given state.

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. However, establishing a chain of trust to individual programs is not straightforward in practice.

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 these 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* (TCB) [15] 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 hardware virtualization support to achieve both security and backward compatibility [8].

[1] All TCG specifications are available on https://www.trustedcomputinggroup.org.

[2] The version 1.2 specification introduces a number of PCRs can be reset by higher privileged (determined by locality) code [10].

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 [20]. 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 subsequently measure 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 runtime 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 [25] 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 [11] 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 [19] 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 [14].

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

3. Software-based attestation on legacy platforms

In this section we present two software-based attestation solution that rely on the timed execution of a checksum function: Pioneer [21,22] and TEAS [7].

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.* [27,28], these checks can be easily circumvented. An adversary can distinguish if code instructions are interpreted or if they are 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 [9]. 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 [23] Seshadri *et al.* describe a remote attestation solution for embedded devices, without the need for a trusted platform module. Later, they proposed an adapted solution for legacy PC systems, called Pioneer [21,22]. 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 like in TCG attestation.

3.2.1. Protocol description

The detailed steps of the protocol are depicted in Fig. 1.

(1) The verifier invokes a verification agent V on the untrusted host by sending a challenge n, and starts timing its execution: $t_1 \leftarrow t_{current}$.

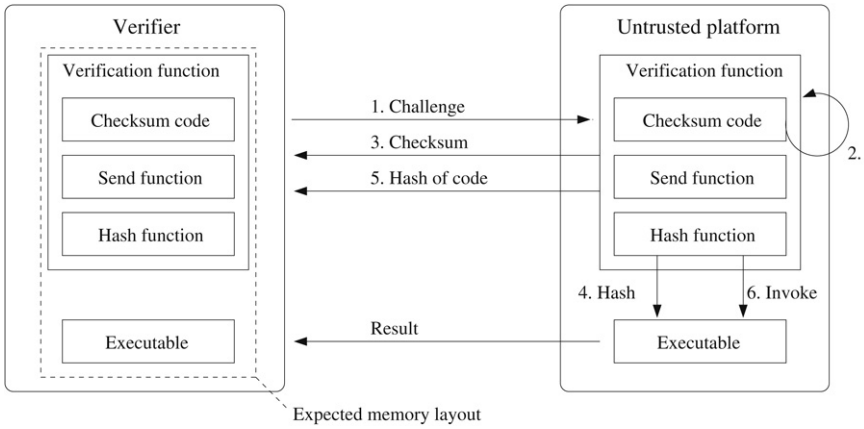

Fig. 1. Schematic overview of the Pioneer protocol.

(2) 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: $c \leftarrow cksum(n, V)$.
(3) The verification agent reports the checksum c to the verifier. The verifier can now check the integrity of the 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 Δt_{cksum} the expected execution time of the checksum function, $\Delta t_{network}$ the network delay, and δt 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.
(4) The verification agent computes a cryptographic hash of the executable E as a function of the original nonce: $h \leftarrow Hash(n, E)$.
(5) 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.
(6) The verification agent invokes the application E and transfer control to it.

3.2.2. Checksum function

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*() in Pioneer was 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 beforehand. 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.

3.2.3. Shortcomings

The security of the Pioneer solution relies on three important assumptions.

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 (TEAS)

Garay and Huelsbergen also rely on the time execution of a verification agent in their Timed Executable 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 agent is mobile in TEAS, while Pioneer uses a single fixed verification function invoked by a random challenge.

The motivation is that an attacker has to reverse-engineer the obfuscated and unpredictable agent (i.e., gain information of the checksum function used) within the expected time, in order to fool the verification entity. It should be noted that the verification entity still has to keep track of execution time to detect hardware assisted memory copy attacks.

4. Local execution time measurement with TPMs

In this section, we describe the time stamping feature of TPMs. This functionality can be used to enhance software-based attestation schemes that rely in timed execution. As such, we can invalidate the strong assumptions of these schemes, which are unrealistic in some deployment scenarios (see Section 3), but avoid the need for a secure operating system.

4.1. TPM time stamping

Time stamping is one of the new features in version 1.2 of the TPM specification. The TPM can create a time stamp on a blob: $TS \leftarrow Sign_{SK}(blob||t||TSN)$ with SK a signature key, $blob$ 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.

4.1.1. Tick session

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 oscillator). 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 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.

4.1.2. Tick counter resolution

According to the specification the tick counter will have a maximum resolution of 1 μs, and the minimum resolution should be 1 ms. Initial experiments (see Appendix for example program) 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[3]! This is not the first instance of non-compliance of TPM chips with the TCG specification [18].

4.2. Improved pioneer protocol

The Pioneer protocol can be improved by employing the tick stamping functionality described above (see Fig. 2).

(1) The verifier sends a challenge n to the verification agent.
(2) The verification agent uses the TPM to create a tick stamp on this nonce: $TS_1 \leftarrow Sign_{SK}(n||t_1||TSN_1)$. The result TS_1 is sent to the verifier.
(3) The verification agent uses TS_1 as seed for the pseudo-random walk through its memory, resulting in a fingerprint: $c \leftarrow cksum(TS_1, V)$.
(4) The calculated checksum gets time stamped by the TPM as well: $TS_2 \leftarrow Sign_{SK}(c||t_2||TSN_2)$. This result TS_2 gets reported to the verifier.
(5) 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 [12]);

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

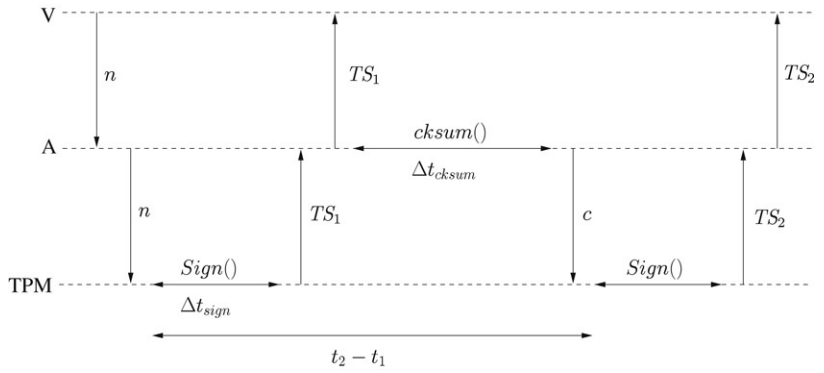

Fig. 2. Time overview of the improved Pioneer protocol.

(c) extract $t_2 - t_1$ from the time stamps and check whether it corresponds with the expected execution time of the checksum function:
$$t_2 - t_1 < \Delta t_{expected} = \Delta t_{cksum} + \Delta t_{sign} + \delta t$$
with Δt_{cksum} the expected execution time of the checksum function, Δt_{sign} the TPM signing duration, and δt 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.

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. Instead the verifier has to know the duration of a TPM signature generation Δt_{sign}, which will depend on the actual TPM used. We expect that this time is constant. Otherwise the TPM would be trivially vulnerable to a timing analysis. Hence, the total expected computation time $\Delta t_{expected}$ can be estimated accurately.

Because each TPM signs with its unique key *SK*, an authenticated channel is established. If a verifier holds a database that links the TPM signing key to the CPU specification of the platform, he can take this into account to estimate the expected execution time of the checksum function (i.e., Δt_{cksum}). It should be noted that the length of the pseudo-random walk calculated by *cksum*() has to be sufficiently large as the resolution of the TPM tick counter is limited.

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 operating system, because it does not rely on TCG attestation. However, an adversary is still able to replace the CPU or install faster memory. In Section 5 we will address this concern with an adapted bootloader.

4.3. 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 time stamp 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 Pioneer 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. Configuration identification with trusted bootloader

The solution can be further improved by using the TPM 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 [12] for instance is around 1000 lines of code[4], 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] The OSLO bootloader [12] uses the AMD SKINIT instruction to create a dynamic root of trust for measurement (DRTM). This has the advantage that the – potentially untrusted – BIOS is not included in the chain of trust.

5.1. 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[5]. 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).

5.2. Runtime 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 (e.g., produced by the TPM random number generator) and measure the required execution time. This timing can be measured accurately with the CPU cycle counter (i.e., 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 benchmark 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.

6. Conclusion

At the moment, commercially available operating systems 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.

Appendix. Program to Read TPM Tick Counter

A.1. TPM Commands

The TPM has two low level commands to get the current tick counter value.

- `TPM_GetTicks` returns a `TPM_CURRENT_TICKS` structure, which contains the current tick counter value.
- `TPM_TickStampBlob` applies a time stamp to the passed blob of 20 byte. Other input parameters are a key handle pointing to a signature key, the authorization data for that key and an anti replay value. The command returns the Tick Stamp Result (i.e., a signature on the blob concatenated with the current tick count value, the anti replay nonce and some fixed text) and the `TPM_CURRENT_TICKS` structure.

The `TPM_CURRENT_TICKS` structure has the following fields:

Tick Count Value: The number of ticks since the start of this tick session, represented with a 64 bit counter.
Tick Increment Rate: The rate at which the tick counter is incremented expressed in μs.
Tick Session Nonce: The 20 byte nonce created by the TPM when resetting the tick counter.

The TCG Software Stack provides higher level functions to application, namely `Tspi_TPM_ReadCurrentTicks` and `Tspi_Hash_TickStampBlob`.

[5] If the OSLO bootloader is used, a resettable PCR can be used to protect against a platform reset attack [12].

A.2. C Code

```
/*
 * \brief   Jueterborg - call a TPM_GetTicks()
 * \date    2007-06-27
 * \author  Bernhard Kauer <kauer@tudos.org>
 */
/*
 * Copyright (C) 2007  Bernhard Kauer <kauer@tudos.org>
 * Technische Universitaet Dresden, Operating Systems Research Group
 *
 * This file is part of the OSLO package, which is distributed under
 * the  terms  of the  GNU General Public Licence 2.  Please see the
 * COPYING file for details.
 */

#include <sys/types.h>
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>

#define ERROR(err, res, format, ...) if (!(res))         \
    {                                                    \
      fprintf(stderr, "ERROR: ");                        \
      fprintf(stderr, format, ##__VA_ARGS__);            \
      fprintf(stderr, "\n");                             \
      return err;                                        \
    }

int
tpm_transmit(unsigned char *buf, unsigned inlen, int outlen)
{
  int fd;
  int res;

  ERROR(-10, -1 != (fd = open("/dev/tpm0", O_RDWR)), "could not open tpm");
  ERROR(-11, inlen == write(fd, buf, inlen), "could not write to the tpm");
  res = read(fd, buf, outlen);
  ERROR(-12, 0 == close(fd), "could not close the fd");
  return res;
}

/**
 * Call a TPM_GetTicks() and print the output.
 */
int main(int argc, char **argv)
{
  int i;
 unsigned char buffer[2+4+4+32] = {0x00, 0xC1, 0x00, 0x00, 0x00, 0x0a, 0x00, 0x00, 0x00, 0xf1};

  i = tpm_transmit(buffer, 10, 42);

  printf("TPM_GetTicks()\n");
  printf("Tag:\t%x\n", htons(*(short *)(buffer+10)));
  printf("Count:\t%llx\n", *(unsigned long long *)(buffer+12));
  printf("Rate:\t%x\n", htons(*(short *)(buffer+20)));
  printf("Nonce:\t");
  for (i=0; i<20; i++)
    printf("%02x", buffer[22+i]);
  printf("\n");
```

```
  ERROR(12, (10 != i && i != 42), "could not send to the TPM");
  ERROR(13, 0xC400 == *(unsigned short *) buffer, "response code mismatch");
  ERROR(15, 0 == *(unsigned *) (buffer+6), "error: %d", htonl(*(unsigned *) (buffer+6)));
 ERROR(14, 0x2a000000 == *(unsigned *) (buffer+2), "size mismatch %x",*(unsigned *)(buffer + 2));

  return 0;
}
```

References

[1] D. Aucsmith, Tamper resistant software: An implementation, in: R.J. Anderson (Ed.), First International Workshop on Information Hiding, Cambridge, U.K., May 30–June 1, 1996, in: Lecture Notes in Computer Science, vol. 1174, Springer, 1996.
[2] B. Balacheff, L. Chen, S. Pearson, D. Plaquin, G. Proudler, Trusted Computing Platforms: TCPA Technology in Context, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.
[3] E.F. Brickell, J. Camenisch, L. Chen, Direct anonymous attestation, in: V. Atluri, B. Pfitzmann, P.D. McDaniel (Eds.), 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25–29, 2004, ACM, 2004.
[4] M. Ceccato, M.D. Preda, J. Nagra, C. Collberg, 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] O. Dvir, M. Herlihy, N. Shavit, Virtual leashing: Internet-based software piracy protection, in: 25th International Conference on Distributed Computing Systems, ICDCS 2005, 6–10 June 2005, Columbus, OH, USA, IEEE Computer Society, 2005.
[6] P. England, B.W. Lampson, J. Manferdelli, M. Peinado, B. Willman, A trusted open platform, IEEE Computer 36 (7) (2003) 55–62.
[7] J.A. Garay, L. Huelsbergen, Software integrity protection using timed executable agents, in: F.-C. Lin, D.-T. Lee, B.-S. Lin, S. Shieh, S. Jajodia (Eds.), 2006 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2006, Taipei, Taiwan, March 21–24, 2006, ACM, 2006.
[8] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh, Terra: A virtual machine-based platform for trusted computing, in: 19th Symposium on Operating System Principles, October 19–22, 2003, Bolton Landing, NY, USA, 2003.
[9] J.T. Giffin, M. Christodorescu, 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, IEEE Computer Society, 2005.
[10] D. Grawrock, The Intel Safer Computing Initiative — Building Blocks for Trusted Computing, Intel Press, 2006.
[11] V. Haldar, D. Chandra, M. Franz, Semantic remote attestation — virtual machine directed approach to trusted computing, in: 3rd Virtual Machine Research and Technology Symposium, May 6–7, 2004, San Jose, CA, USA, USENIX, 2004.
[12] B. Kauer, OSLO: Improving the security of trusted computing, in: 16th USENIX Security Symposium, August 6–10, 2007, Boston, MA, USA, USENIX, 2007.
[13] R. Kennell, L.H. Jamieson, Establishing the genuinity of remote computer systems, in: 12th USENIX Security Symposium, August 4–8, 2003, Washington, DC, USA, USENIX, 2003.
[14] U. Kühn, M. Selhorst, C. Stüble, Realizing property-based attestation and sealing with commonly available hard- and software, in: 2007 ACM Workshop on Scalable Trusted Computing, STC'07, ACM, New York, NY, USA, 2007.
[15] J.M. McCune, B. Parno, A. Perrig, M.K. Reiter, H. Isozaki, Flicker: An execution infrastructure for TCB minimization, in: ACM European Conference in Computer Systems, EuroSys 2008, 2008.
[16] F. Monrose, P. Wyckoff, A.D. Rubin, Distributed execution with remote audit, in: Network and Distributed System Security Symposium, NDSS 1999, San Diego, CA, USA, The Internet Society, 1999.
[17] M. Peinado, Y. Chen, P. England, J. Manferdelli, NGSCB: A trusted open system, in: H. Wang, J. Pieprzyk, V. Varadharajan (Eds.), 9th Australasian Conference on Information Security and Privacy, ACISP 2004, Sydney, Australia, July 13–15, 2004, in: Lecture Notes in Computer Science, vol. 3108, Springer, 2004.
[18] A.-R. Sadeghi, M. Selhorst, C. Stüble, C. Wachsmann, M. Winandy, TCG inside?: A note on TPM specification compliance, in: First ACM Workshop on Scalable Trusted Computing, STC'06, ACM Press, New York, NY, USA, 2006.
[19] A.-R. Sadeghi, C. Stüble, Property-based attestation for computing platforms: Caring about properties, not mechanisms, in: C. Hempelmann, V. Raskin (Eds.), New Security Paradigms Workshop 2004, September 20–23, 2004, Nova Scotia, Canada, ACM, 2004.
[20] R. Sailer, X. Zhang, T. Jaeger, L. van Doorn, Design and implementation of a TCG-based integrity measurement architecture, in: 13th USENIX Security Symposium, August 9–13, 2004, San Diego, CA, USA, USENIX, 2004.
[21] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, P.K. Khosla, Externally verifiable code execution, Commununications of the ACM 49 (9) (2006) 45–49.
[22] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, P.K. Khosla, Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems, in: A. Herbert, K.P. Birman (Eds.), 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23–26, 2005, ACM, 2005.
[23] A. Seshadri, A. Perrig, L. van Doorn, 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, IEEE Computer Society, 2004.
[24] U. Shankar, M. Chew, J.D. Tygar, Side effects are not sufficient to authenticate software, in: 13th USENIX Security Symposium, August 9–13, 2004, San Diego, CA, USA, USENIX, 2004.
[25] E. Shi, A. Perrig, 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, IEEE Computer Society, 2005.
[26] G. Tan, Y. Chen, M.H. Jakubowski, Delayed and controlled failures in tamper-resistant systems, in: 8th Information Hiding, in: Lecture Notes in Computer Science, LNCS, vol. 4437, 2006.
[27] P.C. van Oorschot, A. Somayaji, G. Wurster, Hardware-assisted circumvention of self-hashing software tamper resistance, IEEE Transactions on Dependable and Secure Computing 2 (2) (2005) 82–92.
[28] G. Wurster, P.C. van Oorschot, 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, IEEE Computer Society, 2005.
[29] X. Zhang, 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, IEEE Computer Society, 2003.